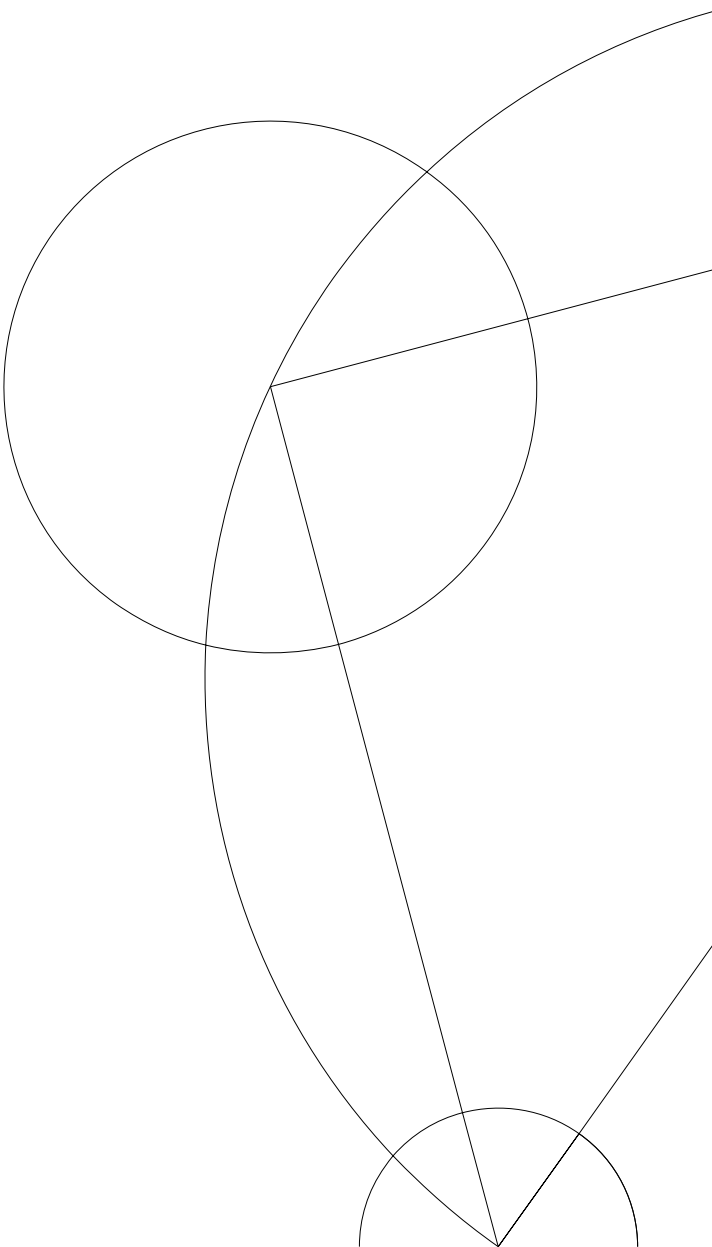
UNIVERSITY OF COPENHAGEN

PhD dissertation

Patrick Bahr

Modular Implementation of Programming Languages and a Partial-Order Approach to Infinitary Rewriting



Academic advisor: Fritz Henglein

Submitted: October 31, 2012

Modular Implementation of Programming Languages and a Partial-Order Approach to Infinitary Rewriting

Patrick Bahr
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

October 31, 2012

PhD Thesis

This thesis has been submitted to the PhD School of Science, Faculty of Science, University of Copenhagen, Denmark

Author:	Patrick Bahr
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark

Title:	Modular Implementation of Programming Languages and a Partial-Order Approach to Infinitary Rewriting
Academic advisor:	Fritz Henglein
Submitted:	October 31, 2012

Short abstract

In this dissertation we investigate two independent areas of research.

In the first part, we develop techniques for implementing programming languages in a modular fashion. Within this problem domain, we focus on operations on typed abstract syntax trees with the goal of developing a framework that facilitates the definition, manipulation and composition of such operations. The result of our work is a comprehensive combinator library that provides these facilities. What sets our approach apart is the use of recursion schemes derived from *tree automata* in order to implement operations on abstract syntax trees.

The second part is concerned with *infinitary rewriting*, a field that studies transfinite rewrite sequences. We extend the established theory of infinitary rewriting in two directions: (1) a novel approach to convergence in infinitary rewriting that replaces convergence in a metric space with the limit inferior in a partially ordered set; (2) extending infinitary term rewriting to infinitary term *graph* rewriting. We show correspondences between the established calculi based on metric convergence and the newly developed calculi based on partial orders. Moreover, we show the advantages of our partial order approach in terms of better confluence and normalisation properties of infinitary term rewriting as well as in terms of better completeness properties for infinitary term graph rewriting.

Abstract

This dissertation is a collection of nine research papers pertaining to two independent areas of research.

In the first part, entitled *Modular Implementation of Programming Languages*, we develop techniques for implementing programming languages in a modular fashion. Within this problem domain, we focus on operations on typed abstract syntax trees with the goal of developing a framework that facilitates the definition, manipulation and composition of such operations. The result of our work is a comprehensive combinator library that provides these facilities.

What sets our approach apart is the use of recursion schemes derived from *tree automata* in order to implement operations on abstract syntax trees. In the first two papers we illustrate the power of this approach by showcasing *tree homomorphisms* – a very limited form of tree automata – as basic building blocks for simple tree transformations. Their simplicity allows us to combine them with monadic effects, manipulate and combine them in a flexible manner, and perform optimisations in the form of *deforestation*. In the third paper, we move to more powerful tree automata. Usually, these more powerful automata are cumbersome to define as they combine different computational aspects. We show, however, that these automata can be constructed from simpler ones, viz. tree homomorphisms and simple state machines. In the second paper, we focus on the important issue of representing variable names and variable binders using a carefully restricted form of *higher-order abstract syntax*.

In the fourth paper, we presents a comprehensive and realistic application of our library: a prototype implementation of a novel *enterprise resource planning* system built around a family of domain-specific languages that make it possible to customise the system in a highly flexible manner. The system combines several highly integrated domain-specific languages, which are implemented using our library.

The second part of this dissertation, entitled *A Partial-Order Approach to Infinitary Rewriting*, is concerned with *infinitary rewriting*, a field that studies transfinite rewrite sequences. We extend the established theory of infinitary rewriting in two directions: (1) a novel approach to convergence in infinitary rewriting that replaces convergence in a metric space with the limit inferior in a partially ordered set; (2) extending infinitary term rewriting to infinitary term *graph* rewriting.

For the first item, we show correspondences between the established calculi based on metric convergence and the newly developed calculi based on partial orders. We also study both approaches on an abstract level and show the advantages of our partial order approach in terms of better confluence and normalisation properties of infinitary term rewriting as well as in terms of better completeness properties for infinitary term graph rewriting.

For the second item, we explore several approaches to convergence on term graphs and analyse the resulting calculi. We distinguish two calculi among them – based on a metric space respectively a partially ordered set – by showing that they satisfy strong soundness and completeness properties w.r.t. infinitary term rewriting.

Dansk Resumé

Denne afhandling er en samling af ni artikler, der falder indenfor to uafhængige forskningsområder.

I den første del udvikler vi teknikker til modulær implementering af programmeringssprog. Indenfor dette problemfelt fokuserer vi på operationer på abstrakte syntaks-træer med det formål at udvikle en rammestruktur, der muliggør definition, manipulation og komposition af sådanne operationer. Resultatet af vores arbejde er et omfattende bibliotek, der tilbyder disse faciliteter.

Det, der adskiller vores tilgang fra andres, er brugen af rekursionssystemer, der stammer fra *træ-automater*, til at implementere operationer på abstrakte syntaks-træer. I de første to artikler illustrerer vi styrken af denne tilgang ved at præsentere *træ-homomorfier* – en meget begrænset form for træ-automater – som fundament for simple træ-transformationer. Deres enkelthed tillader os at kombinere dem med monadiske effekter, manipulere og kombinere dem på en fleksibel måde, samt foretage optimeringer i form af *deforestation*. I den tredje artikel går vi videre med mere udtryksfulde træ-automater. Disse mere udtryksfulde automater er sædvanligvis mere omstændelige at definere, eftersom de kombinerer forskellige beregningsmæssige aspekter. Vi viser imidlertid at disse automater kan konstrueres fra mere simple automater, nemlig træ-homomorfier og simple state machines. I den anden artikel fokuserer vi på det vigtige aspekt at repræsentere variabelnavne og variabel-bindinger vha. en omhyggelig restringeret *højere ordens abstrakt syntaks*.

I den fjerde artikel præsenterer vi en omfattende og realistisk anvendelse af vores bibliotek: en prototype-implementation af et nyt *enterprise resource planning* system bygget op omkring en samling af domænespecifikke sprog, der gør det muligt at skræddersy systemet på en meget fleksibel måde. Systemet kombinerer adskillige højt integrerede domænespecifikke sprog, som er implementeret ved brug af vores bibliotek.

Den anden del af afhandlingen omhandler *transfinit termomskrivning*, et område der beskæftiger sig med uendelige omskrivnings-sekvenser. Vi udvider den eksisterende teori om transfinit termomskrivning i to retninger: (1) En ny tilgang til konvergens i transfinit termomskrivning, der erstatter konvergens i et metrisk rum med *limes inferior* i en partielt ordnet mængde; (2) Udvidelse af transfinit termomskrivning til transfinit termgrafomskrivning.

Med hensyn til det første punkt, viser vi korrespondancen mellem de etablerede kalkyler, baseret på metrisk konvergens, og de nyligt udviklede kalkyler, baseret på partielle ordninger. Derudover undersøger vi begge tilgange på et abstrakt niveau og påviser fordelene ved vores tilgang baseret på partiel ordning i forhold til bedre konfluens- og normaliserings-egenskaber for transfinit termomskrivning, såvel som i forhold til bedre komplethedsegenskaber for transfinit termgrafomskrivning. Angående det andet punkt udforsker vi adskillige tilgange til konvergens vedrørende termgrafer og analyserer de resulterende kalkyler. Vi udvælger to kalkyler iblandt dem – baseret på henholdsvis et metrisk rum og en partiel ordning – og viser, at disse tilfredsstiller stærk sundheds- og komplethedsegenskaber med hensyn til termomskrivning.

Contents

Preface

This dissertation has been submitted to the PhD School of Science, Faculty of Science, University of Copenhagen in partial fulfillment of the requirements for a PhD degree at the Department of Computer Science, University of Copenhagen, Denmark.

This dissertation is written as a synopsis with nine research papers enclosed. The first chapter presents a brief overview of the two topics that this dissertation is concerned with. The subsequent two chapters give a comprehensive overview of the two individual topics. In each of these two chapters, I outline the corresponding area of research, present the problem that I tried to solve, explain the contributions of my work, and relate it to existing results found in the literature. At the end of each of the two chapters, I formulate the conclusions that can be drawn from my results and briefly outline areas of possible future study. The enclosed research papers are found in two appendices at the end of this document.

At this point I would like to take the opportunity to thank the people that helped me during my work on this dissertation. First and foremost, I thank my PhD adviser Fritz Henglein for his guidance. He has been a constant source of inspiration and motivation. I would also like to thank my master's thesis adviser Bernhard Gramlich who introduced me to the field of infinitary term rewriting and gave me the confidence to continue studying it.

I am indebted to my coauthors Tom Hvitved and Jesper Andersen as well as Michael Kirkedal Carøe who kindly commented on an earlier draft of this dissertation. I thank Clemens Grabmayer and Vincent van Oostrom for hosting me at Utrecht University for three months and for providing an inspiring academic environment with many fruitful discussions, sharing new insights and ideas. Likewise I thank my former and current colleagues of the APL group at DIKU for creating an inspiring and enjoyable work environment. In particular, I would like to thank my fellow PhD students and the members of the "lunch club" for creating a great social environment as well.

Last but not least, I owe special thanks to my family and friends for their invaluable support, their patience and understanding during my work on this dissertation.

Patrick Bahr

Chapter 1

Introduction

This dissertation covers two independent topics: modular implementation techniques for programming languages on the one hand and infinitary rewriting on the other hand. Despite their independence on the surface, both topics are direct applications of *rewriting systems* [164].[1]

A rewriting system is simply a binary relation $\to$ over a set of objects – typically some set of *terms*. The intended meaning of $\to$ is usually that of a computation step, i.e. $s \to t$ means s is transformed into t by a single computation step. For example, we can model evaluation of arithmetic expressions as rewriting system such that we get a rewrite step $(1 + 2) * 3 \to 3 * 3$. Arguably the most famous rewriting system is the λ-calculus [18] with its β reduction relation $\to_\beta$.

The relation $\to$ is typically given in a structured form by a set of rules called *rewrite rules*. For example, evaluation of expressions over natural numbers and addition may be defined by the following two rules that represent the familiar recursive definition of addition:

$$x + 0 \to x$$
$$x + s(y) \to s(x + y)$$

These systems will form the foundation for both parts of this dissertation.

Part One

In the first part of this dissertation, we make use of the particular structure of rewrite rules to facilitate modularity in implementations of programming languages. The goal is to structure implementations of programming languages, i.e. compilers, interpreters etc., such that their parts can be reused. Rewriting systems in this approach are programs that manipulate *abstract syntax trees*. The restricted forms of rewrite rules that we consider come in the disguise of names such as *algebras*, *tree homomorphisms*, and *tree automata*. They provide the structure that is needed to facilitate reusing, manipulating, and combining the components of a programming language implementation.

The main question that we investigate here is the following: how to combine two programs P and Q that implement programming language features F and

[1] Admittedly, this seems like a rather loose connection as many rewriting systems enthusiasts are able to see traces of rewriting systems virtually everywhere.

G, respectively, such that the combination of P and Q implements both F and G? This question begs the counter question what the phrase "implements both F and G" actually means. In this dissertation, we take this second question as an instance of the first one since we view the semantics of a programming language or a programming language feature also as a program, viz. an interpreter. Combining the semantics is, thus, a matter of combining programs.

The ultimate goal is to build programming languages (including both their semantic specifications and their implementations, say, in the form of compilers) by combining components from a library of language features. Since one size does not fit all, we also have to ensure that these components are flexible, i.e. that they can be readily manipulated and refactored to fit. And finally, it is vital to have a type system that guides the language designer and implementer in how components can and cannot be combined with each other. The work presented in this dissertation should be understood as a step towards this goal.

Part Two

In the second part, we are less interested in restricting the structure of the rewrite rules but rather loosening the computational interpretation of rewrite systems by also including computations both of *infinite length* and on *infinite objects*. In the informal example given earlier, we first perform a rewrite step $(1 + 2) * 3 \to 3 * 3$ after which we can continue with the step $3 * 3 \to 9$. At this point we have reached the *result* of the computation; the rewrite sequence terminates.

In general, this does not have to be the case. For example, β-reduction in the (untyped) λ-calculus is certainly not guaranteed to terminate after finitely many steps. For a simpler example, consider a rewrite system that performs rewrites of the form $\mathsf{zeros} \to 0 : \mathsf{zeros}$, i.e. any occurrence of zeros is replaced by $0 : \mathsf{zeros}$. Starting with the symbol zeros, we may produce the following rewrite sequence:[2]

$$\mathsf{zeros} \to 0 : \mathsf{zeros} \to 0 : 0 : \mathsf{zeros} \to \dots$$

This rewrite sequence does not terminate with a result. In each intermediate term we find the symbol zeros, which means we may still perform a rewrite step.

While the above rewrite sequence does not terminate, this non-termination is somewhat well-behaved. It may be seen as producing an infinite term $0 : 0 : 0 : \dots$ – an infinite list of zeros. More precisely, the rewrite sequence *converges* to the infinite term $0 : 0 : 0 : \dots$, i.e. the intermediate terms of the rewrite sequence come arbitrarily "close" to the term $0 : 0 : 0 : \dots$.

An infinite rewrite sequence does not always converge, though. Seldom, however, a reduction sequence is not converging at all: there are usually parts of the reduction sequence that contribute in some form to a *partial* result. For example, given a rule $\mathsf{swap}(x, y) \to \mathsf{swap}(y, x)$, which swaps the two argument of the function symbol swap, we obtain the rewrite sequence

$$\mathsf{swap}(0, 1) \to \mathsf{swap}(1, 0) \to \mathsf{swap}(0, 1) \to \dots$$

[2] We assume that ":" is an infix symbol that associates to the right, e.g. $0 : 0 : \mathsf{zeros}$ is parsed as $0 : (0 : \mathsf{zeros})$.

which intuitively does not converge as it alternates between two different terms. On the other hand, the function symbol swap at the top of the terms does not change. So the sequence somehow partially converges to a partial result $\mathsf{swap}(\cdot,\cdot)$ in which the function symbol is known, but the two arguments are unknown or undefined.

In the second part of this dissertation we shall develop notions of convergence that formalise this intuitive understanding with the aim of describing the full spectrum from *converging* to *not converging at all*.

Chapter 2

Modular Implementation of Programming Languages

Programming languages are first and foremost tools for building software systems. With the broadening of information technology in general and applications of software technology in particular, the development of new programming languages has picked up in pace accordingly. And with good reason. New application domains require new languages for productive software development [181]. With the advent (and buzz) of *domain-specific languages (DSLs)* [22] – languages target to a particular problem domain – the proliferation of new languages accelerated further. Today, building DSLs is considered a standard technique in software engineering and is even covered in introductory text for mainstream programming languages [115, 144].

On the other hand programming languages are also software systems themselves – usually consisting of a *compiler* or an *interpreter* and optionally a tool ecosystem consisting of an *integrated development environment*, analysis tools etc. Advances in implementation techniques for programming languages have helped making the development of new programming languages less expensive and, thus, an attractive option for increasing software development productivity [181].

Nevertheless, compilers and interpreters are qualitatively different from other pieces of software. Typically, the performance and correctness of the implementation have greater priority as these properties are propagated to the software that is written in that language. In other words, if a compiler produces poorly performing or even wrong code, then programs compiled with it also perform poorly or are defect, respectively. Additionally, compiler and interpreter implementations are complex; their components are usually subtly interwoven. Changing the implementation of one particular language feature may unexpectedly change the implementation of another one with potentially unintended consequences for the semantics of the implemented language.

The objective of this part of the dissertation is to develop new techniques to structure programming language implementations in a modular fashion and thereby overcome the issues described above. Section 2.1 describes what modularity means in this setting, what particular kind of modularity we attempt to achieve, and what the expected benefits are. In Section 2.2, we describe what has to be done to deal with semantics in a modular fashion. Section 2.3 briefly

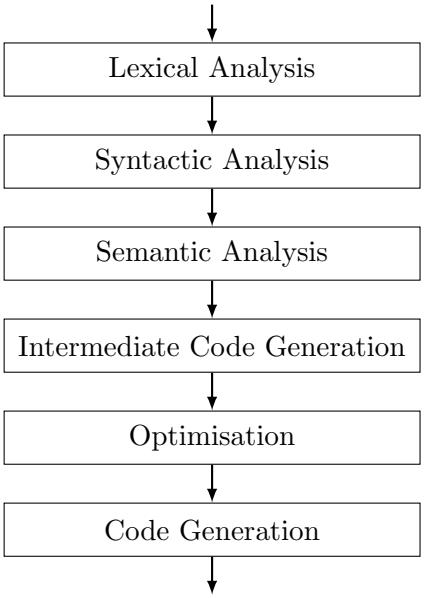


Figure 2.1: Phases of a typical compiler.

surveys modular implementation techniques that are known in the literature and in Section 2.4 the contributions made in this dissertation are summarised. We conclude this chapter in Section 2.5. The papers that make up the contributions of this part of the dissertation are included in Appendix A and are referred to subsequently as Paper **A1** to Paper **A4**.

2.1 Modularity

Modularity is a fundamental principle in software engineering that copes with the complexity of a software system by structuring it into interchangeable and reusable modules. This being the case, modularity seems to be the right approach to curb the abovementioned problems of programming language implementations. However, the very same problems make establishing modularity in programming language implementations difficult in the first place. In particular, modularising a compiler or an interpreter according to the features of the implemented programming language is still a challenge.

Notwithstanding the above, it would be fallacious to claim that compilers are designed as completely monolithic pieces of software. The nature of the task performed by a compiler, viz. transforming a programming language into a different one, lends itself to a modularisation akin to an assembly line [1]. Figure 2.1 illustrates this structure. According to this scheme, compilation is performed in *phases*, each of which only performs a certain task. Some of the phases, like *semantic analysis* and *optimisation*, may in turn be performed in several phases or runs. For example, some optimisation techniques complement each other and are thus performed one after the other, e.g. *constant propagation* and *dead code elimination*.

This subdivision of a compiler into phases is extremely helpful for building, maintaining, and extending compilers. The resulting modularity along the vertical dimension follows the principle of *separation of concerns* in a straightforward manner: each phase has a clearly defined functionality, there is little overlap between the phases, and there is a concise interface between neighbouring phases.

The kind of modularity that we aim for in this dissertation is orthogonal to the internal separation of concerns described above. We aim to modularise language implementations along *language features* as well. With regard to Figure 2.1, this modularity runs horizontally. For example, we want to split the *intermediate code generation* phase into modules for arithmetic, exception handing, bulk operations etc. Each module only covers a certain language feature.

This additional dimension of modularity is qualitatively different from the modularity depicted in Figure 2.1 and offers far more that simply a higher degree in modularity. By structuring programming language implementations along language features, we open up new ways of building, maintaining, and evolving them. It is unlikely that the implementation of the semantic analysis phase for one language can be reused for a different language. If this implementation is, however, in itself structured in terms of language features, we may reuse *some* of those language features for other languages that happen to have those languages features as well. Moreover, if it is possible to manipulate the implementation of a language feature with appropriate operations or by parametrisation, then it is far more likely that this implementation can be reused in a different context. Given these properties of such a modular architecture, we can furthermore build a library consisting of implementations for different language features. Building a compiler or an interpreter then becomes a matter of combining these implementations. The long-term goal of this architecture is to have a DSL for building programming language implementations.

Due to the importance of programming language development, in particular for constructing DSLs, advancing modularity in compiler and interpreter implementations promises a number of benefits for software engineering. The additional dimension of modularity described above is expected to make programming language development simpler, more efficient, and less error-prone. The cost of developing DSLs has to be weighted against the productivity benefit it offers [125]. Thus, reducing the development costs makes the benefits of DSLs available to a potentially larger class of applications and problem domains. Apart from that, modular implementations of languages also reduce the cost and risk of incremental changes to a language, which typically pose serious problems in practice [167].

2.2 Modular Semantics

When building a programming language, producing the actual implementation of it is only part of the process.[1] As mentioned earlier, the *correctness* of the implementation has typically high priority. That is, the implementation has to respect the *semantic* description of the programming language it is expected

[1] Well, at least ideally.

to implement. This means that a compiler has to produce, for each program, corresponding machine code that has the same[2] semantics as the input program. Consequently, in order to devise such a compiler in a modular fashion, we must be able to relate each module of an implementation to a corresponding part of the semantic description of the language. That means, we also need a modular approach to the semantics of programming languages.

There exist a number of semantic frameworks that allow for a modular semantic description of a language. *Modular monadic semantics* was the first such approach. It is based on Moggi's idea to use *monads* to describe computational effects in a denotational semantics [128, 130]. When he developed his monadic semantics, Moggi already considered *monad transformers* as a tool to construct monads in a modular fashion [129]. This idea of constructing monads was subsequently used as a foundation to develop a modular semantic framework [111, 112]. Moreover, monads were also used to introduce modularity into frameworks such as *action semantics* [141, 180], which subsequently also spawned the work on modular action semantics [50, 84, 132] independently from monadic semantics. Recent work has strengthened the modularity aspect of monad transformers further [85–87, 154], both on the theoretical and the practical level.

Monadic semantics has proven to be an excellent starting point for a modular programming language implementation. Monad transformers were used to implement programming language interpreters [111, 113, 160] and compilers [71–73, 112] in a modular fashion. The appeal of these approaches lie in the fact that the implementation is derived from the semantics and is thus highly reliable.

Another major approach to modular semantics is based on *operational semantics*, in particular Plotkin's *structured operational semantics* (*SOS*) [148]. Mosses [131, 134] has pioneered a modular approach to SOS. *Modular SOS* has found particularly strong adoption in rewriting logic [27, 31, 44, 126], which provides a rich tool set for specifying and prototyping programming languages [127]. Apart from that, Jaskelioff et al. [88] have also produced an implementation of a modular variant of Turi and Plotkin's *mathematical operational semantics* [166]. While a general technique for building compilers on the basis of a modular SOS description has not been developed, modular SOS has the advantage of being better suited for concurrency than monadic semantics.

While other techniques may also offer some form of modularity, for example the *Vienna Development Model* [133], *abstract state machines* [69] and *evaluation contexts* [59], the modularity obtained from these approaches is not sufficient at the moment.

2.3 Modular Implementation Techniques

2.3.1 Parsing

The concrete syntax is arguably the easiest component of a programming language to fit into the modularity principle. The traditional approach of describing syntax, viz. in the form of a *formal grammar*, is already modular. A *parser*

[2] In general, we only require that the semantics of the produced code *refines* the input program's semantics.

generator is then able to produce a parser from such a formal grammar specification. This being the case, formal grammar languages, as they are used by parser generators, provide a specification framework for writing parsers in a modular fashion.

More flexibility for combining and manipulating parser fragments in order to facilitate reuse and evolution of existing implementations is offered by *parser combinators* [81]. Instead of specifying a parser by giving a grammar, a parser is constructed by combining and manipulating simple parsers using higher-order functions, which are called parser combinators. In this approach, the specification *is* the implementation, which obviates the use of a generator to produce the actual implementation. Moreover, the use of higher-order functions provides additional flexibility for reuse and compositionality. At the same time, the use of a monadic interface for parser combinator libraries [82] makes parser implementations readable and easily maintainable. Today, parser combinator are a mature and well-established technology [110, 161] and we find powerful parser combinator libraries for virtually all major programming languages.

2.3.2 Typing ASTs

A parser, however, only solves half of the problem in dealing with the syntax of programming languages, viz. transforming the concrete syntax of a program into an *abstract syntax tree* (AST), which is then the input for the compiler, interpreter or other tools. Dealing with abstract syntax in a suitable fashion that facilitates modularity is the other half. The underlying problem that we face with abstract syntax is summarised in what Wadler [178] calls the *expression problem*:

> "The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

Several solutions to this problem have been suggested. In object-oriented languages such solutions are often based on *generics* [29, 138, 165, 178]. Other approaches try to address the problem by introducing new language features such as *virtual types* [28, 185], *multi-methods* [32], *open classes* [36], *virtual classes* [118, 137] and *program units* [60, 120]. One should note that some approaches *do* offer the required extensibility but lack type safety [108, 140, 173].

Since the functional programming paradigm is closer to the theoretical underpinnings that we intend to use for programming language implementations, we are more interested in solutions to the expression problem in a functional programming style. The functional programming paradigm also becomes helpful for representing the operations on ASTs that are the subject of the section below. Broadly speaking, there are two qualitatively different approaches to the expression problem: one represents AST types as fixed points of regular functors [162]; the other uses a Church encoding [139] instead. In our work on modularity, we adopt the fixed points approach based on Swierstra's *data types à la carte* [162]. Day and Hutton [43] recently used this approach in order to structure a compiler implementation. Axelsson [8] uses a variant of the fixed-point approach

that represents the AST in an applicative fashion, which facilitates generic programming. The Church encoding was recently also suggested for facilitating modularity and reuse in the context of mechanised metatheory [46]. Similarly to object-oriented languages, a number of language features have been introduced for functional languages that address the expression problem, including *polymorphic variants* [63, 64], *extensible algebraic data types* [184], and *open data types* [117].

2.3.3 Operations on ASTs

The previous section gave an overview of the solutions to the expression problem. Recall that the expression problem asks for methods to extend data types *and* to add function over these data types. However, due to the complexity of compilers, simply adding functions is not sufficient in order to provide a modular implementation framework. We also need to bee able to manipulate and combine existing functions on ASTs much in the same way as parser combinators are able to manipulate and combine parsers.

The essential operations performed by compilers and interpreters can be summarised as traversals and transformations of ASTs. And indeed this is the view that we take in this dissertation. In order to facilitate reuse, these operations have to be given in a structured form such that they can be manipulated and combined in a flexible yet predictable manner. Both *attribute grammars* [106] and *tree automata* [37] provide such structure.

Attribute Grammars

Attribute grammars, originally devised by Knuth [106] to describe semantic aspects of programming languages, are a practical tool for defining – in a declarative manner – syntax-directed computations as they are needed for implementing a programming language. As such they have proved to be a powerful tool for compiler construction [52, 62, 68, 107]. Thus, several approaches to extend attribute grammars were investigated in order to enable modular specifications [51, 57, 90, 95, 153]. This includes, in particular, higher-order attribute grammars [61, 163, 168, 174], which make it possible to combine several passes of a compiler and techniques borrowed from object-oriented programming [75, 76, 124].

The abovementioned approaches to modular attribute grammars are not powerful enough for the high degree of modularity that we have in mind, though. More promising, is the first-class attribute grammars approach of Viera et al. [171], who embed an attribute grammar system as a DSL in Haskell. Embedding attribute grammars in a functional language as first-class objects means that they can be manipulated and combined with higher-order functions akin to parser combinators [170]. And as Viera et al. [172] show, despite its flexibility, this approach can be efficiently implemented as well.

Tree Automata

Tree automata [37] are chiefly theoretical tools to study properties of certain classes of operations on trees. For the purpose of programming language implementations we are mostly interested in *tree transducers*, which are automata that

implement tree transformations. The compositionality that is typically studied in the tree automata community is sequential composition [53, 54]. While sequential composition merely implements function composition of the tree transformations described by tree transducers, this composition is nevertheless useful for improving time and space efficiency [109, 175, 176] as it avoids the construction of intermediate data structures [177].

An important stepping stone both for devising a rich set of operations on tree transducers and for embedding them in an implementation language is the category theoretic formalisation by Jürgensen [91], Jürgensen and Vogler [92] and Hasuo et al. [74]. In fact, Andersen and Brabrand [2] recently used *tree homomorphisms*[3] – a restricted class of tree transducers – in order to build a combinator library of syntax transformations. Tree homomorphisms are, however, quite limited in terms of expressiveness, which is why the combinator library of Andersen and Brabrand [2] is only used to define so-called *syntax extensions* – syntactic sugar that is translated to some core syntax.[4]

In our work, we have borrowed heavily from tree automata in order to construct an expressive combinator library. While both Paper **A1** and Paper **A2** have their foundation in catamorphisms and anamorphisms [123], we also employ tree homomorphisms, on which we define a comprehensive set of combinators in order to facilitate reuse. Similarly to Andersen and Brabrand [2], we demonstrate the translation of syntactic sugar to a core language as the characteristic application of tree homomorphisms. In Paper **A3**, we expand on the role of tree automata as the basis for modular recursion schemes. We use both bottom-up and top-down tree transducers, which allow transformations beyond simple reduction of syntactic sugar. However, instead of constructing these transducers directly, which becomes cumbersome for realistic transformations, transducers are constructed by combining tree homomorphisms and an ordinary bottom-up or top-down state machine.

2.3.4 Names and Binders

While not necessarily an issue inherent to modular implementation techniques, dealing with (variable) names and binders in ASTs is a perpetual source of headache for programming language implementers and thus needs to be addressed in satisfying manner. The straightforward way of using explicit variable names to represent variables in an AST is conceptually easy; but extra effort has to be put in to ensure both that functions defined on ASTs are in fact invariant under α-renaming and that free variables are not accidentally captured by binders, e.g. in substitutions.

Higher-order abstract syntax (HOAS) [143] addresses this issue by using the host language's variable binding mechanism to represent binders in the object language. This is achieved by representing a binder as a function type, with the argument type representing the bound variable and the return type representing

[3]Though, Andersen and Brabrand [2] use the name *constructive catamorphism* instead.

[4]Conversely, syntax extensions in the form of syntactic sugar only require tree homomorphisms; thus Andersen and Brabrand [2] were able exploit the unique structure of tree homomorphisms to build a combinator library.

the scope of the binder. While this approach is popular in the context of mechanised metatheory, the inclusion of function types in the type of ASTs poses considerable problems for implementing efficient recursion schemes [58, 122, 155, 182]. This problem can be avoided by restricting the function space that is used for encoding binders. Fegaras and Sheard [58] realised that a restriction to *parametric functions* provides a solution to this problem. While Fegaras and Sheard relied on a custom type system for this, Chlipala [35], as well as Washburn and Weirich [182], later developed an elegant encoding that only requires *parametric polymorphism* as available in System F. The use of parametric polymorphism can be easily integrated with the Church encoding of ASTs with virtually no overhead [30, 182]; the representation of variable binders in the Church encoding goes back to Coquand and Huet [38]. The use of HOAS is particularly useful for embedded DSLs as HOAS requires little syntactic overhead.

In this dissertation, we shall use this parametric HOAS approach in order to deal with names and binders. But there are other approaches as well. This includes de Bruijn indices [45] and nominal sets [145, 146]. These two approaches have formed the foundation for numerous libraries [33, 105, 150, 151, 183], tools and language extensions [34, 149, 159] for dealing with name binders.

2.4 Contributions of this Dissertation

As stated in Chapter 1, the goal of this part of the dissertation is to develop techniques that allow a language implementer to structure a compiler or an interpreter for a language in a modular fashion. This modularity should allow the implementer to modify and reuse components of an implementation. Our focus is put on developing a combinator library for such implementation modules similar to parser combinator libraries.

The approach that we developed is chiefly based on Swierstra's *data types à la carte* [162], a functional pearl that combines ideas from the work on modular monadic interpreters of Liang et al. [113] and the framework of programming in terms of catamorphisms and anamorphisms popularised by Meijer et al. [123].

Paper **A1** expands Swierstra's work with the goal of producing a Haskell [119] library – dubbed *compositional data types* – suitable for realistic applications. In this paper, we implement various recursion schemes based on Vene's PhD dissertation [169] and monadic versions thereof. We implement comprehensive *generic programming* functionality and show that its run time performance is on par with dedicated generic programming libraries. We extend the scope of Swierstra's approach to mutually recursive data types and generalised algebraic data types [89]. We implement *tree homomorphisms* and illustrate their practical relevance for translating syntactic sugar. Moreover, we show how tree homomorphisms enable program optimisation via deforestation [177]. We also compare the run time of modularly defined transformations in this library with equivalent "monolithic" implementations.

Paper **A2** is a followup to Paper **A1** that extends the compositional data type library with a principled approach to variable names and binders based on Chlipala's *parametric higher-order abstract syntax* (*PHOAS*) [35]. We show that Chlipala's technique can be translated into Haskell and that the results of

Paper **A1** largely carry over to the setting of PHOAS. The only difficulty occurs in the implementation of monadic recursion schemes. While monadic effects cannot be sequenced for arbitrary catamorphisms like in the purely first-order setting, it is still possible for restricted recursion schemes such as tree homomorphisms.

In Paper **A3** we explore the possibilities that recursion schemes derived from tree automata offer. In the preceding two papers we already use a quite restrictive form of tree transducers, viz. tree homomorphisms. In simple terms, tree homomorphisms are tree transducers without state. While tree homomorphisms, as we show in Paper **A1** and Paper **A2**, enjoy a number of advantageous properties, these properties are paid with the price of limited expressiveness. Vice versa, the added expressiveness of bottom-up and top-down tree transducers, which add upwards respectively downwards state propagation, has to be paid as well: they are quite cumbersome to program as they mix tree transformation and state propagation. As solution we propose to build tree transducers by combing tree homomorphisms with a bottom-up or top-down state machine. This decomposes the specification of a tree transducer into two separate parts: tree transformation and state propagation. In the same way we also implement combinators that combine two state machines regardless of whether they are both bottom-up, both top-down or mixed. The resulting combinator library allows for extensive manipulation and reuse of operations on ASTs. The application to modular compiler construction is illustrated with a number of running examples.

Paper **A4** presents a comprehensive and realistic application of the compositional data types library. In the paper, we describe a prototype implementation of a novel *enterprise resource planning* system based on the architecture of *process-oriented event-driven transaction systems* (*POETS*) of Henglein et al. [78]. Instead of relying on a mixture of relational database systems and imperative programming languages, the POETS architecture is build upon a family of DSLs that makes it possible to customise the system in a highly flexible manner. Since the system combines several integrated DSLs, drawing on a modular approach to implement the DSLs does not only reduce the effort to implement the system. It also ensures that common features of the DSLs have in fact the same semantics.

2.5 Conclusions and Perspectives

After reading this chapter it should be clear that the subject of modular programming language implementation is a complex one, touching a wide variety of topics each with many challenging problems of its own. Certainly this dissertation can only offer contributions to a small fraction of these problems.

In our work presented here, we only consider the implementation part of constructing programming languages and within that topic we focused chiefly on the aspect of AST manipulation. We believe, however, that a principled approach to modular programming language construction has to take into consideration the abovementioned surrounding aspects as well.

Particularly challenging is the integration of a modular implementation framework with a corresponding modular semantic framework such that there is a clear correspondence between an implementation module and its semantic description.

Ideally, both the implementation and the semantic description should consist of reusable building blocks akin to Mosses's proposal of a component-based description of programming languages [135].

Taking this approach further, the implementation aspect and the semantic description become expressible within the same formalism and are merely on two opposing ends of the spectrum of abstraction. The question of how to relate between a these two aspects – implementation and semantics – then becomes a question of how to move on this spectrum. And there is a lot to be found in the literature about this, from synthesis techniques like deriving a compiler systematically from formal descriptions [121], to formal verification of compilers by step-wise refinement [136].

Coming back to the core contributions of this dissertation, there are promising approaches to typed modular AST transformations beyond the ones we discussed here and which should be explored as well. The first alternative is, of course, to go back to the foundations and study recursion schemes in general, e.g. folds and unfolds [123] and generalisations thereof [79, 169]. AST transformations are often dependent on information that has to be collected by traversing the AST. This propagation of information is expressed in attribute grammars in the form of attributes and in tree transducers in the form of state. A foundational approach to this kind of propagation of information is studied in the form of upwards and downwards accumulations by Gibbons [65, 66, 67].

Chapter 3

A Partial-Order Approach to Infinitary Rewriting

In this part of the dissertation we deal with "pure" rewriting systems. As mentioned in the introduction, we are interested in infinite rewrite sequences, i.e. infinitely long sequences of consecutive rewrite steps such as the example

$$\mathsf{zeros} \to 0 : \mathsf{zeros} \to 0 : 0 : \mathsf{zeros} \to \dots$$

that we have seen in Chapter 1. In the theory of *infinitary rewriting* we strive to give sensible meaning to such infinite rewrite sequences in the form of a notion of *convergence*. The meaning of the word "sensible" depends on several factors such as the underlying finitary calculus or the problem that should be solved. In Section 3.2, we survey the different notions of convergence that are found in the literature. Before we do that, however, we have a brief look at what we gain by considering infinite rewrite sequences in the first place in Section 3.1. In Section 3.3 we discuss the contributions made in this dissertation, and Section 3.4 concludes. The papers that make up the contributions of this part of the dissertation are found in Appendix B and are referred to subsequently as Paper **B1** to Paper **B5**.

3.1 To Infinity and Beyond! – But Why?

Before we delve into the topic of infinitary rewriting, we reflect on the merits of this endeavour. Of course, the physical realisation of computation has to deal with the restriction of finite time and memory resources. Nevertheless, infinite structures are abundant in computer science as this often makes theoretical analysis simpler. But there are also classes of computational systems, e.g. reactive systems [70], that are in fact designed to run indefinitely.

Closer to the kind of infinite computations that we are considering here are iterative approximation methods that get arbitrarily close to the desired result without reaching its exact value. Examples of this technique are Archimedes' algorithm to calculate π and Newton-Raphson's algorithm to calculate square roots, which we shall discuss later. While these algorithms do not find the result within finitely many iterations, the approximations they produce *converge* to the

desired result. This notion of convergence allows us to evaluate the correctness of such algorithms despite their non-terminating nature and it is the model that infinitary rewriting adopts to give meaning to infinite rewrite sequences.

3.1.1 Non-Strict Evaluation

Surely, as computer scientists we are ultimately concerned with devising machinery that for a given input computes some desired output within reasonable (and thus finite) time. That said, it should come to no surprise to the readers familiar with the Halting Problem that the road to that ultimate goal has to be built upon infinite structures.[1] That is to say a program that terminates for any given input may still be composed of components of which some are infinitary in nature, e.g. loops or recursion.

This infinitary nature of some of the building blocks of a program may in fact be preferable as it facilitates compositionality. For example, John Hughes [80] makes this argument for *non-strict evaluation*[2] in functional programming languages. To illustrate this point with a simple example, we consider the following Haskell function `lineNumbers`:

```
lineNumbers :: String -> String
lineNumbers str = unlines (zipWith mkLine [1 ..] (lines str))
    where mkLine n l = show n ++ " " ++ l
```

The function `lineNumbers` adds line numbers to the string that it is given. For example, it performs the following transformation:

```
Collect Theorems                    1 Collect Theorems
???          ⊢——lineNumbers——→       2 ???
Profit                              3 Profit
```

To achieve this transformation, `lineNumbers` splits the input string `str` into its constituent lines producing a list of strings (`lines str`). The expression `[1 ..]` produces the infinite (!) list of consecutive numbers starting from 1. Then the function combines the two list using `zipWith`, which combines corresponding elements of the two lists using the function `mkLine`, thus prepending each line with the corresponding number from the list. Afterwards, the resulting list of strings is converted into a single string again. The key for this function to work despite the fact that the list `[1 ..]` it uses is infinite, lies the fact that `zipWith` terminates as soon as the end of one of the two input lists is reached. Hence, given that the input string is finite, the function `lineNumbers` terminates.

In order to understand how `lineNumbers` can deal with an infinite list we have to take a closer look at the expression `[1 ..]`. This expression is a shorthand for `enumFrom 1`, where `enumFrom` is defined as follows:

```
enumFrom :: Integer -> [Integer]
enumFrom n = n : enumFrom (n+1)
```

[1] Beware of the potholes of perpetual darkness!

[2] John Hughes [80] argues more specifically for lazy evaluation which we will consider later.

This function does not terminate for any input! Conceptually, given an integer n it produces an infinite list of consecutive natural numbers starting from n. However, the non-strict evaluation strategy that Haskell uses only evaluates a subexpression if its result is *needed*. For example, in case the input string to `lineNumbers` only contains three lines, `enumFrom` is expanded only thrice:

$$\texttt{enumFrom 1} \to^* \texttt{1 : enumFrom 2} \to^* \texttt{1 : 2 : enumFrom 3} \to^* \texttt{1 : 2 : 3 : enumFrom 4}$$

The steps above are interleaved with evaluation steps in which `zipWith` "consumes" the first three elements of the partially generated list. Every time `zipWith` demands a new element of the list, `enumFrom` is expanded.

Of course, the function `lineNumbers` can be reformulated without using an infinite list. For example, we may instead use a function that produces a list with just the right amount of numbers. However, this requires us to count the number of lines first, which adds a redundant iteration through the list of lines and furthermore clutters the definition. The second alternative would be to simply define a recursive function that both maintains a counter and prepends the line numbers in one step. The approach, however, lacks the clarity and compositionality of `lineNumbers`.

Hughes [80] has a number of more involved examples that illustrate the same point that we wanted to make above: the use non-terminating programs often facilitate the construction of *modular* terminating programs.

3.1.2 Sharing

Before continuing the discussion about the motivation of infinitary rewriting, we look a bit closer at non-strict evaluation. Out in the wild, non-strict evaluation rarely comes without its companion, called *sharing*, which prevents expressions to be evaluated more than once [179]. For strict evaluation this problem does not exist; each expression is evaluated exactly once. Non-strict evaluation, on the other hand, defers evaluation of an expression until it is needed. When the expression is eventually evaluated, however, it might have already been duplicated. Hence, such a duplicate has to be evaluated again even though it will evaluate to the same value as the original. Sharing avoids this effect by mimicking duplication of subexpressions by duplication of pointers. This combination of non-strict evaluation and sharing is also known as *lazy evaluation* [77] or *call by need* [179].

To illustrate the importance of sharing, we consider one of Hughes' examples [80] with which he demonstrates the use of lazy evaluation. He implements Newton-Raphson's algorithm to calculate square roots. Starting with some initial guess a_0, the algorithm produces increasingly more accurate approximations of $\sqrt{s}$ using the following rule:

$$a_{i+1} = \frac{a_i + s/a_i}{2}$$

Whenever the resulting sequence (a_i) converges, which depends on the initial guess a_0, it converges to $\sqrt{s}$, i.e. the difference $|\sqrt{s} - a_i|$ tends to 0 as i increases. The above rule can be realised by the following Haskell function `next`:

```
next :: Double -> Double -> Double
next s a = (a + s/a) / 2
```

The sequence (a_i) may be produced by iterating the function `next` with the following combinator `repeat`:

```
repeat :: (a -> a) -> a -> [a]
repeat f a = a : repeat f (f a)
```

The function `repeat` produces an infinite list with an increasing number of applications of the function f that is given as argument. Given an initial guess a_0 and the iteration function $f =$ `next` s, the expression `repeat` $f\ a_0$ yields the infinite list

$$a_0 : f\ a_0 : f(f\ a_0) : f(f(f\ a_0)) : \ldots$$

The implementation of Newton-Raphson's algorithm feeds this list into a function `within`, which checks whether at some point in the list two successive values are not more that some tolerance `eps` apart from each other. If that is the case, the element of the list at that point is returned:

```
within :: Double -> [Double] -> Double
within eps (a : b : r)
  | abs (a - b) <= eps = b
  | otherwise          = within eps (b : r)
```

We then obtain the following implementation of `sqrt`:

```
sqrt :: Double -> Double -> Double -> Double
sqrt a0 eps r = within eps (repeat (next r) a0)
```

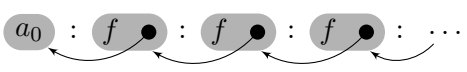
Given that a suitable initial guess a_0 is provided that causes the sequence (a_i) to converge, the above function will terminate despite the fact that the list generated by `repeat` is (conceptually) infinite: at some point n the difference between the list element a_{n+1} and its predecessor a_n is at most `eps`, which means that the value a_{n+1} is returned and, thus, the rest of the list is not evaluated.

The significance of sharing for this algorithm to perform efficiently can be observed in the infinite list generated by `repeat`:

$$a_0\ :\ f\ a_0\ :\ f(f\ a_0)\ :\ f(f(f\ a_0))\ :\ \ldots$$

If non-strict evaluation would not be accompanied by sharing, we would end up recomputing previous approximations in order to compute the next one. The issue lies in the right-hand side of `repeat`, which duplicates the argument `a`. Due to the non-strict semantics, this duplication is done *before* evaluating the expression bound to `a` upon calling `repeat`. Hence, when evaluating the $n+1$-st element of the list, instead of applying f to the *value* of the n-th element, we have to apply f to a_0 $n+1$ times.

Sharing avoids this issue by using pointers instead of duplication. This approach causes the infinite list to look as follows:

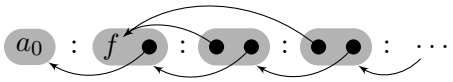


(a) Term `a : repeat f (f a)`. (b) A term graph representation.

Figure 3.1: Terms vs. term graphs.

Instead of having explicit copies of the previous element in the list, a pointer to that element is used. Once an element is evaluated, the resulting value is thus also available to the next element.

Note that for the sake of clarity we simplified the structure of sharing above. A more accurate picture of the sharing is the one below:



Also the occurrence of the function f is shared.

The sharing illustrated above is achieved by moving from a term representation to a *term graph* representation of expressions. For example, in order to achieve the sharing illustrated above, the right-hand side of the definition of `repeat` has to be represented as a term graph as shown in Figure 3.1b.

The introduction of infinitary term graph rewriting calculi that allow us to study both transfinite rewrite sequences and sharing is one of the main contributions of this part of the dissertation. More details about our approach are given in Section 3.3.

3.1.3 Cyclic Structures

Section 3.1.1 exemplified a general observation that can be found throughout mathematics and computer science: dealing with infinite objects is easier than dealing with finite representations thereof. Another example of this general principle is the use of infinite trees to represent the minimal relevant information contained in a λ-term. Examples of such structures are Berarducci trees [23], Lévy-Longo trees [114, 116] and Böhm trees [18]. Each of these representations capture the relevant meaning of a lambda term in a single tree, e.g. the Böhm trees of two λ-terms coincide iff they are equal according to Scotts's P_ω model [18].

Indeed, one of the early motivations for studying infinitary rewriting follows the same line of reasoning: instead of dealing with finite but cyclic term graphs, consider their unravelling to potentially infinite trees.

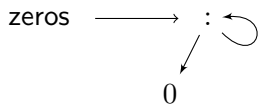
Term graph rewriting [20] may be viewed as a generalisation of term rewriting. It generalises from terms as the objects on which rewriting is performed to *term graphs*, which – as opposed to terms – allow each node to have more that one

parent node. Simply put, term graph rewriting is term rewriting extended with sharing.

To compare term graphs with ordinary terms, reconsider the definition of the function `repeat` from Section 3.1.2 and the representation of its right-hand side as term and as a term graph illustrated in Figure 3.1.

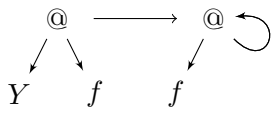
Term graph rewriting occurs quite naturally in the implementation of functional languages [21, 142, 147]. Expressions in functional languages are represented by pointer structures which allow the *sharing* of common subexpression. As discussed in Section 3.1.2, this sharing is the natural companion of non-strict evaluation, which we considered in Section 3.1.1.

The idea of using graphs to efficiently implement term rewriting goes back to Wadsworth [179]. Apart from the acyclic (or horizontal) sharing that avoids redundant evaluation of subexpressions [21], term graphs also offer cyclic (or vertical) sharing. Also the latter form of sharing is used in the implementation of functional programming languages [142] for optimising certain forms of recursively defined function. For example, the term rewrite rule $\mathsf{zeros} \to 0 : \mathsf{zeros}$ that we have seen in Chapter 1 may be represented as follows



Instead of replicating the left-hand side of the rule, viz. zeros, in right-hand side, an edge to the root of the right-hand side is used. Thus, the rule has to be applied only once; subsequent applications of the original rule are emulated by unfolding the cycle. Farmer and Watro [55, 56] coined the name *redex capturing* for this phenomenon, which they studied using infinitary term rewriting. As they point out, redex capturing does not only have positive aspects but may also lead to issues with garbage collection.

A more realistic example is the definition of the fixed point combinator Y in the form of the term rewrite rule $Y\ f \to f(Y\ f)$. Similarly to the example above, this rule may be represented by the following term graph rewrite rule:



Barendregt et al. [19] were the first to give term graph rewriting an operational semantics similar to that of term rewriting. There are two other major approaches to term graph rewriting: a category theoretic approach [42] and an equational approach [5]. But we shall focus on the operational approach as it is closer to the implementation of functional languages.

The appeal of infinitary term rewriting for studying term graph rewriting is twofold: (1) It allows us to study whether term graph rewriting faithfully implements term rewriting. (2) The unravelling of term graphs to potentially infinite terms avoids the hassle of dealing with explicit sharing. Both points are of course just two sides of the same coin, viz. the correspondence between cyclic term graph rewriting and infinitary term rewriting.

As long as we are confined to term graph rewriting with only acyclic sharing, the theory of term rewriting is sufficient [98]. However, as soon as cycles are introduced into term graphs, like in the abovementioned examples, finite terms and finite reduction sequences are not necessarily sufficient anymore. For instance, the term graph given on the right-hand side of the term graph rewrite rule for zeros unravels to the infinite term $0 : 0 : \ldots$. That means, a single rewrite step with that rule represents a transformation from the term zeros into the term $0 : 0 : \ldots$. In order to replicate this transformation with the original term rewrite rule, we have to perform infinitely many rewrite steps

$$\mathsf{zeros} \to 0 : \mathsf{zeros} \to 0 : 0 : \mathsf{zeros} \to \ \ldots$$

This infinite term rewrite sequence also transforms zeros into $0 : 0 : \ldots$.

Kennaway et al. [98] provide a thorough characterisation of the relationship between cyclic term graph rewriting and infinitary term rewriting. Following the notion of *rational terms*, which are (possibly infinite) terms that arise as unravelling of finite term graphs, they characterise *rational rewrite sequences*, which are (possibly infinite) rewrite sequences that can be adequately represented by finite cyclic term graph rewrite sequences. An alternative characterisation is provided by Corradini and Drewes [40] who present infinite parallel term reductions as the counterpart of cyclic term rewriting.

3.2 Notions of Convergence and All That

There are a number of different approaches to infinitary rewriting, which we shall review here. The majority of infinitary calculi is based on some notion of convergence derived from a topology on the objects of the rewriting system, but we shall also discus other approaches.

3.2.1 Metric Convergence

Infinitary term rewriting was initially introduced by Dershowitz et al. [47–49]. In this work, the authors use the metric space on terms based on the distance measure $\mathbf{d}(s,t)$ given by $\mathbf{d}(s,t) = 0$ if $s = t$ and $\mathbf{d}(s,t) = 2^{-d}$ otherwise, where d is the smallest depth at which s and t differ [7]. A rewrite sequence S, which produces some sequence of terms $(t_\iota)_{\iota<\alpha}$, converges to a term t if the sequence $(t_\iota)_{\iota<\alpha}$ converges to t in the metric space and at each limit ordinal $\lambda < \alpha$ the sequence $(t_\iota)_{\iota<\lambda}$ converges to t_λ. The latter condition ensures that rewrite sequences are *continuous* at limit ordinals. Intuitively, convergence in the metric space of terms means that the differences between the terms in the rewrite sequence are pushed deeper and deeper in the term structure as the sequence approaches a limit ordinal from below. Apart from a notion of convergence, the metric also provides a canonical construction of the set of finite and infinite terms from the set of finite terms by means of *metric completion*.

Shortly after this initial approach, Farmer and Watro [55, 56] used infinitary term rewriting to study cyclic term graph rewriting. They used, however, a stronger notion of convergence than Dershowitz et al. In addition to requiring convergence in the metric space of terms, Farmer and Watro require also the depth

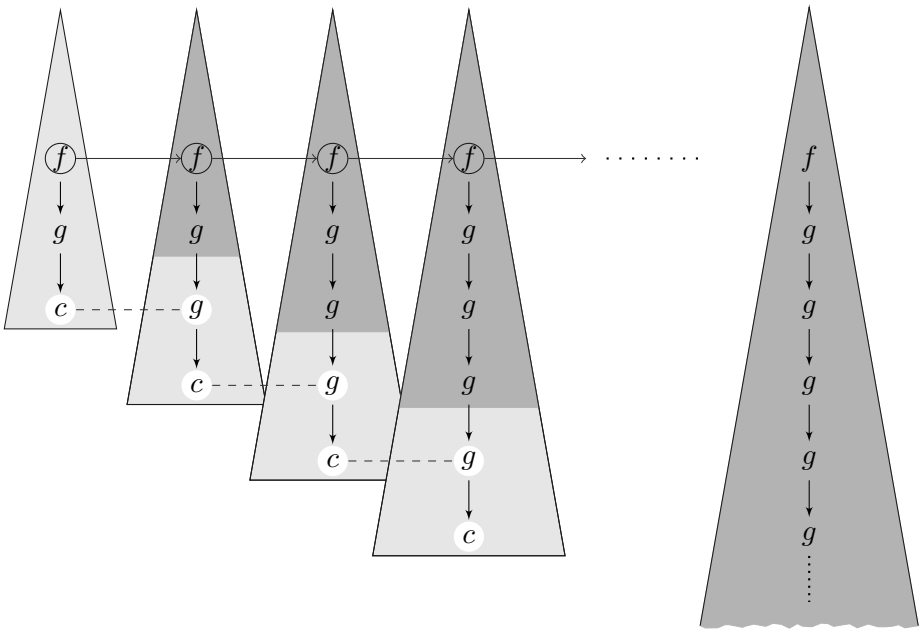


Figure 3.2: Weakly convergent reduction sequence.

of the contracted redexes to tend to infinity as the sequence approaches a limit ordinal from below. This notion of convergence, called *strong convergence* [99], is based on a conservative underapproximation of the convergence of Dershowitz et al., which in turn is referred to as *weak convergence*. Instead of only stipulating that the depth of differences between terms tends to infinity, strong convergence requires the depth of the contracted redex positions to tend to infinity. The fact that the latter implies the former makes strong convergence a conservative underapproximation of weak convergence that is somewhat independent from the actual result of contracting redexes.

Figure 3.2 and Figure 3.3 illustrate the difference between weak and strong convergence. Both figures show the same rewrite sequence; the only difference between them is the rewrite rule that is applied and the positions at which it is applied. The rewrite sequence in Figure 3.2 is generated by the rewrite rule $f(x) \to f(g(x))$, which adds a function symbol g *below* f. The rule is repeatedly applied at the root of the term. That means, the rewrite sequence cannot strongly converge – the depth of the contracted redexes is constantly 0 and thus does not tend to infinity. However, the rewrite sequence does weakly converge to the term $f(g(g(\dots)))$ as illustrated: the depth d_i of the differences between two consecutive terms t_i, t_{i+1} (indicated by dashed lines) tends to infinity, which means the metric distance $\mathbf{d}(t_i, t_{i+1}) = 2^{-d_i}$ between them tends to 0. In other words, the part of the term that remains unchanged in a rewrite step (indicated by a darker shade of grey) grows continuously in the course of the rewrite sequence.

In Figure 3.3, we instead use the rewrite rule $g(c) \to g(g(c))$, which adds a function symbol g *above* c. The sequence of terms that is generated in this rewrite sequence is the same as in Figure 3.2, but the contraction of redexes takes place at different positions. In particular, the depth of the contracted redexes (indicated by the rewrite arrows) tends to infinity. In other words, the part of the term that

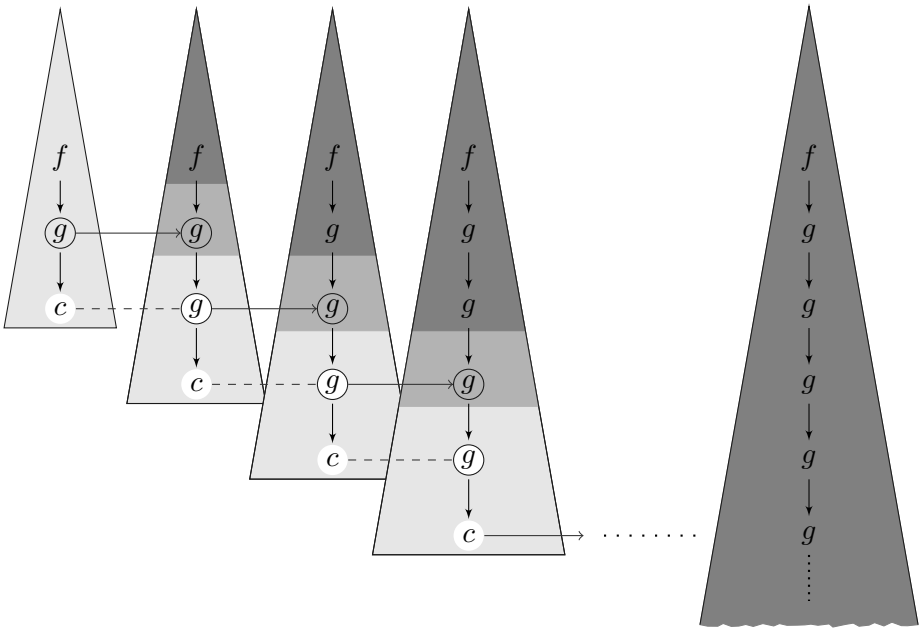


Figure 3.3: Strongly convergent reduction sequence.

is entirely untouched by rewriting (indicated by the yet darker shade of grey) grows continuously. Thus, the rewrite sequence not only weakly converges to $f(g(g(\dots)))$, it also strongly converges to $f(g(g(\dots)))$.

The majority of the literature on infinitary rewriting is based on these two notions of weak and strong convergence, with a particular focus on the latter. Strong convergence is arguably more intuitive than weak convergence: a node in the term can only contribute to the limit of an infinite rewrite sequence if it is not rewritten from some point on. Additionally, strong convergence has much better properties. The latter argument was made by Kennaway et al. in their seminal paper on strong convergence [99]. Strong convergence was subsequently also used to develop infinitary λ-calculi [100, 101] and infinitary higher-order term rewriting [103, 104].

One of the properties that sets apart weak and strong convergence is the *compression property* for left-linear systems: a strongly convergent rewrite sequence can be "compressed" into a strongly convergent rewrite sequence with the same start and end terms but with length at most ω. Also in terms of confluence properties, strong convergence is more well-behaved. However, unlike for finitary confluence, orthogonality is in general not sufficient in order to obtain infinitary confluence. The origin of this discrepancy was identified by Kennaway et al. [99] to lie in the presence of *collapsing* rules, which collapse a redex to one of its proper subterms.

The canonical example that exhibits this phenomenon is a rewriting system with the rules $\rho_1\colon f(x) \to x$ and $\rho_2\colon g(x) \to x$ and the infinite starting term $t = f(g(f(g(\dots))))$ that alternates between function symbols f and g. We can then produce two infinite rewrite sequences, which strongly converge to the term $t_1 = g(g(g(\dots)))$ respectively $t_2 = f(f(f(\dots)))$ by exhaustively applying ρ_1 respectively ρ_2. The resulting terms t_1 and t_2 can only be rewritten to themselves

and can, therefore, not be joined.

Kennaway et al. [99] furthermore recognised that terms containing an infinite tower of collapsing redexes, e.g. $f(g(f(g(\dots))))$ in the example given above, are the chief culprits for this failure of infinitary confluence: if these terms – named *hypercollapsing terms* – are equated, then the resulting calculus is in fact infinitarily confluent for orthogonal systems. The same idea was subsequently also used in infinitary λ-calculus to establish infinitary confluence modulo equality of so-called 0-active terms [101]. Both approaches were later generalised to a notion of meaningless terms [102]. The set of hypercollapsing terms as well as the set of 0-active terms are examples of such a set of meaningless terms. Sets of meaningless terms are defined axiomatically and yield a rewrite relation called *Böhm reduction*, which introduces new rules to a term rewriting system (respectively λ-calculus) that admits rewriting a meaningless term directly to $\bot$ (a fresh constant symbol). This approach effectively equates all elements of a set of meaningless terms. The resulting reduction is both infinitarily normalising and infinitarily confluent for orthogonal term rewriting systems as well as λ-calculus. Consequently, each term has a unique infinitary normal form. This normal form generalises the notion of Böhm-like trees (including Berarducci trees, Lévy-Longo trees etc.) [101]. Recently, the notion of meaningless terms has again received further attention [156–158].

An important feature that sets apart the work of Kennaway et al. [101] on λ-calculus from infinitary first-order rewriting is the use of different variants of the metric $\mathbf{d}$ described above. These variants differ only in the way in which the depth of a position in a term is measured: they simply discount certain edges. The resulting metric spaces yield metric completions of the set of finite terms that omit certain undesired infinite λ-terms. Depending on which metric is chosen, the resulting unique infinite normal forms of Böhm reduction correspond to different notions of Böhm-like trees.

3.2.2 Other Notions of Convergence

Other approaches to convergence have received less treatment in the literature. One approach closely related to the metric-based approaches from the previous section is Rodenburg's purely topological notion of convergence [152]. The topology Rodenburg uses is similar to the topology induced by the metric $\mathbf{d}$ on terms in Section 3.2.1 above. However, since Rodenburg also considers function symbols of infinite arity, he introduces a topology that not only considers the depth of a position in a term but also the number of siblings to the left of it. Given a rewrite rule $a \to b$, this topology causes the rewrite sequence

$$f(a,a,a,a,\dots) \to f(b,a,a,a,\dots) \to f(b,b,a,a,\dots) \to \dots$$

to converge to $f(b,b,b,b,\dots)$, whereas it would not converge in the metric space induced by the metric $\mathbf{d}$ (appropriately extended to signatures with infinite arity).

Corradini [39] introduced a notion of convergence based on the partial order $\leq_\bot$ defined on the set of *partial terms* – terms extended with a fresh constant symbol $\bot$. Intuitively, $s \leq_\bot t$ means that s is "less defined" than t; it is the least congruence relation satisfying $\bot \leq_\bot t$ for all partial terms t. Similarly to

the metric on terms, this partial order provides a canonical construction of the set of finite and infinite (partial) terms, namely *ideal completion* [24]. Corradini [39] uses the resulting complete partial order structure to define the outcome of contracting an infinite number of parallel redexes in a term. The underlying calculus of term rewriting is however non-standard and allows partial matching of left-hand sides. The resulting infinitary rewriting calculus provides an adequate counterpart for cyclic term graph rewriting [40]. Later, Corradini and Gadducci [41] generalised the calculus to the setting of rewriting logic.

Also Blom [25] uses a construction based on the partial order $\leq_\perp$ to define convergence. Although he fails to mention it, the construction Blom uses corresponds to the *limit inferior* w.r.t. the partial order $\leq_\perp$.[3] His notion of convergence provides an alternative to the Böhm reduction introduced by Kennaway et al. [101]. Instead of explicitly introducing rules that admit the contraction of meaningless terms to $\perp$, the limit inferior construction of the convergence maps such terms implicitly to $\perp$. Blom shows that his calculus indeed simulates the Böhm reductions of Kennaway et al. [101] faithfully.

Recognising this partial order approach to infinitary rewriting and exploring the merits of it as an alternative to the now-standard metric approach is one of the main contributions of this part of the dissertation. We shall come back to this point in Section 3.3.

3.2.3 Abstract Notions of Convergence

Since the inception of infinitary rewriting, several researchers have approached the subject from an abstract angle, abstracting from the term structure or from the mode of convergence. The first abstract treatment was pursued by Kennaway [96] who took the notion of strong convergence of Kennaway et al. [97] and abstracted from the term structure. The only structure of these *metric abstract reduction systems* is a metric distance on the objects and a height on the rewrite steps. When instantiated to terms rewriting systems, the metric distance becomes the metric distance $\mathbf{d}$ on terms and the height of a rewrite step becomes 2^{-d}, where d is the depth of the contracted redex.

Kahrs [93] investigates a more concrete abstraction in which he still maintains the term structure but considers a variety of metric distance measures on them, which he calls *term metrics*. The notion of convergence that he considers is purely topological and thus generalises weak convergence. In later work, Kahrs [94] considers a wide variety of alternative notions of convergence such as topological adherence (instead of convergence) and various closures of binary relations.

In his master's thesis Bongaerts [26] presents a comprehensive abstract framework of infinitary rewriting based purely on topological convergence. It covers a wide variety of notions of convergence known in the literature. Naturally, this topological framework generalises weak convergence. But, more interestingly, by defining an appropriate topology not only on the objects but also the rewrite steps in between, Bongaerts manages to capture strong convergence as well. Moreover, using the Scott topology of a partial order, he is also able to capture a partial

[3] This can be observed from Corollary 4.7 in Paper **B4**, which characterises the limit inferior for a partial order on term graphs that generalises $\leq_\perp$.

order convergence. While this notion of convergence is not precisely partial order convergence based on the limit inferior of the underlying order, it is a weakening thereof known as $\mathcal{S}$-*limit*.

3.3 Contributions of this Dissertation

3.3.1 Overview

In this dissertation we explore an alternative approach to infinitary rewriting, which instead of metric topologies is based on partial orders. Instead of the limit $\lim_{\iota\to\alpha} a_\iota$ in a metric space, we use the *limit inferior* $\liminf_{\iota\to\alpha} a_\iota$ of a partial order, which is defined as the least upper bound of the greatest lower bounds of all suffixes of $(a_\iota)_{\iota<\alpha}$:

$$\lim_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha} \left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \right)$$

To get an intuitive understanding of the limit inferior we assume that the partial order $\leq$ on objects denotes a form of "information inclusion", i.e. $a \leq b$ means that all the information encoded in a is also present in b. This reading applies, for example, to the partial order $\leq_\bot$ on terms that we discussed in Section 3.2.2.

Given this intuitive interpretation of $\leq$, the greatest lower bound $\bigsqcap A$ of a set of objects A is the object that encodes the maximum amount of information that objects in A agree on. For the partial order $\leq_\bot$ on terms we have for example $\bigsqcap\{f(a,b), f(a,g(b))\} = f(a,\bot)$. Dually, we have that the least upper bound $\bigsqcup A$ of a set of objects A is the object that encodes the union of the information encoded in the objects in A. Given this interpretation, the limit inferior $\lim_{\iota\to\alpha} a_\iota$ of a sequence $(a_\iota)_{\iota<\alpha}$ is the object that contains a piece of information iff that piece of information is present in each element of the sequence from one point $\beta < \alpha$ on. This being the case, the limit inferior is a device that extracts from a sequence of objects as much *consistent* information as possible.

In general, the limit inferior is not guaranteed to exist for every sequence. However, the partial orders we consider – including $\leq_\bot$ – each form a so-called *complete semilattice*, a class of partially ordered sets for which the limit inferior always exists. Consequently, every (continuous) reduction converges in this partial order model of convergence. This observation, of course, begs the question: What good is a mode of convergence that makes everything converge?

While each rewrite sequence converges, the outcome of such a convergent rewrite sequence indicates the "degree of convergence". For example, consider the rewrite sequence induced by `zeros`:

$$\mathsf{zeros} \to 0 : \mathsf{zeros} \to 0 : 0 : \mathsf{zeros} \to \dots$$

This sequence converges to the term $0 : 0 : \dots$ also in the partial order model. On the other hand, considering the rule $\mathsf{swap}(x,y) \to \mathsf{swap}(y,x)$, we obtain the rewrite sequence

$$\mathsf{swap}(0,1) \to \mathsf{swap}(1,0) \to \mathsf{swap}(0,1) \to \dots$$

which does not converge in the metric model of convergence but it does converge to $\mathsf{swap}(\bot, \bot)$ in the partial order model. The two occurrences of $\bot$ indicate positions at which the rewrite sequence diverges. In general, the partial order model allows us to distinguish the degree to which a rewrite sequence converges, where the least element of the partially ordered set indicates complete divergence and maximal elements indicate complete convergence.

The goal of this part of the dissertation is to explore this new approach, compare it with the established metric approach, and identify potential benefits and drawbacks.

3.3.2 Concrete Contributions

In order to build a foundation on which to build the partial order approach, Paper **B1** introduces an abstract notion of partial order convergence based on the limit inferior and compares it with the metric approach. To this end, an abstract axiomatic notion of infinitary reduction systems – called *transfinite abstract reduction systems* (*TARSs*) – is introduced. We show that finitary abstract reduction systems are a trivial instances of such TARSs. But more importantly, we introduce both *metric reduction systems* (*MRSs*) – based on the metric abstract reduction systems of Kennaway [96] – and *partial reduction systems* (*PRSs*). MRSs abstract from the metric notion of convergence found in the literature both in its weak and strong variant. PRSs induce corresponding notions of convergence based on partially ordered sets and their limit inferior. Also for PRSs we consider a weak and a strong variant. Each of the two abstract models induces two TARSs, one for weak and one for strong convergence.

We generalise some of the basic theorems about confluence and termination properties known for finitary abstract reduction systems to all of these infinitary systems by proving them for the axiomatically defined TARSs. We also show correspondences between strong and weak notions of convergence. Both MRSs and PRSs are instantiated to term rewriting systems; the instantiation of MRSs yields the standard notions of weak and strong convergence from the literature. The instantiation of PRSs to lambda calculus with the partial order $\leq_\bot$ yields the ad hoc construction of Blom [25] as strong convergence.[4] All calculi presented in Paper **B2** to Paper **B5** are instantiations of MRSs and PRSs.

Paper **B2** explores the instantiation of PRSs to term rewriting in detail. In order to distinguish the resulting partial order-based notions of convergence from the standard metric-based ones, the former is referred to as *p-convergence* and the latter as *m-convergence*. The main focus of the paper is the concrete relation between p- and m-convergence. The take-away message from this paper is that p-convergence is a conservative extension of m-convergence – both for the weak and the strong variant. That is, the difference between p- and m-convergence lies solely in the terms that contain '$\bot$'s, which are precisely those terms that p-convergence adds compared to m-convergence. However, our results show that the correspondence goes even deeper: in orthogonal systems, the Böhm extension

[4] This holds true for the 111 depth measure. For other xyz depth measures, a corresponding different partial order has to be considered. The resulting limit inferior of these partial orders is however different from the base construction of Blom [25] who uniformly uses the limit inferior of the subset partial ordering regardless of the xyz depth measure.

of strong m-convergence, which essentially adds a rule $t \to \bot$ for each so-called *root-active* term, coincides with strong p-convergence. As a corollary we thus obtain that, for orthogonal systems, strong p-convergence – unlike strong m-convergence – is infinitarily normalising and confluent.

Note that for orthogonal term rewriting systems, the set of root-active terms is the *smallest* set of meaningless terms [102]. Thus, both strong p-convergence – with the intuition we described in Section 3.3.1 – and the Böhm extension of strong m-convergence w.r.t. *root-active* terms extract as much (consistent) information from an infinite rewrite sequence as possible. Our result shows that both approaches agree on what the outcome of this extraction is.

The remaining three papers aim to extend infinitary rewriting to term graph rewriting using both the metric and the partial order approach by instantiating MRSs respectively PRSs to term graph rewriting systems. In Section 3.1.2, we have seen an example that shows the need to be able to reason about infinite computations and structures as well as sharing, viz. in the context of lazy evaluation.

Paper **B3** explores weak notions of convergence derived from a metric and a partial order on term graphs that generalises $\mathbf{d}$ and $\leq_\bot$, respectively. We discuss several alternatives for suitable metrics and partial orders and compare their relative merits. After that we identify a particular metric and a particular partial order as the favourable approach. For this pair of structures, we obtain a correspondence result similar to the ones found in Paper **B2**: weak p-convergence is a conservative extension of weak m-convergence. We also show a basic soundness property of the resulting infinitary term graph rewriting calculi w.r.t. infinitary term rewriting.

Unfortunately, these properties of the two modes of convergence are bought dearly: (1) the definitions of the metric and partial order are cumbersome, (2) the soundness property is very weak, and (3) there is no clear path to extend the weak notions of convergence to strong ones in order to gain better soundness and completeness properties.

In order to obtain infinitary calculi of strong convergence, which might then help us to establish better soundness and completeness properties, we backtrack and reconsider a pair of candidates that we have dismissed in Paper **B3** due to their ostensible inappropriateness for week convergence. This pair of metric and partial order, which we call *simple* to distinguish it from the corresponding structures in Paper **B3**, which we call *rigid*, are studied in Paper **B4** and Paper **B5**.

In Paper **B4** we still consider weak convergence only. We illustrate the issues that the simple partial order causes (in comparison to the rigid partial order of Paper **B3**). In particular, we show that we do not obtain the correspondence result that weak p-convergence conservatively extends weak m-convergence. However, we show that we get at least one direction of this correspondence, viz. weak m-convergence implies weak p-convergence. Moreover, we can replicate the soundness result of Paper **B3**. More importantly, though, we show that both structures have better algebraic properties than the rigid structures of Paper **B3**. This manifests itself particularly in the fact that both metric completion and ideal completion yield the full set of (partial) infinitary term graphs, which is not

true for the rigid metric respectively partial order.

The true shine of these simple structures is revealed in Paper **B5** in which we then finally extend the weak notions of convergence from Paper **B4** to strong convergence. This move from weak to strong convergence turns out to eradicate all the issues that we have identified for weak convergence in Paper **B4**. Strong p-convergence does not have the unintuitive quirks that we have observed for weak p-convergence. This fact is also manifested in the full correspondence result between strong p-convergence and m-convergence: strong p-convergence is a conservative extension of strong m-convergence. We obtain a full soundness theorem for both strong m- and p-convergence stating that strongly converging term graph reductions can be simulated by strongly converging term reductions. We also obtain completeness properties for both calculi. For strong m-convergence this completeness result is rather limited, but this was expected since Kennaway et al. [98] already gave a counterexample for the full completeness property. Astonishingly, however, strong p-convergence defies this counterexample and indeed has the full completeness property in the form given as part the notion of adequacy of Kennaway et al. [98].

Our treatment of infinitary term graph rewriting is the first formalisation of a fully infinitary calculus of term graph rewriting. Kennaway et al. [98] introduce infinitary term graph rewriting informally in order to present the abovementioned counterexample for its completeness w.r.t. infinitary term rewriting. Kennaway et al. take this failure of completeness as an argument to not consider infinitary rewriting any further. Our formal treatment of strong m-convergence confirms their counterexample. However, we were able to show that strong m-convergence is in fact complete if we only consider normalising reductions. What is more, if we consider strong p-convergence, we do in fact obtain a full completeness result.

Although our formal treatment of infinitary term graph rewriting is the first one in this vein, infinitary features have been studied in the context of calculi with explicit sharing before. Ariola and Blom [3, 4] developed a notion of skew confluence for λ-calculi with *letrec* that allows them to define infinite normal forms similar to Böhm trees. This construction of infinite normal forms reconciles the issue that these λ-calculi fail to be confluent [6].

3.4 Conclusions and Perspectives

We believe that our partial order approach provides a valuable alternative to the metric approach to infinitary rewriting. As we have argued in this chapter, our approach makes it possible to recognise degrees of convergence. Admittedly, as we have shown in Paper **B2**, the theory of meaningless terms and Böhm reduction does achieve this as well. To some extend, the two approaches – partial order convergence and Böhm reduction – complement each other. The former provides an intrinsic justification: the limit inferior collects the maximum amount of information possible. The latter is parametrised by a set of meaningless terms and is thus much more general.

The generality of Böhm reductions comes at a cost though: we have to move from a rewrite system with finitely many rules to one with infinitely many rules in order to replicate partial convergence on top of the metric convergence framework.

Moreover, on its own terms, it is difficult to intuitively justify the additional rules provided the Böhm reductions. After all, the main motivation behind the axioms that define the notion of meaningless terms seems to be that "it makes the proof of confluence go through".[5] This is a perfectly valid motivation, to be sure, but it does not give justification for Böhm reduction beyond that. In this regard, our correspondence result for Böhm reductions in Paper **B2** can be seen as additional justification for the merit of Böhm reductions – at least w.r.t. the smallest set of meaningless terms. It would be interesting to explore whether it is possible to extend this correspondence to other sets of meaningless terms by varying the partial order.

We should reiterate the importance of the correspondence properties that we have established for term rewriting (Paper **B2**) as well as term graph rewriting (Paper **B5**), which state that p-convergence is a conservative extension of m-convergence. In other words, when moving from m-convergence to p-convergence, we do not lose anything.

In our work, we have explored several ways to formalise a notion of infinitary term *graph* rewriting. After evaluating our findings and experience in this realm, we believe that the approach introduced in Paper **B5** – based on the simple metric and partial order together with strong convergence – is the preferable one for most purposes. It provides a calculus that is essentially infinitary term rewriting + sharing. Paper **B5** illustrates this point with soundness and completeness theorems. Having established a satisfactory notion of infinitary term graph rewriting, we can now begin studying its properties, such as confluence and normalisation. Early results of our current work indicate that confluence properties of orthogonal systems carry over from infinitary term rewriting without too much hassle. Another interesting question is in what way the theory of meaningless terms can be translated to the realm of infinitary term graph rewriting.

A direction that we did not pursue here is higher-order systems including λ-calculus in particular. From the point of view of convergence, the study of infinitary λ-calculi offers an interesting detail that sets it apart from first-order infinitary term rewriting. In their work on infinitary λ-calculi, Kennaway et al. [101] consider a family of different metric spaces, which deviate from the ordinary metric $\mathbf{d}$ only by discounting some edges of a term when calculating the depth of a node. Blom [25] is able to capture the resulting notions of convergence with his approximation-based approach. Blom [25] uses an ad hoc construction that cuts off subterms depending on the metric that it should replicate in terms of convergence. Results of our current work show that it is possible to construct a family of partial orders corresponding to the family of metric spaces of Kennaway et al. [101]. We then obtain correspondences between p-convergence and Böhm reduction, but these correspondences are not as clear-cut as in the first-order case (cf. Paper **B2**) or as in Blom's approach [25].

[5] For example, Severi and de Vries [156] propose to weaken one of the axioms by showing that confluence and normalisation can still be proven using the weakened axiom instead.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[2] J. Andersen and C. Brabrand. Syntactic Language Extension via an Algebra of Languages and Transformations. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, pages 19–35, 2010. doi: 10.1016/j.entcs.2010.08.029.

[3] Z. Ariola and S. Blom. Skew and ω-Skew Confluence and Abstract Böhm Semantics. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838 of *Lecture Notes in Computer Science*, pages 368–403. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30911-6. doi: 10.1007/11601548˙19.

[4] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117(1-3):95–168, 2002. ISSN 0168-0072. doi: 10.1016/S0168-0072(01)00104-X.

[5] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3-4):207–240, 1996. ISSN 0169-2968. doi: 10.3233/FI-1996-263401.

[6] Z. M. Ariola and J. W. Klop. Lambda Calculus with Explicit Recursion. *Information and Computation*, 139(2):154–233, 1997. ISSN 0890-5401. doi: 10.1006/inco.1997.2651.

[7] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[8] E. Axelsson. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, pages 323–334, New York, NY, USA, 2012. ACM. doi: 10.1145/2364527.2364573.

[9] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[10] P. Bahr. Partial Order Infinitary Term Rewriting and Böhm Trees. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.67.

[11] P. Bahr. Modes of Convergence for Term Graph Rewriting. In M. Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 139–154, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.RTA.2011.139.

[12] P. Bahr. Modes of Convergence for Term Graph Rewriting. *Logical Methods in Computer Science*, 8(2):6, 2012. doi: 10.2168/LMCS-8(2:6)2012.

[13] P. Bahr. Modular Tree Automata. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 263–299. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-31113-0˙14.

[14] P. Bahr. Convergence in Infinitary Term Graph Rewriting Systems is Simple. Submitted to Math. Struct. in Comp. Science, 2012.

[15] P. Bahr. Infinitary Term Graph Rewriting is Simple, Sound and Complete. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69–84, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2012.69.

[16] P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, New York, NY, USA, 2011. ACM. doi: 10.1145/2036918.2036930.

[17] P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–24. Open Publishing Association, 2012. doi: 10.4204/EPTCS.76.3.

[18] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathemantics*. Elsevier Science, revised ed edition, 1984.

[19] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In P. C. T. de Bakker A. J. Nijman, editor, *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer*

Science, pages 141–158. Springer Berlin / Heidelberg, 1987. doi: 10.1007/3-540-17945-3˙8.

[20] E. Barendsen. Term Graph Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 13, pages 712–743. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[21] E. Barendsen and S. Smetsers. Graph rewriting aspects of functional programming. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, volume 2, pages 63–102. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. ISBN 981-02-4020-1.

[22] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986. ISSN 0001-0782. doi: 10.1145/6424.315691.

[23] A. Berarducci. Infinite λ-calculus and non-sensible models. In A. Ursini and P. Aglianó, editors, *Logic and algebra*, number 180 in Lecture Notes in Pure and Applied Mathematics, pages 339–378. CRC Press, 1996.

[24] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 215–226, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512971.

[25] S. Blom. An Approximation Based Approach to Infinitary Lambda Calculi. In V. van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin / Heidelberg, 2004. doi: 10.1007/b98160.

[26] J. Bongaerts. Topological Convergence in Infinitary Abstract Rewriting. Master's thesis, Utrecht University, 2011.

[27] C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. In *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 393–416, 2005. doi: 10.1016/j.entcs.2004.06.019.

[28] K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In E. Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer Berlin / Heidelberg, 1998. doi: 10.1007/BFb0054106.

[29] K. B. Bruce. Some Challenging Typing Issues in Object-Oriented Languages: Extended Abstract. In *Proceedings of the Workshop on Object Oriented Developments*, volume 82 of *Electronic Notes in Theoretical Computer Science*, pages 1–29, 2003. doi: 10.1016/S1571-0661(04)80799-0.

[30] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009. doi: 10.1017/S0956796809007205.

[31] F. Chalub and C. Braga. Maude MSOS Tool. In *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 133–146, 2007. doi: 10.1016/j.entcs.2007.06.012.

[32] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. In *Proceedings of the ninth annual Conference on Object-Oriented Programming Systems, Language, and Applications*, pages 1–15, New York, NY, USA, 1994. ACM. doi: 10.1145/191080.191083.

[33] J. Cheney. Scrap your nameplate (functional pearl). In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 180–191, New York, NY, USA, 2005. ACM. doi: 10.1145/1086365.1086389.

[34] J. Cheney and C. Urban. αProlog: A Logic Programming Language with Names, Binding and α-Equivalence. In B. Demoen and V. Lifschitz, editors, *International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer Berlin / Heidelberg, 2004. doi: 10.1007/978-3-540-27775-0˙19.

[35] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 143–156, New York, NY, USA, 2008. ACM. doi: 10.1145/1411204.1411226.

[36] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, New York, NY, USA, 2000. ACM. doi: 10.1145/353171.353181.

[37] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on `http://www.grappa.univ-lille3.fr/tata`, 2008.

[38] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer Berlin / Heidelberg, 1985. doi: 10.1007/3-540-15983-5˙13.

[39] A. Corradini. Term rewriting in CTΣ. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 468–484. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56610-4˙83.

[40] A. Corradini and F. Drewes. Term Graph Rewriting and Parallel Term Rewriting. In *TERMGRAPH*, pages 3–18, 2011. doi: 10.4204/EPTCS.48.3.

[41] A. Corradini and F. Gadducci. CPO models for infinite term rewriting. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 368–384. Springer Berlin / Heidelberg, 1995. doi: 10.1007/3-540-60043-4˙65.

[42] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. University of Pisa, 1997. ISBN 9810228848.

[43] L. Day and G. Hutton. Towards Modular Compilers For Effects. In *Proceedings of the Symposium on Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, Madrid, Spain, 2011. Springer-Verlag. doi: 10.1007/978-3-642-32037-8˙4.

[44] C. de Braga, E. Haeusler, J. Meseguer, and P. Mosses. Mapping Modular SOS to Rewriting Logic. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, page 957. Springer Berlin / Heidelberg, 2003. doi: 10.1007/3-540-45013-0˙21.

[45] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381 – 392, 1972. ISSN 1385-7258. doi: 10.1016/1385-7258(72)90034-0.

[46] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-Theory à la Carte. To appear at POPL '13, 2013.

[47] N. Dershowitz and S. Kaplan. Rewrite, rewrite, rewrite, rewrite, rewrite... In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–259, New York, NY, USA, 1989. ACM. doi: 10.1145/75277.75299.

[48] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Infinite normal forms. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium*, volume 372 of *Lecture Notes in Computer Science*, pages 249–262. Springer Berlin / Heidelberg, 1989. doi: 10.1007/BFb0035765.

[49] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90040-9.

[50] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47 (1):3–36, 2003. ISSN 01676423. doi: 10.1016/S0167-6423(02)00107-7.

[51] G. D. P. Dueck and G. V. Cormack. Modular Attribute Grammars. *The Computer Journal*, 33(2):164–172, 1990. doi: 10.1093/comjnl/33.2.164.

[52] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.02.003.

[53] J. Engelfriet. Three hierarchies of transducers. *Mathematical Systems Theory*, 15(2):95–125, 1982. ISSN 1432-4350. doi: 10.1007/BF01786975.

[54] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71 – 146, 1985. ISSN 0022-0000. doi: 10.1016/0022-0000(85)90066-2.

[55] W. M. Farmer and R. J. Watro. Redex capturing in term graph rewriting. *International Journal of Foundations of Computer Science*, 1:369–386, 1990. ISSN 0129-0541. doi: 10.1142/S0129054190000266.

[56] W. M. Farmer and R. J. Watro. Redex capturing in term graph rewriting (concise version). In *Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 13–24. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3-540-53904-2˙82.

[57] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: support for modularity in translator design and implementation. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 223–234, New York, NY, USA, 1992. ACM. doi: 10.1145/143165.143210.

[58] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, New York, NY, USA, 1996. ACM. doi: 10.1145/237721.237792.

[59] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90014-7.

[60] M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming language Design and Implementation*, pages 236–248, New York, NY, USA, 1998. ACM. doi: 10.1145/277650.277730.

[61] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 157–170, New York, NY, USA, 1984. ACM. doi: 10.1145/502874.502890.

[62] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and Extensible Notations for Modeling Languages. In M. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in*

Computer Science, pages 102–116. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-71289-3˙9.

[63] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 230. Baltimore, 1998.

[64] J. Garrigue. Code Reuse Through Polymorphic Variants. In *Workshop on Foundations of Software Engineering*, 2000.

[65] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56625-2˙11.

[66] J. Gibbons. Polytypic downwards accumulations. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 207–233. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64591-7. doi: 10.1007/BFb0054292.

[67] J. Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37(1-3):37–65, 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00022-2.

[68] J. Grosch and H. Emmelmann. A tool box for compiler construction. In D. Hammer, editor, *Compiler Compilers*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3-540-53669-8˙77.

[69] Y. Gurevich and J. Morris. Algebraic operational semantics and modula-2. In E. Börger, H. Büning, and M. Richter, editors, *CSL '87*, volume 329 of *Lecture Notes in Computer Science*, pages 81–101. Springer Berlin / Heidelberg, 1988. ISBN 978-3-540-50241-8. doi: 10.1007/3-540-50241-6˙31.

[70] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8.

[71] W. Harrison and S. Kamin. Metacomputation-Based Compiler Architecture. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 213–229. Springer Berlin / Heidelberg, 2000. doi: 10.1007/10722010˙14.

[72] W. L. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD dissertation, University of Illinois at Urbana-Champaign, 2001.

[73] W. L. Harrison and S. N. Kamin. Modular Compilers Based on Monad Transformers. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 122–131, Washington, DC, USA, 1998. IEEE Computer Society. doi: 10.1109/ICCL.1998.674163.

[74] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical Views on Computations on Trees (Extended Abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 619–630. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73420-8˙54.

[75] G. Hedin. An Object-Oriented Notation for Attribute Grammars. In *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming*, BCS Workshop Series, pages 329–345. Cambridge University Press, 1989.

[76] G. Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3): 301–317, 2000.

[77] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 95–103, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811543.

[78] F. Henglein, K. F. Larsen, J. G. Simonsen, and C. Stefansen. POETS: process-oriented event-driven transaction system. *The Journal of Logic and Algebraic Programming*, 78:381–401, 2009. ISSN 1567-8326. doi: 10.1016/j.jlap.2008.08.007.

[79] R. Hinze. Adjoint folds and unfolds—An extended study. *Science of Computer Programming*, 2012. ISSN 0167-6423. doi: 10.1016/j.scico.2012.07.011. In Press.

[80] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

[81] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(03):323–343, 1992. doi: 10.1017/S0956796800000411.

[82] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(04):437–444, 1998. doi: 10.1017/S0956796898003050.

[83] T. Hvitved, P. Bahr, and J. Andersen. Domain-Specific Languages for Enterprise Systems. Technical report, Department of Computer Science, University of Copenhagen, 2011.

[84] J. Iversen and P. D. Mosses. Constructive Action Semantics for Core ML. *IEE Proceedings - Software*, 152(2):79–98, 2005. doi: 10.1049/ip-sen:20041182.

[85] M. Jaskelioff. Modular Monad Transformers. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 64–79, Berlin, Heidelberg, 2009. Springer-Verlag. doi: 10.1007/978-3-642-00590-9˙6.

[86] M. Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[87] M. Jaskelioff. Monatron: An Extensible Monad Transformer Library. In S.-B. Scholz and O. Chitil, editors, *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248, Berlin, Heidelberg, 2011. Springer-Verlag. doi: 10.1007/978-3-642-24452-0˙13.

[88] M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and Implementation of Mathematical Operational Semantics. *Proceedings of the Second Workshop on Mathematically Structured Functional Programming*, 229(5):75–95, 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.02.017.

[89] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL '08*, pages 297–308, New York, New York, USA, 2008. ACM Press. doi: 10.1145/1328438.1328475.

[90] M. Jourdan, C. Le Bellec, D. Parigot, and G. Roussel. Specification and implementation of grammar couplings using attribute grammars. In M. Bruynooghe and J. Penjam, editors, *Progamming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 123–136. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-57186-8˙75.

[91] C. Jürgensen. *Categorical semantics and composition of tree transducers.* PhD thesis, Technischen Universität Dresden, 2003.

[92] C. Jürgensen and H. Vogler. Syntactic composition of top-down tree transducers is short cut fusion. *Mathematical Structures in Computer Science*, 14(2):215–282, 2004. ISSN 0960-1295. doi: 10.1017/S0960129503004109.

[93] S. Kahrs. Infinitary rewriting: meta-theory and convergence. *Acta Informatica*, 44(2):91–121, 2007. ISSN 0001-5903 (Print) 1432-0525 (Online). doi: 10.1007/s00236-007-0043-2.

[94] S. Kahrs. Infinitary Rewriting: Foundations Revisited. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 161–176, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.161.

[95] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994. ISSN 0001-5903. doi: 10.1007/BF01177548.

[96] R. Kennaway. On transfinite abstract reduction systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

[97] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. In R. V. Book, editor, *Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3-540-53904-2˙81.

[98] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. ISSN 0164-0925. doi: 10.1145/177492.177577.

[99] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[100] R. Kennaway, J. W. Klop, R. Sleep, and F.-J. de Vries. Infinitary lambda calculi and Böhm models. In J. Hsiang, editor, *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 257–270. Springer Berlin / Heidelberg, 1995. doi: 10.1007/3-540-59200-8˙62.

[101] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)00171-5.

[102] R. Kennaway, V. van Oostrom, and F.-J. de Vries. Meaningless Terms in Rewriting. *Journal of Functional and Logic Programming*, 1999(1):1–35, 1999.

[103] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems. In J. Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin / Heidelberg, 2005. doi: 10.1007/b135673.

[104] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems. *Information and Computation*, 209(6):893–926, 2011. ISSN 0890-5401. doi: 10.1016/j.ic.2011.01.007.

[105] S. Keuchel and J. T. Jeuring. Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, pages 57–68, New York, NY, USA, 2012. ACM. doi: 10.1145/2364394.2364403.

[106] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968. ISSN 1432-4350. doi: 10.1007/BF01692511.

[107] K. Koskimies, K.-J. Räihä, and M. Sarjakoski. Compiler construction using attribute grammars. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 153–159, New York, NY, USA, 1982. ACM. doi: 10.1145/800230.806991.

[108] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Berlin / Heidelberg, 1998. doi: 10.1007/BFb0054088.

[109] A. Kühnemann. Benefits of Tree Transducers for Optimizing Functional Programs. In V. Arvind and S. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, page 1046. Springer Berlin / Heidelberg, 1998. doi: 10.1007/978-3-540-49382-2˙13.

[110] D. Leijen and E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[111] S. Liang. *Abstract Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1997.

[112] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In H. Nielson, editor, *Programming Languages and Systems - ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer Berlin / Heidelberg, 1996. doi: 10.1007/3-540-61055-3˙39.

[113] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, 1995. ACM. doi: 10.1145/199448.199528.

[114] G. Longo. Set-theoretical models of λ-calculus: theories, expansions, isomorphisms. *Annals of pure and applied logic*, 24(2):153–188, 1983.

[115] C. K. K. Loverdos and A. Syropoulos. *Steps in Scala: An Introduction to Object-Functional Programming*. Cambridge University Press, 2010. ISBN 0521747589.

[116] J.-J. Lévy. An Algebraic Interpretation of the $\lambda\beta$K-Calculus; and an Application of a Labelled λ-Calculus. *Theoretical Computer Science*, 2(1): 97–114, 1976. doi: 10.1016/0304-3975(76)90009-8.

[117] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 133–144, New York, NY, USA, 2006. ACM. doi: 10.1145/1140335.1140352.

[118] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 397–406, New York, NY, USA, 1989. ACM. doi: 10.1145/74877.74919.

[119] S. Marlow. Haskell 2010 Language Report, 2010.

[120] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: new-age components for old-fasioned Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222, New York, NY, USA, 2001. ACM. doi: 10.1145/504282.504298.

[121] E. Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[122] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the seventh International Conference on Functional Programming languages and computer architecture*, pages 324–333, New York, NY, USA, 1995. ACM. doi: 10.1145/224164.224225.

[123] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961˙7.

[124] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Multiple Attribute Grammar Inheritance. *Informatica (Slovenia)*, 24(3):319–328, 2000.

[125] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892.

[126] J. Meseguer and C. O. Braga. Modular Rewriting Semantics of Programming Languages. In *Algebraic Methodology and Software Technology (AMAST)*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer Berlin / Heidelberg, 2004. doi: 10.1007/b98770.

[127] J. Meseguer and G. Rocu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.018.

[128] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press. doi: 10.1109/LICS.1989.39155.

[129] E. Moggi. An Abstract View of Programming Languages. Technical report, Edinburgh University, 1989.

[130] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4.

[131] P. Mosses. Foundations of Modular SOS. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer Berlin / Heidelberg, 1999. doi: 10.1007/3-540-48340-3˙7.

[132] P. Mosses. Constructive Action Semantics in OBJ. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation*, volume 4060 of *Lecture Notes in Computer Science*, pages 281–295. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-35462-8. doi: 10.1007/11780274˙15.

[133] P. Mosses. VDM semantics of programming languages: combinators and monads. *Formal Aspects of Computing*, 23(2):221–238, 2011. ISSN 0934-5043. doi: 10.1007/s00165-009-0145-4.

[134] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(0):195–228, 2004. ISSN 1567-8326. doi: 10.1016/j.jlap.2004.03.008.

[135] P. D. Mosses. Component-Based Description of Programming Languages. In *Proceedings of the 2008 international conference on Visions of Computer Science: BCS International Academic Conference*, pages 275–286, Swinton, UK, UK, 2008. British Computer Society.

[136] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998. ISBN 3540634061.

[137] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 21–36, New York, NY, USA, 2006. ACM. doi: 10.1145/1167473.1167476.

[138] B. Oliveira and W. Cook. Extensibility for the Masses. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-31057-7˙2.

[139] B. C. d. S. Oliveira, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, pages 199–216, 2006.

[140] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society. doi: 10.1109/CMPSAC.1998.716629.

[141] W. Penczek and A. Szalas, editors. *Theory and practice of action semantics*, volume 1113 of *Lecture Notes in Computer Science*, 1996. Springer Berlin / Heidelberg. ISBN 978-3-540-61550-7. doi: 10.1007/3-540-61550-4˙139.

[142] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. ISBN 013453333X.

[143] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM. doi: 10.1145/53990.54010.

[144] R. Pickering. *Beginning F#*. Apress, 2010. ISBN 1430223898.

[145] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006. doi: 10.1145/1147954.1147961.

[146] A. M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 21(03):235–286, 2011. doi: 10.1017/S0956796811000116.

[147] R. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0201416638.

[148] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.

[149] F. Pottier. An Overview of Cαml. In *Proceedings of the ACM-SIGPLAN Workshop on ML*, volume 148, pages 27 – 52, 2006. doi: 10.1016/j.entcs.2005.11.039.

[150] N. Pouillard. *Namely, Painless: A unifying approach to safe programming with first-order syntax with binders*. PhD thesis, Université Paris Diderot (Paris 7), 2012.

[151] N. Pouillard and F. Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 217–228, New York, NY, USA, 2010. ACM. doi: 10.1145/1863543.1863575.

[152] P. H. Rodenburg. Termination and Confluence in Infinitary Term Rewriting. *The Journal of Symbolic Logic*, 63(4):1286–1296, 1998. ISSN 00224812.

[153] J. Saraiva and D. Swierstra. Generic Attribute Grammars. In *Second Workshop on Attribute Gramars and their Applications*, pages 185–204, 1999.

[154] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 32–44, New York, NY, USA, 2011. ACM. doi: 10.1145/2034773.2034781.

[155] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00418-7.

[156] P. Severi and F.-J. de Vries. Weakening the Axiom of Overlap in Infinitary Lambda Calculus. In M. Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 313–328, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2011.313.

[157] P. Severi and F.-J. de Vries. Decomposing the Lattice of Meaningless Sets in the Infinitary Lambda Calculus. In L. Beklemishev and R. de Queiroz, editors, *Logic, Language, Information and Computation*, volume 6642 of *Lecture Notes in Computer Science*, pages 210–227. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-20920-8˙22.

[158] P. Severi and F.-J. de Vries. Meaningless Sets in Infinitary Combinatory Logic. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 288–304, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2012.288.

[159] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, New York, NY, USA, 2003. ACM. doi: 10.1145/944705.944729.

[160] G. L. Steele Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, New York, NY, USA, 1994. ACM. doi: 10.1145/174675.178068.

[161] S. Swierstra. Combinator Parsers: From Toys to Tools. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*, pages 38 – 59, 2000. doi: 10.1016/S1571-0661(05)80545-6.

[162] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758.

[163] T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 197–208, New York, NY, USA, 1990. ACM. doi: 10.1145/93542.93567.

[164] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[165] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–44. Springer Berlin / Heidelberg, 2004. doi: 10.1007/978-3-540-24851-4˙6.

[166] D. Turi and G. D. Plotkin. Towards a Mathematical Operational Semantics. In *LICS '97 Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 280–291, 1997.

[167] A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998. ISSN 1096-908X. doi: 10.1002/(SICI)1096-908X(199803/04)10:2¡75::AID-SMR168¿3.0.CO;2-5.

[168] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 137–165. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45937-5˙11.

[169] V. Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, Estonia, 2000.

[170] M. Viera and D. Swierstra. Attribute Grammar Macros. In F. de Carvalho Junior and L. Barbosa, editors, *Proc. Brazilian Symposium on Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 150–164. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-33182-4˙12.

[171] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming - ICFP '09*, page 245, New York, New York, USA, 2009. ACM Press. doi: 10.1145/1596550.1596586.

[172] M. Viera, S. D. Swierstra, and A. Middelkoop. UUAG Meets AspectAG: How to make Attribute Grammars First-Class. In *Proceedings of the 12th Workshop on Language Descriptions Tools and Applications*, Electronic Notes in Theoretical Computer Science, 2012.

[173] J. Visser. Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 270–282, New York, NY, USA, 2001. ACM. doi: 10.1145/504282.504302.

[174] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 131–145, New York, NY, USA, 1989. ACM. doi: 10.1145/73141.74830.

[175] J. Voigtländer. Conditions for Efficiency Improvement by Tree Transducer Composition. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 57–100. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45610-4˙16.

[176] J. Voigtländer. Formal Efficiency Analysis for Tree Transducer Composition. *Theory of Computing Systems*, 41(4):619–689, 2007. ISSN 1432-4350. doi: 10.1007/s00224-006-1235-9.

[177] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990. doi: 10.1016/0304-3975(90)90147-A.

[178] P. Wadler. The Expression Problem. Available on `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[179] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, University of Oxford, 1971.

[180] K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 1–13, Berkeley, CA, USA, 1997. USENIX Association.

[181] M. P. Ward. Language-Oriented Programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

[182] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(1):87–140, 2008. doi: 10.1017/S0956796807006557.

[183] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 333–345, New York, NY, USA, 2011. ACM. doi: 10.1145/2034773.2034818.

[184] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 241–252, New York, NY, USA, 2001. ACM. doi: 10.1145/507635.507665.

[185] M. Zenger and M. Odersky. Independently Extensible Solutions to the Expression Problem. In *Proceedings of he Twelfth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2005.

Appendix A

Papers on Modular Implementation of Programming Languages

Paper A1 P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, New York, NY, USA, 2011. ACM. doi: 10.1145/2036918.2036930

Paper A2 P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–24. Open Publishing Association, 2012. doi: 10.4204/EPTCS.76.3

Paper A3 P. Bahr. Modular Tree Automata. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 263–299. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-31113-0˙14

Paper A4 T. Hvitved, P. Bahr, and J. Andersen. Domain-Specific Languages for Enterprise Systems. Technical report, Department of Computer Science, University of Copenhagen, 2011

Compositional Data Types

Patrick Bahr Tom Hvitved

Department of Computer Science, University of Copenhagen

Abstract

Building on Wouter Swierstra's *Data types à la carte*, we present a comprehensive Haskell library of *compositional data types* suitable for practical applications. In this framework, data types and functions on them can be defined in a modular fashion. We extend the existing work by implementing a wide array of recursion schemes including monadic computations. Above all, we generalise recursive data types to *contexts*, which allow us to characterise a special yet frequent kind of catamorphisms. The thus established notion of *term homomorphisms* allows for flexible reuse and enables short-cut fusion style deforestation which yields considerable speedups. We demonstrate our framework in the setting of compiler construction, and moreover, we compare compositional data types with generic programming techniques and show that both are comparable in run-time performance and expressivity while our approach allows for stricter types. We substantiate this conclusion by lifting compositional data types to mutually recursive data types and generalised algebraic data types. Lastly, we compare the run-time performance of our techniques with traditional implementations over algebraic data types. The results are surprisingly good.

Contents

1 Introduction

Static typing provides a valuable tool for expressing invariants of a program. Yet, all too often, this tool is not leveraged to its full extent because it is simply not practical. Vice versa, if we want to use the full power of a type system, we often find ourselves writing large chunks of boilerplate code or—even worse—duplicating code. For example, consider the type of non-empty lists. Even though having such a type at your disposal is quite useful, you would rarely find it in use since—in a practical type system such as Haskell's—it would require the duplication of functions which work both on general and non-empty lists.

The situation illustrated above is an ubiquitous issue in compiler construction: In a compiler, an abstract syntax tree (AST) is produced from a source file, which then goes through different transformation and analysis phases, and is finally transformed into the target code. As functional programmers, we want to reflect the changes of each transformation step in the type of the AST. For example, consider the desugaring phase of a compiler which reduces syntactic sugar to the core syntax of the object language. To properly reflect this structural change also in the types, we have to create and maintain a variant of the data type defining the AST for the core syntax. Then, however, functions have to be defined for both types independently, i.e. code cannot be readily reused for both types! If we add annotations in an analysis step of the compiler, the type of the AST has to be changed again. But some functions should ignore certain annotations while being aware of others. And it gets even worse if we allow extensions to the object language that can be turned on and off independently, or if we want to implement several domain-specific languages which share a common core. This quickly becomes a nightmare with the choice of either duplicating lots of code or giving up static type safety by using a huge AST data type that covers all cases.

The essence of this problem can be summarised as the *Expression Problem*, i.e. "the goal [...] to define a datatype by cases, where one can add new cases

to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety" [24]. Wouter Swierstra [19] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits invaluable simplicity and clarity, it lacks abilities necessary to apply it in a practical setting beyond the confined simplicity of the expression problem.

The goal of this paper is to extend Swierstra's work in order to enhance its flexibility, improve its performance and broaden its scope of applications. In concrete terms, our contributions are:

- We implement recursion schemes other than catamorphisms (Section 4.5) and also account for recursion schemes over *monadic computations* (Section 3.2).
- We show how *generic programming* techniques can be efficiently implemented on top of the compositional data types framework (Section 3.1), providing a performance competitive with top-performing dedicated generic programming libraries.
- By generalising terms—i.e. recursive data types—to contexts—i.e. recursive data types with holes—we are able to capture the notion of term homomorphisms (Section 4.4), a special but common case of term algebras. In contrast to general algebras, term homomorphisms can easily be lifted to different data types, readily reused, and composed (also with algebras). The latter allows us to perform optimisations via short-cut fusion rules that provide considerable speedups (Section 6.2).
- We further extend the scope of applications by capturing compositional mutually recursive data types and GADTs via the construction of Johann and Ghani [6] (Section 5).
- Finally, we show the practical competitiveness of compositional data types by reducing their syntactic overhead using Template Haskell [17] (Section 6.1), and by comparing the run-time of typical functions with corresponding implementations over ordinary recursive data types (Section 6.2).

The framework of compositional data types that we present here is available from Hackage[1]. It contains the complete source code, numerous examples, and the benchmarks whose results we present in this paper. All code fragments presented throughout the paper are written in Haskell [9].

2 Data Types à la Carte

This section serves as an introduction to Swierstra's *data types à la carte* [19] (from here on, *compositional data types*), using our slightly revised notation and terminology. We demonstrate the application of compositional data types to a setting consisting of a family of expression languages that pairwise share some sublanguage, and operations that provide transformations between some of them.

[1] See `http://hackage.haskell.org/package/compdata`.

We illustrate the merits of this method on two examples: expression evaluation and desugaring.

2.1 Evaluating Expressions

Consider a simple language of expressions over integers and tuples, together with an evaluation function:

```
data Exp = Const Int | Mult Exp Exp
         | Pair Exp Exp | Fst Exp | Snd Exp
data Value = VConst Int | VPair Value Value
eval :: Exp → Value
eval (Const n)  = VConst n
eval (Mult x y) = let VConst m = eval x
                      VConst n = eval y
                  in  VConst (m * n)
eval (Pair x y) = VPair (eval x) (eval y)
eval (Fst x)    = let VPair v _ = eval x in v
eval (Snd x)    = let VPair _ v = eval x in v
```

In order to statically guarantee that the evaluation function produces values—a sublanguage of the expression language—we are forced to replicate parts of the expression structure in order to represent values. Consequently, we are also forced to duplicate common functionality such as pretty printing. Compositional data types provide a solution to this problem by relying on the well-known technique [11] of separating the recursive structure of terms from their signatures (functors). Recursive functions, in the form of catamorphisms, can then be specified by algebras on these signatures.

For our example, it suffices to define the following two signatures in order to separate values from general expressions:

```
data Val e = Const Int | Pair e e
data Op e  = Mult e e | Fst e | Snd e
```

The novelty of compositional data types then is to combine signatures—and algebras defined on them—in a modular fashion, by means of a formal sum of functors:

```
data (f :+: g) a = Inl (f a) | Inr (g a)
```

It is easy to show that $f :+: g$ is a functor whenever f and g are functors. We thus obtain the combined signature for expressions:

```
type Sig = Op :+: Val
```

Finally, the type of terms over a (potentially compound) signature f can be constructed as the fixed point of the signature f:

```
data Term f = Term {unTerm :: (f (Term f))}
```

We then have that $\mathit{Term}\ \mathit{Sig} \cong \mathit{Exp}$ and $\mathit{Term}\ \mathit{Val} \cong \mathit{Value}$.[2]

However, using compound signatures constructed by formal sums means that we have to explicitly tag constructors with the right injections. For instance, the term $1 * 2$ has to be written as

$$
\begin{aligned}
&e :: \mathit{Term}\ \mathit{Sig} \\
&e = \mathit{Term}\ \$\ \mathit{Inl}\ \$\ \mathit{Mult}\ (\mathit{Term}\ \$\ \mathit{Inr}\ \$\ \mathit{Const}\ 1)\ (\mathit{Term}\ \$\ \mathit{Inr}\ \$\ \mathit{Const}\ 2)
\end{aligned}
$$

Even worse, if we want to embed the term e into a type over an extended signature, say with syntactic sugar, then we have to add another level of injections *throughout* its definition. To overcome this problem, injections are derived using a type class:

$$
\begin{aligned}
&\textbf{class}\ \mathit{sub} :\prec \mathit{sup}\ \textbf{where} \\
&\quad \mathit{inj} :: \mathit{sub}\ a \to \mathit{sup}\ a \\
&\quad \mathit{proj} :: \mathit{sup}\ a \to \mathit{Maybe}\ (\mathit{sub}\ a)
\end{aligned}
$$

Using *overlapping instance* declarations, the sub-signature relation $:\prec$ can be constructively defined. However, due to restrictions of the type class system, we have to restrict ourselves to instances of the form $f :\prec g$ where f is atomic, i.e. not a sum, and g is a right-associative sum, e.g. $g_1 :+: (g_2 :+: g_3)$ but not $(g_1 :+: g_2) :+: g_3$.[3] Using the carefully defined instances for $:\prec$, we can then define injection and projection functions:

$$
\begin{aligned}
&\mathit{inject} :: (g :\prec f) \Rightarrow g\ (\mathit{Term}\ f) \to \mathit{Term}\ f \\
&\mathit{inject} = \mathit{Term}\ .\ \mathit{inj} \\
&\mathit{project} :: (g :\prec f) \Rightarrow \mathit{Term}\ f \to \mathit{Maybe}\ (g\ (\mathit{Term}\ f)) \\
&\mathit{project} = \mathit{proj}\ .\ \mathit{unTerm}
\end{aligned}
$$

Additionally, to reduce the syntactic overhead, we use smart constructors—which can be derived automatically, cf. Section 6.1—that already comprise the injection:

$$
\begin{aligned}
&\mathit{iMult} :: (\mathit{Op} :\prec f) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ f \to \mathit{Term}\ f \\
&\mathit{iMult}\ x\ y = \mathit{inject}\ \$\ \mathit{Mult}\ x\ y
\end{aligned}
$$

The term $1 * 2$ can now be written without syntactic overhead

$$
\begin{aligned}
&e :: \mathit{Term}\ \mathit{Sig} \\
&e = \mathit{iConst}\ 1\ `\mathit{iMult}`\ \mathit{iConst}\ 2
\end{aligned}
$$

and we can even give e the *open* type $(\mathit{Val} :\prec f, \mathit{Op} :\prec f) \Rightarrow \mathit{Term}\ f$. That is, e can be used as a term over any signature containing at least values and operators.

Next, we want to define the evaluation function, i.e. a function with the type $\mathit{Term}\ \mathit{Sig} \to \mathit{Term}\ \mathit{Val}$. To this end, we define the following *algebra class Eval*:

$$
\textbf{type}\ \mathit{Alg}\ f\ a = f\ a \to a
$$

[2]For clarity, we have omitted the strictness annotation to the constructor *Term* which is necessary in order to obtain the indicated isomorphisms.

[3]We encourage the reader to consult Swierstra's original paper [19] for the proper definition of the $:\prec$ relation.

```
class Eval f v where evalAlg :: Alg f (Term v)
instance (Eval f v, Eval g v) ⇒ Eval (f :+: g) v where
  evalAlg (Inl x) =  evalAlg x
  evalAlg (Inr x) =  evalAlg x
```

The instance declaration for sums is crucial, as it defines how to combine instances for the different signatures—yet the structure of its declaration is independent from the particular algebra class, and it can be automatically derived for any algebra. Thus, we will omit the instance declarations lifting algebras to sums from now on. The actual evaluation function can then be obtained from instances of this algebra class as a *catamorphism*. In order to perform the necessary recursion, we require the signature f to be an instance of *Functor* providing the method $fmap :: (a \to b) \to f\ a \to f\ b$:

```
cata :: Functor f ⇒ Alg f a → Term f → a
cata f = f . fmap (cata f) . unTerm
eval :: (Functor f, Eval f v) ⇒ Term f → Term v
eval = cata evalAlg
```

What remains is to define the algebra instances for *Val* and *Op*. One approach is to define instances *Eval Val Val* and *Eval Op Val*, however such definitions are problematic if we later want to add a signature to the language which also extends the signature for values, say with Boolean values. We could hope to achieve such extendability by defining the instance

```
instance (Eval f v, v :≺: v') ⇒ Eval f v'
```

but this is problematic for two reasons: First, the relation $:\prec:$ only works for atomic left-hand sides, and second, we can in fact not define this instance because the function $evalAlg :: f\ (Term\ v) \to Term\ v$ cannot be lifted to the type $f\ (Term\ v') \to Term\ v'$, as the type of the domain also changes. Instead, the correct approach is to leave the instance declarations open in the target signature:

```
instance (Val :≺: v) ⇒ Eval Val v where
  evalAlg = inject
instance (Val :≺: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst $ projC x * projC y
  evalAlg (Fst x)    = fst $ projP x
  evalAlg (Snd x)    = snd $ projP x
projC :: (Val :≺: v) ⇒ Term v → Int
projC v = case project v of Just (Const n) → n
projP :: (Val :≺: v) ⇒ Term v → (Term v, Term v)
projP v = case project v of Just (Pair x y) → (x, y)
```

Notice how the constructors *Const* and *Pair* are treated with a single *inject*, as these are already part of the value signature.

2.2 Adding Sugar on Top

We now consider an extension of the expression language with *syntactic sugar*, exemplified via negation and swapping of pairs:

```
data Sug e = Neg e | Swap e
type Sig'  = Sug :+: Sig
```

Defining a desugaring function $Term\ Sig' \to Term\ Sig$ then amounts to instantiating the following algebra class:

```
class Desug f g where
  desugAlg :: Alg f (Term g)
desug :: (Functor f, Desug f g) ⇒ Term f → Term g
desug = cata desugAlg
```

Using overlapping instances, we can define a default translation for *Val* and *Op*, so we only have to write the "interesting" cases:

```
instance (f :≺: g) ⇒ Desug f g where
  desugAlg = inject
instance (Val :≺: f, Op :≺: f) ⇒ Desug Sug f where
  desugAlg (Neg x)  = iConst (−1) `iMult` x
  desugAlg (Swap x) = iSnd x `iPair` iFst x
```

Note how the context of the last instance reveals that desugaring of the extended syntax requires a target signature with at least base values, $Val :\prec: f$, and operators, $Op :\prec: f$. By composing *desug* and *eval*, we get an evaluation function for the extended language:

```
eval' :: Term Sig' → Term Val
eval' = eval . (desug :: Term Sig' → Term Sig)
```

The definition above shows that there is a small price to pay for leaving the algebra instances open: We have to annotate the desugaring function in order to pin down the intermediate signature *Sig*.

3 Extensions

In this section, we introduce some rather straightforward extensions to the compositional data types framework: Generic programming combinators, monadic computations, and annotations.

3.1 Generic Programming

Most of the functions that are definable in the common generic programming frameworks [14] can be categorised as either query functions $d \to r$, which analyse a data structure of type d by extracting some relevant information of type r from parts of the input and compose them, or as transformation functions $d \to d$,

which recursively apply some type preserving functions to parts of the input. The benefit that generic programming frameworks offer is that programmers only need to specify the "interesting" parts of the computation. We will show how we can easily reproduce this experience on top of compositional data types.

Applying a type-preserving function recursively throughout a term can be implemented easily. The function below applies a given function in a bottom-up manner:

```
trans :: Functor f ⇒ (Term f → Term f) → (Term f → Term f)
trans f = cata (f . Term)
```

Other recursion schemes can be implemented just as easily.

In order to implement generic querying functions, we need a means to combine the result of querying a functorial value. The standard type class *Foldable* generalises folds over lists and thus provides us with exactly the interface we need:[4]

```
class Foldable f where
  foldl :: (a → b → a) → a → f b → a
```

For example, an appropriate instance for the functor *Val* can be defined like this:

```
instance Foldable Val where
  foldl _ a (Const _) = a
  foldl f a (Pair x y) = (a `f` x) `f` y
```

With *Foldable*, a generic querying function can be implemented easily. It takes a function $q :: Term\ f \to r$ to query a single node of the term and a function $c :: r \to r \to r$ to combine two results:

```
query :: Foldable f ⇒ (Term f → r) → (r → r → r) → Term f → r
query q c t =
  foldl (λr x → r `c` query q c x) (q t) (unTerm t)
```

We can instantiate this scheme, for example, to implement a generic size function:

```
gsize :: Foldable f ⇒ Term f → Int
gsize = query (const 1) (+)
```

A very convenient scheme of query functions introduced by Mitchell and Runciman [12], in the form of the *universe* combinator, simply returns a list of all subterms. Specific queries can then be written rather succinctly using list comprehensions. Such a combinator can be implemented easily via *query*:

```
subs :: Foldable f ⇒ Term f → [Term f]
subs = query (λx → [x]) (++)
```

[4] *Foldable* also has other fold functions, but they are derivable from *foldl* and are not relevant for our purposes.

However, in order to make the pattern matching in list comprehensions work, we need to project the terms to the functor that contains the constructor we want to match against:

$$subs' :: (Foldable\ f, g :\prec f) \Rightarrow Term\ f \to [g\ (Term\ f)]$$
$$subs' = mapMaybe\ project\ .\ subs$$

With this in place we can for example easily sum up all integer literals in an expression:

$$sumInts :: (Val :\prec f) \Rightarrow Term\ f \to Int$$
$$sumInts\ t = sum\ [i \mid Const\ i \leftarrow subs'\ t]$$

This shows that we can obtain functionality similar to what dedicated generic programming frameworks offer. In contrast to generic programming, however, the compositional data type approach provides additional tools that allow us to define functions with a stricter type that reflects the underlying transformation. For example, we could have defined the desugaring function in terms of *trans*, but that would have resulted in the "weaker" type $Term\ Sig' \to Term\ Sig'$ instead of $Term\ Sig' \to Term\ Sig$. The latter type witnesses that indeed all syntactic sugar is removed!

Nevertheless, the examples show that at least the querying combinators *query* and *subs′* provide an added value to our framework. Moreover, by applying standard optimisation techniques we can obtain run-time performance comparable with top-performing generic programming libraries (cf. Section 6.2). In contrast to common generic programming libraries [14], we only considered combinators that work on a single recursive data type. This restriction is lifted in Section 5 when we move to mutually recursive data types.

3.2 Monadic Computations

We saw in Section 2 how to realise a modular evaluation function for a small expression language in terms of catamorphisms defined by algebras. In order to deal with type mismatches, we employed non-exhaustive case expressions. Clearly, it would be better to use a monad instead. However, a monadic carrier type $m\ a$ would yield an algebra $f\ (m\ a) \to m\ a$ which means that we have to explicitly sequence the nested monadic values of the argument. What we would rather like to do is to write a *monadic algebra* [3]

$$\textbf{type}\ AlgM\ m\ f\ a = f\ a \to m\ a$$

where the nested sequencing is done automatically and thus the monadic type only occurs in the codomain. Again we are looking for a function that we already know from lists:

$$sequence :: Monad\ m \Rightarrow [m\ a] \to m\ [a]$$

The standard type class *Traversable* [10] provides the appropriate generalisation to functors:

```
class (Functor f, Foldable f) ⇒ Traversable f where
  sequence :: Monad m ⇒ f (m a) → m (f a)
  mapM     :: Monad m ⇒ (a → m b) → f a → m (f b)
```

Here, *mapM* is simply the composition of *sequence* and *fmap*.

The definition of a monadic variant of catamorphisms can then be derived by replacing *fmap* with *mapM* and function composition with monadic function composition ⟸:

```
cataM :: (Traversable f, Monad m) ⇒ AlgM m f a → Term f → m a
cataM f = f ⟸ mapM (cataM f) . unTerm
```

The following definitions illustrate how monadic catamorphisms can be used to define a *safe* version of the evaluation function from Section 2, which properly handles errors when applied to a *bad term* (using the *Maybe* monad for simplicity):

```
class EvalM f v where
  evalAlgM :: AlgM Maybe f (Term v)
evalM :: (Traversable f, EvalM f v) ⇒ Term f → Maybe (Term v)
evalM = cataM evalAlgM
instance (Val :≺: v) ⇒ EvalM Val v where
  evalAlgM = return . inject
instance (Val :≺: v) ⇒ EvalM Op v where
  evalAlgM (Mult x y) = liftM iConst $ liftM2 (∗) (projCM x) (projCM y)
  evalAlgM (Fst x)    = liftM fst $ projPM x
  evalAlgM (Snd x)    = liftM snd $ projPM x
projCM :: (Val :≺: v) ⇒ Term v → Maybe Int
projCM v = case project v of
  Just (Const n) → return n
  _              → Nothing
projPM :: (Val :≺: v) ⇒ Term v → Maybe (Term v, Term v)
projPM v = case project v of
  Just (Pair x y) → return (x, y)
  _               → Nothing
```

3.3 Products and Annotations

We have seen in Section 2 how the sum :+: can be used to combine signatures. This inevitably leads to the dual construction:

```
data (f :∗: g) a = f a :∗: g a
```

In its general form, the product :∗: seems of little use: Each constructor of f can be paired with each constructor of g. The special case, however, where g is a constant functor, is easy to comprehend yet immensely useful:

```
data (f :&: c) a = f a :&: c
```

Now, every value of type $(f :\&: c)\ a$ is value from $f\ a$ annotated with a value in c. On the term level, this means that a term over $f :\&: c$ is a term over f in which each subterm is annotated with a value in c.

This addresses a common problem in compiler implementations: How to deal with annotations of AST nodes such as source positions or type information which have only a limited lifespan or are only of interest for some parts of the compiler?

Given the signature *Sig* for our simple expression language and a type *Pos* which represents source position information such as a file name and a line number, we can represent ASTs with source position annotations as $Term\ (Sig :\&: Pos)$ and write a parser that provides such annotations [21].

The resulting representation yields a clean separation between the actual data—the AST—and the annotation data—the source positions—which is purely supplemental for supplying better error messages. The separation allows us to write a generic function that strips off annotations when they are not needed:

$$
\begin{aligned}
&remA :: (f :\&: p)\ a \to f\ a\\
&remA\ (v :\&: \_) = v\\
&stripA :: Functor\ f \Rightarrow Term\ (f :\&: p) \to Term\ f\\
&stripA = cata\ (Term\ .\ remA)
\end{aligned}
$$

With this in place, we can provide a generic combinator that lifts a function on terms to a function on terms with annotations

$$
\begin{aligned}
&liftA :: Functor\ f \Rightarrow (Term\ f \to t) \to Term\ (f :\&: p) \to t\\
&liftA\ f = f\ .\ stripA
\end{aligned}
$$

which works for instance for the evaluation function:

$$liftA\ eval :: Term\ (Sig :\&: Pos) \to Term\ Val$$

But how do we actually define an algebra that uses the position annotations? We are faced with the problem that the product :&: is applied to a *sum*, viz. $Sig = Op :+: Val$. When defining the algebra for one of the summands, say *Val*, we do not have immediate access to the factor *Pos* which is outside of the sum.

We can solve this issue in two ways: (a) Propagating the annotation using a *Reader* monad or (b) providing operations that allow us to make use of the right-distributivity of :&: over :+:. For the first approach, we only need to move from algebras $Alg\ f\ a$ to monadic algebras $AlgM\ (Reader\ p)\ f\ a$, for p the type of the annotations. Given an algebra class, e.g. for type inference

$$
\begin{aligned}
&\textbf{class}\ Infer\ f\ \textbf{where}\\
&\quad inferAlg :: AlgM\ (Reader\ Pos)\ f\ Type
\end{aligned}
$$

we can lift it to annotated signatures:[5]

$$
\begin{aligned}
&\textbf{instance}\ Infer\ f \Rightarrow Infer\ (f :\&: Pos)\ \textbf{where}\\
&\quad inferAlg\ (v :\&: p) = local\ (const\ p)\ (inferAlg\ v)
\end{aligned}
$$

[5]The standard function $local :: (r \to r) \to Reader\ r\ a \to Reader\ r\ a$ updates the environment by the function given as first argument.

When defining the other instances of the class, we can use the monadic function $ask :: Reader\ Pos\ Pos$ to query the annotation of the current subterm. This provides a clean interface to the annotations. It requires, however, that we define a monadic algebra.

The alternative approach is to distribute the annotations over the sum, i.e. instead of $Sig :\&: Pos$ we use the type

$$\textbf{type}\ SigP = Op :\&: Pos :+: Val :\&: Pos$$

Now, we are able to define a direct instance of the form

$$\begin{aligned}&\textbf{instance}\ Infer\ (Val :\&: Pos)\ \textbf{where}\\ &\quad inferAlg\ (v :\&: p) = ...\end{aligned}$$

where we have direct access to the position annotation p. However, now we have the dual problem: We do not have immediate access to the annotation at the outermost level of the sum. Hence, we cannot use the function *liftA* to lift functions to annotated terms. Yet, this direction—propagating annotations outwards—is easier to deal with. We have to generalise the function *remA* to also deal with annotations distributed over sums. This is an easy exercise:

$$\begin{aligned}&\textbf{class}\ RemA\ f\ g \mid f \to g\ \textbf{where}\\ &\quad remA :: f\ a \to g\ a\\ &\textbf{instance}\ RemA\ (f :\&: p)\ f\ \textbf{where}\\ &\quad remA\ (v :\&: \_) = v\\ &\textbf{instance}\ RemA\ f\ f' \Rightarrow RemA\ (g :\&: p :+: f)\ (g :+: f')\ \textbf{where}\\ &\quad remA\ (Inl\ (v :\&: \_)) = Inl\ v\\ &\quad remA\ (Inr\ v) = Inr\ (remA\ v)\end{aligned}$$

Now the function *remA* works as before, but it can also deal with signatures such as *SigP*, and the type of *liftA* becomes:

$$(Functor\ f, RemA\ f\ g) \Rightarrow (Term\ g \to t) \to Term\ f \to t$$

Both approaches have their share of benefits and drawbacks. The monadic approach provides a cleaner interface but necessitates a monadic style. The explicit distribution is more flexible as it both allows us to access the annotations directly by pattern matching or to thread them through a monad if that is more convenient. On the other hand, it means that adding annotations is not straightforwardly compositional anymore. The annotation $:\&: A$ has to be added to each summand—just like compound signatures are not straightforwardly compositional, e.g. we have to write the sum $f :+: g$, for a signature $f = f_1 :+: f_2$, explicitly as $f_1 :+: f_2 :+: g$.

4 Context Matters

In this section, we will discuss two problems that arise when defining term algebras, i.e. algebras with a carrier of the form $Term\ f$. These problems occur when we want to lift term algebras to algebras on annotated terms, and when trying

to compose term algebras. We will show how these problems can be addressed by *term homomorphisms*, a quite common special case of term algebras. In order to make this work, we shall generalise terms to contexts by using generalised algebraic data types (GADTs) [16].

4.1 Propagating Annotations

As we have seen in Section 3.3, it is easy to lift functions on terms to functions on annotated terms. It only amounts to removing all annotations before passing the term to the original function.

But what if we do not want to completely ignore the annotation but propagate it in a meaningful way to the output? Take for example the desugaring function *desug* we have defined in Section 2 and which transforms terms over Sig' to terms over Sig. How do we lift this function easily to a function of type

$$Term\ (Sig' :\&: Pos) \to Term\ (Sig :\&: Pos)$$

which propagates the annotations such that each annotation of a subterm in the result is taken from the subterm it originated? For example, in the desugaring of a term $iSwap\ x$ to the term $iSnd\ x\ `iPair`\ iFst\ x$, the top-most $Pair$-term, as well as the two terms $Snd\ x$ and $Fst\ x$ should get the same annotation as the original subterm $iSwap\ x$.

This propagation is independent of the transformation function. The same scheme can also be used for the type inference function in order to annotate the inferred type terms with the positions of the code that is responsible for each part of the type terms.

It is clear that we will not be able provide a combinator of type

$$(Term\ f \to Term\ g) \to Term\ (f :\&: p) \to Term\ (g :\&: p)$$

that lifts any function to one that propagates annotations meaningfully. We cannot tell from a plain function of type $Term\ f \to Term\ g$ where the subterms of the result term are originated in the input term. However, restricting ourselves to term algebras will not be sufficient either. That is, also a combinator of type

$$Alg\ f\ (Term\ g) \to Alg\ (f :\&: p)\ (Term\ (g :\&: p))$$

is out of reach. While we can tell from a term algebra, i.e. a function of type $f\ (Term\ g) \to Term\ g$, that some initial parts of the result term originate from the f-constructor at the root of the input, we do not know which parts. The term algebra only returns a *uniform* term of type $Term\ g$ which provides no information as to which parts were constructed from the f-part of the $f\ (Term\ g)$ argument and which were copied from the $(Term\ g)$-part.

Term algebras are still too general! We need to move to a function type that clearly states which parts are constructed from the "current" top-level symbol in f and which are copied from its arguments in $Term\ g$. In order to express that certain parts are just copied, we can make use of parametric polymorphism.

Instead of an algebra, we can define a function on terms also by a *natural transformation*, a function of type $\forall\ a\ .\ f\ a \to g\ a$. Such a function can only

transform an f-constructor into a g-constructor and copy its arguments around. Since the copying is made explicit in the type, defining a function that propagates annotations through natural transformations is straightforward:

$$\begin{aligned}&prop :: (f\ a \to g\ a) \to (f :\&: p)\ a \to (g :\&: p)\ a\\&prop\ f\ (v :\&: p) = f\ v :\&: p\end{aligned}$$

Unfortunately, natural transformations are also quite limited. They only allow us to transform each constructor of the original term to exactly one constructor in the target term. This is for example not sufficient for the desugaring function, which translates a constructor application $iSwap\ x$ into three constructor applications $iSnd\ x\ `iPair`\ iFst\ x$. In order to lift this restriction, we need to be able to define a function of type $\forall\ a\ .\ f\ a \to Context\ g\ a$ which transforms an f-constructor application to a *g-context* application, i.e. several nested applications of g-constructors potentially with some "holes" filled by values of type a.

We shall return to this idea in Section 4.4.

4.2 Composing Term Algebras

The benefit of having a desugaring function $desug :: Term\ Sig' \to Term\ Sig$, which is able to reduce terms over the richer signature Sig' to terms over the core signature Sig, is that it allows us to easily lift functions that are defined on terms over Sig—such as evaluation and type inference—to terms over Sig':

$$\begin{aligned}&eval' :: Term\ Sig' \to Term\ Val\\&eval' = eval\ .\ (desug :: Term\ Sig' \to Term\ Sig)\end{aligned}$$

However, looking at how $eval$ and $desug$ are defined, viz. as catamorphisms, we notice a familiar pattern:

$$eval' = cata\ evalAlg\ .\ cata\ desugAlg$$

This looks quite similar to the classic example of *short-cut fusion*:

$$map\ f\ .\ map\ g \quad \rightsquigarrow \quad map\ (f\ .\ g)$$

An expression that traverses a data structure twice is transformed into one that only does this once.

To replicate this on terms, we need an appropriately defined composition operator $\circledcirc$ on term algebras that allows us to perform a similar semantics-preserving transformation:

$$cata\ f\ .\ cata\ g \quad \rightsquigarrow \quad cata\ (f \circledcirc g)$$

As a result, the input term only needs to be traversed once instead of twice and the composition and decomposition of an intermediate term is avoided. The type of $\circledcirc$ should be

$$Alg\ g\ (Term\ h) \to Alg\ f\ (Term\ g) \to Alg\ f\ (Term\ h)$$

Since term algebras are functions, the only way to compose them is by first making them compatible and then performing function composition. Given two

term algebras $a :: Alg\ g\ (Term\ h)$ and $b :: Alg\ f\ (Term\ g)$, we can turn them into compatible functions by lifting a to terms via *cata*. The problem now is that the composition $cata\ a\ .\ b$ has type $f\ (Term\ g) \to Term\ h$, which is only an algebra if $g = h$. This issue arises due to the simple fact that the carrier of an algebra occurs in both the domain and the codomain of the function! Instead of a term algebra of type $f\ (Term\ g) \to Term\ g$, we need a function type in which the domain is more independent from the codomain in order to allow composition. Again, a type of the form $\forall\ a\ .\ f\ a \to Context\ g\ a$ provides a solution.

4.3 From Terms to Contexts and back

We have seen in the two preceding sections that we need an appropriate notion of *contexts*, i.e. a term which can also contain "holes" filled with values of a certain type. Starting from the definition of terms, we can easily generalise it to contexts by simply adding an additional case:

$$\begin{aligned}&\textbf{data}\ Context\ f\ a = Context\ (f\ (Context\ f\ a))\\ &\qquad\qquad\qquad\quad \mid\ Hole\ a\end{aligned}$$

Note that we can obtain a type isomorphic to the one above using summation: $Context\ f\ a \cong Term\ (f :+: K\ a)$ for a type

$$\textbf{data}\ K\ a\ b = K\ a$$

Since we will use contexts quite often, we will use the direct representation. Moreover, this allows us to tightly integrate contexts into our framework. Since contexts are terms with holes, we also want to go the other way around by defining terms as contexts *without holes*! This will allow us to lift functions defined on terms—catamorphisms, injections etc.—to functions on contexts that provide the original term-valued function as a special case.

The idea of defining terms as contexts without holes can be encoded in Haskell quite easily as a *generalised algebraic data type (GADT)* [16] with a *phantom type Hole*:

$$\begin{aligned}&\textbf{data}\ Cxt :: * \to (* \to *) \to * \to *\ \ \textbf{where}\\ &\quad Term :: f\ (Cxt\ h\ f\ a) \to Cxt\ h\ f\ a\\ &\quad Hole :: a \to Cxt\ Hole\ f\ a\\ &\textbf{data}\ Hole\end{aligned}$$

In this representation, we add an additional type argument that indicates whether the context might contain holes or not. A context that does have a hole must have a type of the form $Cxt\ Hole\ f\ a$. Our initial definition of contexts can be recovered by defining:

$$\textbf{type}\ Context = Cxt\ Hole$$

That is, contexts *may* contain holes. On the other hand, terms must not contain holes. This can be defined by:

$$\textbf{type}\ Term\ f = \forall\ h\ a\ .\ Cxt\ h\ f\ a$$

While this is a natural representation of terms as a special case of the more general concept of contexts, this usually causes some difficulties because of the impredicative polymorphism. We therefore prefer an approximation of this type that will do fine in almost any relevant case. Instead of universal quantification, we use empty data types *NoHole* and *Nothing*:

$$\textbf{type}\ \mathit{Term}\ f = \mathit{Cxt}\ \mathit{NoHole}\ f\ \mathit{Nothing}$$

In practice, this does not pose any restriction whatsoever. Both *NoHole* and *Nothing* are phantom types and do not contribute to the internal representation of values. For the former this is obvious, for the latter this follows from the fact that the phantom type *NoHole* witnesses that the context has indeed no holes which would otherwise enforce the type *Nothing*. Hence, we can transform a term to any context type over any type of holes:

$$\begin{aligned}&\mathit{toCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Term}\ f \to \forall\ h\ a\ .\ \mathit{Cxt}\ h\ f\ a\\&\mathit{toCxt}\ (\mathit{Term}\ t) = \mathit{Term}\ (\mathit{fmap}\ \mathit{toCxt}\ t)\end{aligned}$$

In fact, *toCxt* does not change the representation of the input term. Looking at its definition, *toCxt* is operationally equivalent to the identity. Thus, we can safely use the function $\mathit{unsafeCoerce} :: a \to b$ in order to avoid run-time overhead:

$$\begin{aligned}&\mathit{toCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Term}\ f \to \forall\ h\ a\ .\ \mathit{Cxt}\ h\ f\ a\\&\mathit{toCxt} = \mathit{unsafeCoerce}\end{aligned}$$

This representation of contexts and terms allows us to uniformly define functions which work on both types. The function *inject* can be defined as before, but now has the type

$$\mathit{inject} :: (g :\prec: f) \Rightarrow g\ (\mathit{Cxt}\ h\ f\ a) \to \mathit{Cxt}\ h\ f\ a$$

and thus works for both terms and proper contexts. The projection function has to be extended slightly to accommodate for holes:

$$\begin{aligned}&\mathit{project} :: (g :\prec: f) \Rightarrow \mathit{Cxt}\ h\ f\ a \to \mathit{Maybe}\ (g\ (\mathit{Cxt}\ h\ f\ a))\\&\mathit{project}\ (\mathit{Term}\ t) = \mathit{proj}\ t\\&\mathit{project}\ (\mathit{Hole}\ \_) = \mathit{Nothing}\end{aligned}$$

The relation between terms and contexts can also be illustrated algebraically: If we ignore for a moment the ability to define infinite terms due to Haskell's non-strict semantics, the type *Term F* represents the initial $\mathcal{F}$-algebra which has the carrier $\mathcal{T}(\mathcal{F})$, the terms over signature $\mathcal{F}$. The type of contexts *Context F X* on the other hand represents the free $\mathcal{F}$-algebra generated by $\mathcal{X}$ which has the carrier $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the terms over signature $\mathcal{F}$ and variables $\mathcal{X}$.

Thus, for recursion schemes, we can move naturally from catamorphisms, i.e. initial algebra semantics, to free algebra semantics:

$$\begin{aligned}&\mathit{free} :: \mathit{Functor}\ f \Rightarrow \mathit{Alg}\ f\ b \to (a \to b) \to \mathit{Cxt}\ h\ f\ a \to b\\&\mathit{free}\ \mathit{alg}\ \ v\ (\mathit{Term}\ t) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{free}\ \mathit{alg}\ v)\ t)\\&\mathit{free}\ \_\quad v\ (\mathit{Hole}\ x) = v\ x\end{aligned}$$

$$
\begin{aligned}
&\mathit{freeM} :: (\mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \\
&\qquad \mathit{AlgM}\ m\ f\ b \to (a \to m\ b) \to \mathit{Cxt}\ h\ f\ a \to m\ b \\
&\mathit{freeM}\ \mathit{alg}\ v\ (\mathit{Term}\ t) = \mathit{alg} =\!\!\ll \mathit{mapM}\ (\mathit{freeM}\ \mathit{alg}\ v)\ t \\
&\mathit{freeM}\ \_\ \ v\ (\mathit{Hole}\ x)\ = v\ x
\end{aligned}
$$

This yields the central function for working with contexts:

$$
\begin{aligned}
&\mathit{appCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Context}\ f\ (\mathit{Cxt}\ h\ f\ a) \to \mathit{Cxt}\ h\ f\ a \\
&\mathit{appCxt} = \mathit{free}\ \mathit{Term}\ \mathit{id}
\end{aligned}
$$

This function takes a context whose holes are terms (or contexts) and returns the term (respectively context) that is obtained by merging the two—essentially by removing each constructor *Hole*. Notice how the type variables h and a are propagated from the input context's holes to the return type. In this way, we can uniformly treat both terms and contexts.

4.4 Term Homomorphisms

The examples from Sections 4.1 and 4.2 have illustrated the need for defining functions on terms by functions of the form $\forall\ a\ .\ f\ \ a \to \mathit{Context}\ g\ \ a$. Such functions can then be transformed to term algebras via *appCxt* and, thus, be lifted to terms:

$$
\begin{aligned}
&\mathit{termHom}\ \ ::\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \\
&\qquad \Rightarrow (\forall\ a\ .\ f\ \ a \to \mathit{Context}\ g\ \ a) \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{termHom}\ f = \mathit{cata}\ (\mathit{appCxt}\ .\ f)
\end{aligned}
$$

In fact, the polymorphism in the type $\forall\ a\ .\ f\ \ a \to \mathit{Context}\ g\ \ a$ guarantees that arguments of the functor f can only be copied—not inspected or modified. This restriction captures a well-known concept from tree automata theory:

Definition 1 (term homomorphisms[6] [2, 20])**.** Let $\mathcal{F}$ and $\mathcal{G}$ be two sets of function symbols, possibly not disjoint. For each $n > 0$, let $\mathcal{X}_n = \{x_1, \ldots, x_n\}$ be a set of variables disjoint from $\mathcal{F}$ and $\mathcal{G}$. Let $h_{\mathcal{F}}$ be a mapping which, with $f \in \mathcal{F}$ of arity n, associates a context $t_f \in \mathcal{T}(\mathcal{G}, \mathcal{X}_n)$. The *term homomorphism* $h\colon \mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{G})$ determined by $h_{\mathcal{F}}$ is defined as follows:

$$h(f(t_1, \ldots, t_n)) = t_f\,\{x_1 \mapsto h(t_1), \ldots, x_n \mapsto h(t_n)\}$$

The term homomorphism h is called *symbol-to-symbol* if, for each $f \in \mathcal{F}$, $t_f = g(y_1, \ldots, y_m)$ with $g \in \mathcal{G}, y_1, \ldots, y_m \in \mathcal{X}_n$, i.e. each t_f is a context of height 1. It is called *ε-free* if, for each $f \in \mathcal{F}$, $t_f \notin \mathcal{X}_n$, i.e. each t_f is a context of height at least 1.

Applying the *placeholders-via-naturality* principle of Hasuo et al. [5], term homomorphisms are captured by the following type:

$$\textbf{type}\ \mathit{TermHom}\ f\ g = \forall\ a\ .\ f\ \ a \to \mathit{Context}\ g\ \ a$$

[6] Actually, Thatcher [20] calls them "tree homomorphisms". But we prefer the notion "term" over "tree" in our context.

As we did for other functions on terms, we can generalise the application of term homomorphism uniformly to contexts:

$$
\begin{aligned}
&\mathit{termHom} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \\
&\qquad \Rightarrow \mathit{TermHom}\ f\ g \to \mathit{Cxt}\ h\ f\ a \to \mathit{Cxt}\ h\ g\ a \\
&\mathit{termHom}\ f\ (\mathit{Term}\ t) = \mathit{appCxt}\ (f\ (\mathit{fmap}\ (\mathit{termHom}\ f)\ t)) \\
&\mathit{termHom}\ \_\ (\mathit{Hole}\ b) = \mathit{Hole}\ b
\end{aligned}
$$

The use of explicit pattern matching in lieu of defining the function as a free algebra homomorphism $\mathit{free}\ (\mathit{appCxt} \,.\, f)\ \mathit{Hole}$ is essential in order to obtain this general type. In particular, the use of the proper GADT constructor *Hole*, which has result type $\mathit{Context}\ g\ a$, makes this necessary.

Of course, the polymorphic type of term homomorphisms restricts the class of functions that can be defined in this way. It can be considered as a special form of term algebra: $\mathit{appCxt} \,.\, f$ is the term algebra corresponding to the term homomorphism f. But not every catamorphism is also a term homomorphism. For certain term algebras we actually need to inspect the arguments of the functor instead of only shuffling them around. For example, we cannot hope to define the evaluation function *eval* as a term homomorphism.

Some catamorphisms, however, can be represented as term homomorphisms, e.g. the desugaring function *desug*:

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
&\quad \mathit{desugHom} :: \mathit{TermHom}\ f\ g
\end{aligned}
$$

Lifting term homomorphisms to sums is standard. The instances for the functors that do not need to be desugared can be implemented by turning a single functor application to a context of height 1, and using overlapping instances:

$$
\begin{aligned}
&\mathit{simpCxt} :: \mathit{Functor}\ f \Rightarrow f\ a \to \mathit{Context}\ f\ a \\
&\mathit{simpCxt} = \mathit{Term} \,.\, \mathit{fmap}\ \mathit{Hole} \\
&\textbf{instance}\ (f :\prec: g, \mathit{Functor}\ g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
&\quad \mathit{desugHom} = \mathit{simpCxt} \,.\, \mathit{inj}
\end{aligned}
$$

Turning to the instance for *Sug*, we can see why a term homomorphism suffices for implementing *desug*. In the original catamorphic definition, we had for example

$$\mathit{desugAlg}\ (\mathit{Neg}\ x) = \mathit{iConst}\ (-1)\ `\mathit{iMult}`\ x$$

Here we only need to copy the argument x of the constructor *Neg* and define the appropriate context around it. This definition can be copied almost verbatim for the term homomorphism:

$$\mathit{desugHom}\ (\mathit{Neg}\ x) = \mathit{iConst}\ (-1)\ `\mathit{iMult}`\ \mathit{Hole}\ x$$

We only need to embed the x as a hole. The same also applies to the other defining equation. In order to make the definitions more readable we add a convenience function to the class *Desug*:

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
&\quad \mathit{desugHom} :: \mathit{TermHom}\ f\ g
\end{aligned}
$$

```
desugHom = desugHom' . fmap Hole
desugHom' :: Alg f (Context g a)
desugHom' x = appCxt (desugHom x)
```

Now we can actually copy the catamorphic definition one-to-one:

```
instance (Op :≺: f, Val :≺: f, Functor f) ⇒ Desug Sug f where
  desugHom' (Neg x)  = iConst (−1) `iMult` x
  desugHom' (Swap x) = iSnd x `iPair` iFst x
```

In the next two sections, we will show what we actually gain by adopting the term homomorphism approach. We will reconsider and address the issues that we identified in Sections 4.1 and 4.2.

4.4.1 Propagating Annotations through Term Homomorphisms

The goal is now to take advantage of the structure of term homomorphisms in order to automatically propagate annotations. This boils down to transforming a function of type *TermHom f g* to a function of type *TermHom* (*f* :&: *p*) (*g* :&: *p*). In order to do this, we need a function that is able to annotate a context with a fixed annotation. Such a function is in fact itself a term homomorphism:

```
ann :: Functor f ⇒ p → Cxt h f a → Cxt h (f :&: p) a
ann p = termHom (simpCxt . (:&: p))
```

To be more precise, this function is a *symbol-to-symbol* term homomorphism—(:&:*p*) is of type ∀ *a* . *f a* → (*f* :&: *p*) *a*—that maps each constructor to exactly one constructor. The composition with *simpCxt* lifts it to the type of general term homomorphisms.

The propagation of annotations is now simple:

```
propAnn :: Functor g ⇒ TermHom f g → TermHom (f :&: p) (g :&: p)
propAnn f (t :&: p) = ann p (f t)
```

The annotation of the current subterm is propagated to the context created by the original term homomorphism.

This definition can now be generalised—as we did in Section 3.3—such that it can also deal with annotations that have been distributed over a sum of signatures. Unfortunately, the type class *RemA* that we introduced for dealing with such distributed annotations is not enough for this setting as we need to extract and inject annotations now:

```
class DistAnn f p f' | f' → f, f' → p where
  injectA  :: p → f  a →  f' a
  projectA ::      f' a → (f  a, p)
```

An instance of *DistAnn f p f'* indicates that signature *f'* is a variant of *f* annotated with values of type *p*. The relevant instances are straightforward:

```
instance DistAnn f p (f :&: p) where
  injectA c v = v :&: c
```

```
  projectA (v :&: p) = (v, p)
instance DistAnn f p f' ⇒ DistAnn (g :+: f) p ((g :&: p) :+: f') where
  injectA c (Inl v) = Inl (v :&: c)
  injectA c (Inr v) = Inr (injectA c v)
  projectA (Inl (v :&: p)) = (Inl v, p)
  projectA (Inr v)         = let (v', p) = projectA v
                             in (Inr v', p)
```

We can then make use of this infrastructure in the definition of *ann* and *propAnn*:

```
ann :: (DistAnn f p g, Functor f, Functor g)
    ⇒ p → Cxt h f a → Cxt h g a
ann p = termHom (simpCxt . injectA p)
propAnn :: (DistAnn f p f', DistAnn g p g', Functor g, Functor g')
        ⇒ TermHom f g → TermHom f' g'
propAnn f t' = let (t, p) = projectA t' in ann p (f t)
```

We can now use *propAnn* to propagate source position information from a full AST to its desugared version:

```
type SigP' = Sug :&: Pos :+: SigP
desugHom' :: TermHom SigP' SigP
desugHom' = propAnn desugHom
```

4.4.2 Composing Term Homomorphisms

Another benefit of the function type of term homomorphisms over term algebras is the simple fact that its domain $f\ a$ is independent of the target signature g:

```
type TermHom f g = ∀ a . f a → Context g a
```

This enables us to compose term homomorphisms:

```
(⊙) :: (Functor g, Functor h) ⇒
  TermHom g h → TermHom f g → TermHom f h
f ⊙ g = termHom f . g
```

Here we make use of the fact that *termHom* also allows us to apply a term homomorphism to a proper context—*termHom f* has type $\forall\ a\ .\ Context\ g\ a \to Context\ h\ a$.

Although the occurrence of the target signature in the domain of term algebras prevents them from being composed with each other, the composition with a term homomorphism is still possible:

```
(⊡) :: Functor g ⇒ Alg g a → TermHom f g → Alg f a
alg ⊡ talg = free alg id . talg
```

The ability to compose term homomorphisms with term algebras or other term homomorphisms allows us to perform program transformations in the vein of short-cut fusion [4]. For an example, recall that we have extended the evaluation to terms over Sig' by precomposing the evaluation function with the desugaring function:

$$\begin{aligned}&eval' :: Term\ Sig' \to Term\ Val\\&eval' = eval\ .\ desug\end{aligned}$$

The same can be achieved by composing on the level of algebras respectively term homomorphisms instead of the level of functions:

$$\begin{aligned}&eval' :: Term\ Sig' \to Term\ Val\\&eval' = cata\ (evalAlg \boxdot desugHom)\end{aligned}$$

Using the rewrite mechanism of GHC [7], we can make this optimisation automatic, by including the following rewrite rule:

$$\begin{aligned}&\texttt{"cata/termHom"}\ \forall\ (a :: Alg\ g\ d)\ (h :: TermHom\ f\ g)\ x\ .\\&\quad cata\ a\ (termHom\ h\ x) = cata\ (a \boxdot h)\ x\end{aligned}$$

One can easily show that this transformation is sound. Moreover, a similar rule can be devised for composing two term homomorphisms. The run-time benefits of these optimisation rules are considerable as we will see in Section 6.2.

4.4.3 Monadic Term Homomorphisms

Like catamorphisms, we can also easily lift term homomorphisms to monadic computations. We only need to lift the computations to a monadic type and use *mapM* instead of *fmap* for the recursion respectively use monadic function composition $\Lleftarrow$ instead of pure function composition:

$$\begin{aligned}&\textbf{type}\ TermHomM\ m\ f\ g = \forall\ a\ .\ f\ a \to m\ (Context\ g\ a)\\&termHomM :: (Traversable\ f, Functor\ g, Monad\ m)\\&\quad \Rightarrow TermHomM\ m\ f\ g \to Cxt\ h\ f\ a \to m\ (Cxt\ h\ g\ a)\\&termHomM\ f\ (Term\ t) =\\&\quad liftM\ appCxt\ .\ f \Lleftarrow mapM\ (termHomM\ f)\ t\\&termHomM\ \_\ (Hole\ b)\ = return\ (Hole\ b)\end{aligned}$$

The same strategy yields monadic variants of $\circledcirc$ and $\boxdot$

$$\begin{aligned}&(\hat{\circledcirc}) :: (Traversable\ g, Functor\ h, Monad\ m) \Rightarrow\\&\quad TermHomM\ m\ g\ h \to TermHomM\ m\ f\ g\\&\qquad\qquad \to TermHomM\ m\ f\ h\\&f \hat{\circledcirc}\ g = termHomM\ f \Lleftarrow g\\&(\hat{\boxdot}) :: (Traversable\ g, Monad\ m) \Rightarrow\\&\quad AlgM\ m\ g\ a \to TermHomM\ m\ f\ g \to AlgM\ m\ f\ a\\&alg \hat{\boxdot} talg = freeM\ alg\ return \Lleftarrow talg\end{aligned}$$

In contrast to pure term homomorphisms, one has to be careful when applying these composition operators: The fusion equation

$$termHomM\ (f \mathbin{\hat{\circledcirc}} g) = termHomM\ f \lessdot\!\!\!\Leftarrow termHomM\ g$$

does not hold in general! However, Fokkinga [3] showed that for monads satisfying a certain distributivity law, the above equation indeed holds. An example of such a monad is the *Maybe* monad. Furthermore, the equation is also true whenever one of the term homomorphisms is in fact pure, i.e. of the form $return\,.\,h$ for a non-monadic term homomorphism h. The same also applies to the fusion equation for $\hat{\boxdot}$. Nevertheless, it is still possible to devise rewrite rules that perform short-cut fusion under these restrictions.

An example of a monadic term homomorphism is the following function that recursively coerces a term to a sub-signature:

$$\begin{aligned}
&deepProject\ ::\ (Functor\ g, Traversable\ f, g :\prec f)\\
&\qquad\qquad \Rightarrow Term\ f \to Maybe\ (Term\ g)\\
&deepProject = termHomM\ (liftM\ simpCxt\ .\ proj)
\end{aligned}$$

As *proj* is, in fact, a monadic *symbol-to-symbol* term homomorphism we have to compose it with *simpCxt* to obtain a general monadic term homomorphism.

4.5 Beyond Catamorphisms

So far we have only considered (monadic) algebras and their (monadic) catamorphisms. It is straightforward to implement the machinery for programming in coalgebras and their anamorphisms:

$$\begin{aligned}
&\textbf{type}\ Coalg\ f\ a = a \to f\ a\\
&ana :: Functor\ f \Rightarrow Coalg\ f\ a \to a \to Term\ f\\
&ana\ f\ x = Term\ (fmap\ (ana\ f)\ (f\ x))
\end{aligned}$$

In fact, also more advanced recursion schemes can be accounted for in our framework: This includes paramorphisms and histomorphisms as well as their dual notions of apomorphisms and futumorphisms [22]. Similarly, monadic variants of these recursion schemes can be derived using the type class *Traversable*.

As an example of the abovementioned recursion schemes, we want to single out *futumorphisms*, as they can be represented conveniently using contexts and in fact are more natural to program than run-of-the-mill anamorphisms. The algebraic counterpart of futumorphisms are *cv-coalgebras* [22]. In their original algebraic definition they look rather cumbersome (cf. [22, Ch. 4.3]). If we implement cv-coalgebras in Haskell using contexts, the computation they denote becomes clear immediately:

$$\textbf{type}\ CVCoalg\ f\ a = a \to f\ (Context\ f\ a)$$

Anamorphisms only allow us to construct the target term one layer at a time. This can be plainly seen from the type $a \to f\ a$ of coalgebras. Futumorphisms on the other hand allow us to construct an arbitrary large part of the target term. Instead of only producing a single application of a constructor, cv-coalgebras produce a *non-empty* context, i.e. a context of height at least 1. The non-emptiness of the produced contexts guarantees that the resulting futumorphism is *productive*.

For the sake of brevity, we lift this restriction to non-empty contexts and consider *generalised cv-coalgebras*:

$$\textbf{type } \mathit{CVCoalg}\ f\ a = a \to \mathit{Context}\ f\ a$$

Constructing the corresponding futumorphism is simple and almost the same as for anamorphisms:

$$\begin{aligned}&\mathit{futu} :: \mathit{Functor}\ f \Rightarrow \mathit{CVCoalg}\ f\ a \to a \to \mathit{Term}\ f\\&\mathit{futu}\ f\ x = \mathit{appCxt}\ (\mathit{fmap}\ (\mathit{futu}\ f)\ (f\ x))\end{aligned}$$

Generalised cv-coalgebras also occur when composing a coalgebra and a term homomorphism, which can be implemented by plain function composition:

$$\begin{aligned}&\mathit{compCoa} :: \mathit{TermHom}\ f\ g \to \mathit{Coalg}\ f\ a \to \mathit{CVCoalg}\ g\ a\\&\mathit{compCoa}\ \mathit{hom}\ \mathit{coa} = \mathit{hom}\ .\ \mathit{coa}\end{aligned}$$

This can then be lifted to the composition of a generalised cv-coalgebra and a term homomorphism, by running the term homomorphism:

$$\begin{aligned}&\mathit{compCVCoalg} :: (\mathit{Functor}\ f, \mathit{Functor}\ g)\\&\quad \Rightarrow \mathit{TermHom}\ f\ g \to \mathit{CVCoalg}\ f\ a \to \mathit{CVCoalg}\ g\ a\\&\mathit{compCVCoalg}\ \mathit{hom}\ \mathit{coa} = \mathit{termHom}\ \mathit{hom}\ .\ \mathit{coa}\end{aligned}$$

With generalised cv-coalgebras one has to be careful, though, as they might not be productive. However, the above constructions can be replicated with ordinary cv-coalgebras. Instead of general term homomorphisms, we have to restrict ourselves to *ε-free* term homomorphisms [2] which are captured by the type:

$$\textbf{type } \mathit{TermHom}'\ f\ g = \forall\ a\ .\ f\ a \to g\ (\mathit{Context}\ g\ a)$$

This illustrates that with the help of contexts, (generalised) futumorphisms provide a much more natural coalgebraic programming model than anamorphisms.

5 Mutually Recursive Data Types and GADTs

Up to this point we have only considered the setting of a single recursively defined data type. We argue that this is the most common setting in the area we are targeting, viz. processing and analysing abstract syntax trees. Sometimes it is, however, convenient to encode certain invariants of the data structure, e.g. well-typing of ASTs, as mutually recursive data types or GADTs. In this section, we will show how this can be encoded as a family of compositional data types by transferring the construction of Johann and Ghani [6] to compositional data types.

Recall that the idea of representing recursive data types as fixed points of functors is to abstract from the recursive reference to the data type that should be defined. Instead of a recursive data type

$$\textbf{data } \mathit{Exp}\ = \cdots \mid \mathit{Mult}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Fst}\ \mathit{Exp}$$

we define a functor

$$\textbf{data}\ \mathit{Sig}\ e = \cdots \mid \mathit{Mult}\ e \quad e \quad \mid \mathit{Fst}\ e$$

The trick for defining mutually recursive data types is to use phantom types as labels that indicate which data type we are currently in. As an example, reconsider our simple expression language over integers and pairs. But now we define them in a family of two mutually recursive data types in order to encode the expected invariants of the expression language, e.g. the sum of two integers yields an integer:

$$\begin{aligned}\textbf{data}\ \mathit{IExp} &= \mathit{Const}\ \mathit{Int} \mid \mathit{Mult}\ \mathit{IExp}\ \mathit{IExp} \\ &\mid \mathit{Fst}\ \mathit{PExp} \mid \mathit{Snd}\ \mathit{PExp} \\ \textbf{data}\ \mathit{PExp} &= \mathit{Pair}\ \mathit{IExp}\ \mathit{IExp}\end{aligned}$$

We can encode this on signatures by adding an additional type argument which indicates the data types we are expecting as arguments to the constructors:

$$\begin{aligned}&\textbf{data}\ \mathit{Pair} \\ &\textbf{data}\ \mathit{ISig}\ e\ l = \mathit{Const}\ \mathit{Int} \mid \mathit{Mult}\ (e\ \mathit{Int})\ (e\ \mathit{Int}) \\ &\qquad\qquad \mid \mathit{Fst}\ (e\ \mathit{Pair}) \mid \mathit{Snd}\ (e\ \mathit{Pair}) \\ &\textbf{data}\ \mathit{PSig}\ e\ l = \mathit{Pair}\ (e\ \mathit{Int})\ (e\ \mathit{Int})\end{aligned}$$

Notice that the type variable e that is inserted in lieu of recursion is now of kind $* \to *$ as we consider a family of types. The "label type"—*Int* respectively *Pair*—then selects the desired type from this family. The definitions above, however, only indicate which data type we are expecting, e.g. *Mult* expects two integer expressions and *Swap* a pair expression. In order to also label the result type accordingly, we rather want to define *ISig* and *PSig* as

$$\begin{aligned}&\textbf{data}\ \mathit{ISig}\ e\ \mathit{Int} = \ldots \\ &\textbf{data}\ \mathit{PSig}\ e\ \mathit{Pair} = \ldots\end{aligned}$$

Using GADTs we can do this, although in a syntactically more verbose way:

$$\begin{aligned}&\textbf{data}\ \mathit{ISig}\ e\ l\ \textbf{where} \\ &\quad \mathit{Const} :: \mathit{Int} \to \mathit{ISig}\ e\ \mathit{Int} \\ &\quad \mathit{Mult} :: e\ \mathit{Int} \to e\ \mathit{Int} \to \mathit{ISig}\ e\ \mathit{Int} \\ &\quad \mathit{Fst}, \mathit{Snd} :: e\ \mathit{Pair} \to \mathit{ISig}\ e\ \mathit{Int} \\ &\textbf{data}\ \mathit{PSig}\ e\ l\ \textbf{where} \\ &\quad \mathit{Pair} :: e\ \mathit{Int} \to e\ \mathit{Int} \to \mathit{PSig}\ e\ \mathit{Pair}\end{aligned}$$

Notice that signatures are not functors of kind $* \to *$ anymore. Instead, they have the kind $(* \to *) \to (* \to *)$, thus adding one level of indirection.

Following previous work [6, 25], we can formulate the actual recursive definition of terms as follows:

$$\textbf{data}\ \mathit{Term}\ f\ l = \mathit{Term}\ (f\ (\mathit{Term}\ f)\ l)$$

The first argument f is a signature, i.e. has the kind $(* \to *) \to (* \to *)$. The type constructor *Term* recursively applies the signature f while propagating the

label l according to the signature. Note that *Term f* is of kind $* \to *$. A value of type *Term f l* is a mutually recursive data structure with topmost label l. In the recursive definition, *Term f* is applied to a signature f, i.e. in the case of f being *ISig* or *PSig* it instantiates the type variable e in their respective definitions. The type signatures of *ISig* and *PSig* can thus be read as propagation rules for the labels: For example, *Fst* takes a term with top-level labeling *Pair* and returns a term with top-level labeling *Int*.

5.1 Higher-Order Functors

It is important to realise that the transition to a family of mutually recursive data types amounts to nothing more than adding a layer of indirection. A signature, which has previously been a functor, is now a (generalised) *higher-order functor* [6]:

```
type f :-> g = forall a . f a -> g a
class HFunctor h where
  hfmap :: f :-> g -> h f :-> h g
instance HFunctor ISig where
  hfmap _ (Const i)  = Const i
  hfmap f (Mult x y) = Mult (f x) (f y)
  hfmap f (Fst x)    = Fst (f x)
```

The function *hfmap* witnesses that a natural transformation $f \dot{\to} g$ from functor f to functor g is mapped to a natural transformation $h\ f \dot{\to} h\ g$.

Observe the simplicity of the pattern that we used to lift our representation of compositional data types to mutually recursive types: Replace functors with higher-order functors, and instead of the function space $\to$ consider the natural transformation space $\dot{\to}$. This simple pattern will turn out to be sufficient in order to lift most of the concepts of compositional data types to mutually recursive data types. Sums and injections can thus be represented as follows:

```
data (f :+: g) (a :: * -> *) l = Inl (f a l) | Inr (g a l)
type NatM m f g = forall i . f i -> m (g i)
class (sub :: (* -> *) -> * -> *) :<: sup where
  inj :: sub a :-> sup a
  proj :: NatM Maybe (sup a) (sub a)
```

Lifting *HFunctor* instances to sums works in the same way as we have seen for *Functor*. The same goes for instances of $:\prec:$.

With the summation :+: in place we can define the family of data types that defines integer and pair expressions:

```
type Expr = Term (ISig :+: PSig)
```

This is indeed a family of types. We obtain the type of integer expressions with *Expr Int* and the type of pair expressions as *Expr Pair*.

5.2 Representing GADTs

Before we continue with lifting recursion schemes such as catamorphisms to the higher-order setting, we reconsider our example of mutually recursive data types. In contrast to the representation using a single recursive data type, the definition of *IExp* and *PExp* does not allow nested pairs—pairs are always built from integer expressions. The same goes for *Expr Int* and *Expr Pair*, respectively. This restriction is easily lifted by using a GADT instead:

$$
\begin{array}{l}
\textbf{data}\ \mathit{SExp}\ l\ \textbf{where} \\
\quad \begin{array}{lllll}
\mathit{Const} & :: & \mathit{Int} & \to \mathit{SExp}\ \mathit{Int} \\
\mathit{Mult} & :: \mathit{SExp}\ \mathit{Int} \to & \mathit{SExp}\ \mathit{Int} & \to \mathit{SExp}\ \mathit{Int} \\
\mathit{Fst} & :: & \mathit{SExp}\ (s,t) & \to \mathit{SExp}\ s \\
\mathit{Snd} & :: & \mathit{SExp}\ (s,t) & \to \mathit{SExp}\ t \\
\mathit{Pair} & :: \mathit{SExp}\ s \to & \mathit{SExp}\ t & \to \mathit{SExp}\ (s,t)
\end{array}
\end{array}
$$

This standard GADT representation can be mapped directly to our signature definitions. However, instead of defining a single GADT, we proceed as we did with non-mutually recursive compositional data types. We split the signature into values and operations:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Val}\ e\ l\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Const} & :: & \mathit{Int} \to \mathit{Val}\ e\ \mathit{Int} \\
\mathit{Pair} & :: e\ s \to e\ t & \to \mathit{Val}\ e\ (s,t)
\end{array} \\
\textbf{data}\ \mathit{Op}\ e\ l\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Mult} & :: e\ \mathit{Int} \to e\ \mathit{Int} & \to \mathit{Op}\ e\ \mathit{Int} \\
\mathit{Fst} & :: & e\ (s,t) \to \mathit{Op}\ e\ s \\
\mathit{Snd} & :: & e\ (s,t) \to \mathit{Op}\ e\ t
\end{array} \\
\textbf{type}\ \mathit{Sig} = \mathit{Op} :+: \mathit{Val}
\end{array}
$$

Combining the above two signatures then yields the desired family of mutually recursive data types $\mathit{Term}\ \mathit{Sig} \cong \mathit{SExp}$.

This shows that the transition to higher-order functors also allows us to naturally represent GADTs in a modular fashion.

5.3 Recursion Schemes

We shall continue to apply the pattern for shifting to mutually recursive data types: Replace *Functor* with *HFunctor* and function space $\to$ with the space of natural transformations $\dot{\to}$. Take, for example, algebras and catamorphisms:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Alg}\ f\ a = f\ a \dot{\to} a \\
\mathit{cata} :: \mathit{HFunctor}\ f \Rightarrow \mathit{Alg}\ f\ a \to \mathit{Term}\ f \dot{\to} a \\
\mathit{cata}\ f\ (\mathit{Term}\ t) = f\ (\mathit{hfmap}\ (\mathit{cata}\ f)\ t)
\end{array}
$$

Now, an algebra has a family of types $a :: * \to *$ as carrier. That is, we have to move from algebras to *many-sorted* algebras. Representing many-sorted algebras comes quite natural in most cases. For example, the evaluation algebra class can be recast as a many-sorted algebra class as follows:

```
class Eval e v where
  evalAlg :: Alg e (Term v)
eval :: (HFunctor e, Eval e v) ⇒ Term e ·→ Term v
eval = cata evalAlg
```

Here, we can make use of the fact that *Term v* is in fact a family of types and can thus be used as a carrier of a many-sorted algebra.

Except for the slightly more precise type of *projC* and *projP*, the definition of *Eval* is syntactically equal to its non-mutually recursive original from Section 2.1:

```
instance (Val :≺ v) ⇒ Eval Val v where
  evalAlg = inject
instance (Val :≺ v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst $ projC x ∗ projC y
  evalAlg (Fst x)    = fst $ projP x
  evalAlg (Snd x)    = snd $ projP x
projC :: (Val :≺ v) ⇒ Term v Int → Int
projC v = case project v of Just (Const n) → n
projP :: (Val :≺ v) ⇒ Term v (s, t) → (Term v s, Term v t)
projP v = case project v of Just (Pair x y) → (x, y)
```

In some cases, it might be a bit more cumbersome to define and use the carrier of a many-sorted algebra. However, most cases are well-behaved and we can use the family of terms *Term f* as above or alternatively the identity respectively the constant functor:

```
data I a    = I {unI :: a}
data K a b = K {unK :: a}
```

For example, a many-sorted algebra class to evaluate expressions directly into Haskell values of the corresponding types can be defined as follows:

```
class EvalI f where
  evalAlgI :: Alg f I
evalI :: (EvalI f, HFunctor f) ⇒ Term f t → t
evalI = unI . cata evalAlgI
```

The lifting of other recursion schemes whether algebraic or coalgebraic can be achieved in the same way as illustrated for catamorphisms above. The necessary changes are again quite simple. Similarly to the type class *HFunctor*, we can obtain lifted versions of *Foldable* and *Traversable* which can then be used to implement generic programming techniques and to perform monadic computations, respectively. The generalisation of terms to contexts and the corresponding notion of term homomorphisms is also straightforward. The same short-cut fusion rules that we have considered for simple compositional data types can be implemented without any surprises as well.

The only real issue worth mentioning is that the generic querying combinator *query* needs to produce result values of a fixed type as opposed to a family of

types. The propagation of types defined by GADTs cannot be captured by the simple pattern of the querying combinator. Thus, the querying combinator is typed as follows:

$$\begin{aligned} &\mathit{query} :: \mathit{HFoldable}\ f \Rightarrow (\forall\ i\ .\ \mathit{Term}\ f\ i \to r) \\ &\qquad \to (r \to r \to r) \to \mathit{Term}\ f\ i \to r \end{aligned}$$

For the *subs* combinator, which produces a list of all subterms, the issue is similar: *Term f* is a type family, thus [*Term f*] is not a valid type. However, we can obtain the desired type of list of terms by existentially quantifying over the index type using the GADT

$$\mathbf{data}\ A\ f = \forall\ i\ .\ A\ (f\ i)$$

The type of *subs* can now be stated as follows:

$$\mathit{subs} :: \mathit{HFoldable}\ f \Rightarrow \mathit{Term}\ f\ i \to [A\ (\mathit{Term}\ f)]$$

6 Practical Considerations

Besides showing the expressiveness and usefulness of the framework of compositional data types, we also want to showcase its practical applicability as a software development tool. To this end, we consider aspects of *usability* and *performance impacts* as well.

6.1 Generating Boilerplate Code

The implementation of recursion schemes depends on the signatures being instances of the type class *Functor*. For generic programming techniques and monadic computations, we rely on the type classes *Foldable* and *Traversable*, respectively. Additionally, higher-order functors necessitate a set of lifted variants of the abovementioned type classes. That is a lot of boilerplate code! Writing and maintaining this code would almost entirely defeat the advantage of using compositional data types in the first place.

Luckily, by leveraging Template Haskell [17], instance declarations of all generic type classes that we have mentioned in this paper can be generated automatically at compile time similar to Haskell's **deriving** mechanism. Even though some Haskell packages such as *derive* already provide automatically derived instances for some of the standard classes like *Functor*, *Foldable* and *Traversable*, we chose to implement the instance generators for these as well. The heavy use of the methods of these classes for implementing recursion schemes means that they contribute considerably to the computational overhead! Automatically deriving instance declarations with carefully optimised implementations of each of the class methods, have proven to yield substantial run-time improvements, especially for monadic computations.

We already mentioned that we assume with each constructor

$$\mathit{Constr} :: t_1 \to \cdots \to t_n \to f\ a$$

of a signature f, a smart constructor defined by

$$
\begin{aligned}
&\mathit{iConstr} :: f :\prec: g \Rightarrow s_1 \to \cdots \to s_n \to \mathit{Term}\ g \\
&\mathit{iConstr}\ x_1 \ldots x_n = \mathit{inject}\ \$\ \mathit{Constr}\ x_1 \ldots x_n
\end{aligned}
$$

where the types s_i are the same as t_i except with occurrences of the type variable a replaced by $\mathit{Term}\ g$. These smart constructors can be easily generated automatically using Template Haskell.

Another issue is the declaration of instances of type classes *Eq*, *Ord* and *Show* for types of the form $\mathit{Term}\ f$. This can be achieved by lifting these type classes to functors, e.g. for *Eq*:

```
class EqF f where
  eqF :: Eq a ⇒ f a → f a → Bool
```

From instances of this class, corresponding instances of *Eq* for terms and contexts can be derived:

```
instance (EqF f, Eq a) ⇒ Eq (Cxt h f a) where
  (≡) (Term t1) (Term t2) = t1 `eqF` t2
  (≡) (Hole h1) (Hole h2) = h1 ≡ h2
  (≡) _         _         = False
```

Instances of *EqF*, *OrdF* and *ShowF* can be derived straightforwardly using Template Haskell which then yield corresponding instances of *Eq*, *Ord* and *Show* for terms and contexts. The thus obtained instances are equivalent to the ones obtained from Haskell's **deriving** mechanism on corresponding recursive data types.

Figure 1 demonstrates the *complete* source code needed in order to implement some of the earlier examples in our library.

6.2 Performance Impact

In order to minimise the overhead of the recursion schemes, we applied some simple optimisations to the implementation of the recursion schemes themselves. For example, *cata* is defined as

```
cata :: ∀ f a . Functor f ⇒ Alg f a → Term f → a
cata f = run
  where run :: Term f → a
        run (Term t) = f (fmap run t)
```

The biggest speedup, however, can be obtained by providing automatically generated, carefully optimised implementations for each method of the type classes *Foldable* and *Traversable*.

In order to gain speedup in the implementation of generic programming combinators, we applied the same techniques as Mitchell and Runciman [12] by leveraging short-cut fusion [4] via *build*. The *subs* combinator is thus defined as:

```
subs :: ∀ f . Foldable f ⇒ Term f → [Term f]
subs t = build (f t) where
```

```
import Data.Comp
import Data.Comp.Derive
import Data.Comp.Show ()
import Data.Comp.Desugar

data Val e = Const Int | Pair e e
data Op e  = Mult e e | Fst e | Snd e
data Sug e = Neg e | Swap e
type Sig   = Op :+: Val
type Sig'  = Sug :+: Sig

$(derive [makeFunctor, makeFoldable, makeTraversable,
          makeShowF, smartConstructors] [''Val, ''Op, ''Sug])

-- * Term Evaluation
class Eval f v where evalAlg :: Alg f (Term v)

$(derive [liftSum] [''Eval]) -- lift Eval to coproducts

eval :: (Functor f, Eval f v) ⇒ Term f → Term v
eval = cata evalAlg

instance (Val :<: v) ⇒ Eval Val v where
  evalAlg = inject

instance (Val :<: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst $ projC x * projC y
  evalAlg (Fst x)    = fst $ projP x
  evalAlg (Snd x)    = snd $ projP x

projC :: (Val :<: v) ⇒ Term v → Int
projC v = case project v of Just (Const n) → n

projP :: (Val :<: v) ⇒ Term v → (Term v, Term v)
projP v = case project v of Just (Pair x y) → (x,y)

-- * Desugaring
instance (Op :<: f, Val :<: f, Functor f) ⇒ Desugar Sug f where
  desugHom' (Neg x)  = iConst (-1) `iMult` x
  desugHom' (Swap x) = iSnd x `iPair` iFst x

eval' :: Term Sig' → Term Val
eval' = eval . (desugar :: Term Sig' → Term Sig)
```

Figure 1: Example usage of the compositional data types library.

Function	hand-written	random (10)	random (20)
desugHom	$3.6 \cdot 10^{-1}$	$5.0 \cdot 10^{-3}$	$6.1 \cdot 10^{-6}$
desugCata	$1.8 \cdot 10^{-1}$	$4.41 \cdot 10^{-3}$	$5.3 \cdot 10^{-6}$
inferDesug	(3.38) 1.11	(3.45) 1.52	(3.14) 0.82
inferDesugM	(2.68) 1.38	(2.87) 1.61	(2.79) 0.84
infer	2.39	2.29	2.65
inferM	1.06	1.30	1.68
evalDesug	(6.40) 2.64	(3.13) 1.79	(4.74) 0.89
evalDesugM	(7.32) 4.34	(6.22) 3.47	(9.69) 2.98
eval	2.58	1.84	1.64
evalDirect	6.10	3.96	3.62
evalM	3.41	4.78	7.52
evalDirectM	5.72	4.90	4.56
contVar	1.92	1.97	3.22
freeVars	1.23	1.26	1.41
contVarC	10.05	7.01	11.68
contVarU	8.24	5.64	11.21
freeVarsC	2.34	2.04	1.68
freeVarsU	2.03	1.75	1.58

Table 1: Run-time of functions on compositional data types (as multiples of the run-time of an implementation using ordinary algebraic data types).

$$f :: Term\ f \to (Term\ f \to b \to b) \to b \to b$$
$$f\ t\ cons\ nil = t\ `cons`\ foldl\ (\lambda u\ s \to f\ s\ cons\ u)\ nil\ (unTerm\ t)$$

Instead of building the result list directly, we use the *build* combinator which then can be eliminated if combined with a consumer such as a fold or a list comprehension.

Table 1 shows the run-time performance of our framework for various functions dealing with ASTs: Desugaring (*desug*), type inference (*infer*), expression evaluation (*eval*), and listing respectively searching for free variables (*freeVars*, *contVar*). The *Hom* and *Cata* version of *desug* differ in that the former is defined as a term homomorphism, the latter as a catamorphism. For *eval* and *infer*, the suffix *Desug* indicates that the computation is prefixed by a desugaring phase (using *desugHom*), the suffix *M* indicates monadic variants (for error handling), and *Direct* indicates that the function was implemented not as a catamorphism but using explicit recursion. The numbers in the table are multiples of the run-time of an implementation using ordinary algebraic data types and recursion. The numbers in parentheses indicate the run-time factor if the automatic short-cut fusion described in Section 4.4.2 is disabled. Each function is tested on three different inputs of increasing size. The first is a hand-written "natural" expression consisting of 16 nodes. The other two expressions are randomly generated expressions of depth 10 and 20, respectively, which corresponds to approximately 800 respectively 200,000 nodes. This should reveal how the overhead of our framework scales. The benchmarks were performed with the *criterion* framework using GHC 7.0.2 with optimisation flag `-O2`.

As a pleasant surprise, we observe that the penalty of using compositional data types is comparatively low. It is in the same ballpark as for generic programming libraries [12, 15]. For some functions we even obtain a speedup! The biggest surprise is, however, the massive speedup gained by the desugaring function. In both its catamorphic and term-homomorphic version, it seems to perform asymptotically better than the classic implementation, yielding a speedup of over five orders of magnitude. We were also surprised to see that (except for one case) functions programmed as catamorphisms outperformed functions using explicit recursion! In fact, with GHC 6.12, the situation was reversed.

Moreover, we observe that the short-cut fusion rules implemented in our framework uniformly yield a considerable speedup of up to factor five. As a setback, however, we have to recognise that implementing desugaring as a term homomorphism yields a slowdown of factor up to two compared to its catamorphic version.

Finally, we compared our implementation of generic programming techniques with Uniplate [12], one of the top-performing generic programming libraries. In particular, we looked at its *universe* combinator which computes the list of all subexpressions. We have implemented this combinator in our framework as *subs*. In Table 1, our implementation is indicated by the suffix C, the Uniplate implementation, working on ordinary algebraic data types, is indicated by U. We can see that we are able to obtain comparable performance in all cases.

7 Discussion

Starting from Swierstra's *data types à la carte* [19], we have constructed a framework for representing data types in a compositional fashion that is readily usable for practical applications. Our biggest contribution is the generalisation of terms to contexts which allow us to capture the notion of term homomorphisms. Term homomorphisms provide a rich structure that allows flexible reuse and enables simple but effective optimisation techniques. Moreover, term homomorphisms can be easily extended with a state. Depending on how the state is propagated, this yields bottom-up respectively top-down tree transducers [2]. The techniques for short-cut fusion and propagation of annotations can be easily adapted.

7.1 Related Work

The definition of monadic catamorphisms that we use goes back to Fokkinga [3]. He only considers monads satisfying a certain distributivity law. However, this distributivity is only needed for the fusion rules of Section 4.4.3 to be valid. Steenbergen et al. [21] use the same approach to implement catamorphisms with errors. In contrast, Visser and Löh [23] consider monadic catamorphism for which the monadic effect is part of the term structure.

The construction to add annotations to functors is also employed by Steenbergen et al. [21] to add detailed source position annotations to ASTs. However, since they are considering general catamorphisms, they are not able to provide a means to propagate annotations. Moreover, since Steenbergen et al. do not account for sums of functors, the distribution of annotations over sums is not an

issue for them. Visser and Löh [23] consider a more general form of annotations via arbitrary *functor transformations*. Unfortunately, this generality prohibits the automatic propagation of annotations as well as their distribution over sums.

Methods to represent mutually recursive data types as fixed points of (regular) functors have been explored to some extent [1, 8, 18, 25]. All of these techniques are limited to mutually recursive data types in which the number of nested data types is limited up front and are thus not compositional. However, in the representation of Yakushev et al. [25], the restriction to mutually recursive data types with a closed set of constituent data types was implemented intentionally. Our representation simply removes these restrictions which would in fact add no benefit in our setting. The resulting notion of higher-order functors that we considered was also used by Johann and Ghani [6] in order to represent GADTs.

7.2 Future Work

There are a number of aspects that are still missing which should be the subject of future work: As we have indicated, the restriction of the subtyping class $:\prec:$ hinders full compositionality of signature summation $:+:$. A remedy could be provided with a richer type system as proposed by Yorgey [26]. This would also allow us to define the right-distributivity of annotations $:\&:$ over sums $:+:$ more directly by a type family. Alternatively, this issue can be addressed with type instance-chains as proposed by Morris and Jones [13]. Another issue of Swierstra's original work is the *project* function which allows us to inspect terms ad-hoc. Unfortunately, it does not allow us to give a complete case analysis. In order to provide this, we need a function of type

$$(f :\prec: g) \Rightarrow \mathit{Term}\ g \to \mathit{Either}\ (f\ (\mathit{Term}\ g))\ ((g :-: f)\ (\mathit{Term}\ g))$$

which allows us to match against the "remainder signature" $g :-: f$.

Bibliography

[1] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999. ISSN 0934-5043. doi: 10.1007/s001650050047.

[2] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on `http://www.grappa.univ-lille3.fr/tata`, 2008.

[3] M. M. Fokkinga. A Gentle Introduction to Category Theory: the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, 1992.

[4] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, New York, NY, USA, 1993. ACM. doi: 10.1145/165180.165214.

[5] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical Views on Computations on Trees (Extended Abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 619–630. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73420-8˙54.

[6] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL '08*, pages 297–308, New York, New York, USA, 2008. ACM Press. doi: 10.1145/1328438.1328475.

[7] S. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, page 203, 2001.

[8] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3):255–279, 1990. ISSN 0167-6423. doi: 10.1016/0167-6423(90)90023-7.

[9] S. Marlow. Haskell 2010 Language Report, 2010.

[10] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. ISSN 09567968. doi: 10.1017/S0956796807006326.

[11] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961˙7.

[12] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 49–60, New York, NY, USA, 2007. ACM. doi: 10.1145/1291201.1291208.

[13] J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 375–386, New York, NY, USA, 2010. ACM. doi: 10.1145/1863543.1863596.

[14] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411301.

[15] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. Technical report, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2008.

[16] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM*

SIGPLAN International Conference on Functional Programming, pages 341–352, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596599.

[17] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, volume 37 of *SIGPLAN Notices*, pages 60–75, New York, NY, USA, 2002. ACM. doi: 10.1145/636517.636528.

[18] S. Swierstra, P. Azero Alcocer, and J. Saraiva. Designing and Implementing Combinator Languages. In S. Swierstra, J. Oliveira, and P. Henriques, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer Berlin / Heidelberg, 1999. doi: 10.1007/10704973˙4.

[19] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758.

[20] J. W. Thatcher. Tree automata: an informal survey. In A. V. Aho, editor, *Currents in the theory of computing*, chapter 4, pages 143–178. Prentice Hall, 1973.

[21] M. Van Steenbergen, J. P. Magalhães, and J. Jeuring. Generic selections of subexpressions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, pages 37–48, New York, NY, USA, 2010. ACM. doi: 10.1145/1863495.1863501.

[22] V. Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, Estonia, 2000.

[23] S. Visser and A. Löh. Generic storage in Haskell. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, pages 25–36, New York, NY, USA, 2010. ACM. doi: 10.1145/1863495.1863500.

[24] P. Wadler. The Expression Problem. Available on `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[25] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596585.

[26] B. Yorgey. Typed type-level functional programming in GHC. Talk at Haskell Implementors Workshop, 2010.

Parametric Compositional Data Types

Patrick Bahr Tom Hvitved

Department of Computer Science, University of Copenhagen

Abstract

In previous work we have illustrated the benefits that *compositional data types* (CDTs) offer for implementing languages and in general for dealing with abstract syntax trees (ASTs). Based on Swierstra's *data types à la carte*, CDTs are implemented as a Haskell library that enables the definition of recursive data types and functions on them in a modular and extendable fashion. Although CDTs provide a powerful tool for analysing and manipulating ASTs, they lack a convenient representation of variable binders. In this paper we remedy this deficiency by combining the framework of CDTs with Chlipala's parametric higher-order abstract syntax (PHOAS). We show how a generalisation from functors to difunctors enables us to capture PHOAS while still maintaining the features of the original implementation of CDTs, in particular its modularity. Unlike previous approaches, we avoid so-called *exotic terms* without resorting to abstract types: this is crucial when we want to perform *transformations* on CDTs that inspect the recursively computed CDTs, e.g. constant folding.

Contents

1 Introduction

When implementing domain-specific languages (DSLs)—either as embedded languages or stand-alone languages—the abstract syntax trees (ASTs) of programs are usually represented as elements of a recursive algebraic data type. These ASTs typically undergo various transformation steps, such as desugaring from a full language to a core language. But reflecting the invariants of these transformations in the type system of the host language can be problematic. For instance, in order to reflect a desugaring transformation in the type system, we must define a separate data type for ASTs of the core language. Unfortunately, this has the side effect that common functionality, such as pretty printing, has to be duplicated.

Wadler identified the essence of this issue as the *Expression Problem*, i.e. "the goal [. . .] to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety" [24]. Swierstra [22] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits invaluable simplicity and clarity, it lacks features necessary to apply it in a practical setting beyond the confined simplicity of the expression problem. To this end, the framework of *compositional data types* (CDTs) [4] provides a rich library for implementing practical functionality on highly modular data types. This includes support of a wide array of recursion schemes in both pure and monadic forms, as well as mutually recursive data types and generalised algebraic data types (GADTs) [18].

What CDTs fail to address, however, is a transparent representation of variable binders that frees the programmer's mind from common issues like computations modulo α-equivalence and capture-avoiding substitutions. The work we present in this paper fills that gap by adopting (a restricted form of) higher-order abstract syntax (HOAS) [15], which uses the host language's variable binding mechanism to represent binders in the object language. Since implementing efficient recursion schemes in the presence of HOAS is challenging [8, 13, 19, 25], integrating this technique with CDTs is a non-trivial task.

Following a brief introduction to CDTs in Section 2, we describe how to achieve this integration as follows:

- We adopt parametric higher-order abstract syntax (PHOAS) [6], and we show how to capture this restricted form of HOAS via difunctors. The

thus obtained *parametric compositional data types* (PCDTs) allow for the definition of modular catamorphisms à la Fegaras and Sheard [8] in the presence of binders. Unlike previous approaches, our technique does not rely on abstract types, which is crucial for modular computations that are also modular in their result type (Section 3).

- We illustrate why monadic computations constitute a challenge in the parametric setting and we show how monadic catamorphisms can still be defined for a restricted class of PCDTs (Section 4).
- We show how to transfer the restricted recursion scheme of *term homomorphisms* [4] to PCDTs. Term homomorphisms enable the same flexibility for reuse and opportunity for deforestation [23] that we know from CDTs (Section 5).
- We show how to represent mutually recursive data types and GADTs by generalising PCDTs in the style of Johann and Ghani [10] (Section 6).
- We illustrate the practical applicability of our framework by means of a complete library example, and we show how to automatically derive functionality for deciding equality (Section 7).

Parametric compositional data types are available as a Haskell library[1], including numerous examples that are not included in this paper. All code fragments presented throughout the paper are written in (literate) Haskell [12], and the library relies on several language extensions that are currently only known to be supported by the Glasgow Haskell Compiler (GHC).

2 Compositional Data Types

Based on Swierstra's *data types à la carte* [22], compositional data types [4] (CDTs) provide a framework for manipulating recursive data structures in a type-safe, modular manner. The prime application of CDTs is within language implementation and AST manipulation, and we present the basic concepts of CDTs in this section. More advanced concepts are introduced in Sections 4, 5, and 6.

2.1 Motivating Example

Consider an extension of the lambda calculus with integers, addition, let expressions, and error signalling:

$$e ::= \lambda x.e \mid x \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{error}$$

Our goal is to implement a pretty printer, a desugaring transformation, constant folding, and a call-by-value interpreter for the simple language above. The desugaring transformation will turn let expressions $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ into $(\lambda x.e_2)\ e_1$. Constant folding and evaluation will take place after desugaring, i.e. both computations are only defined for the core language without let expressions.

[1] See `http://hackage.haskell.org/package/compdata`.

The standard approach to representing the language above is in terms of an algebraic data type:

$$\begin{aligned}&\textbf{type } \mathit{Var} = \mathit{String}\\&\textbf{data } \mathit{Exp} = \mathit{Lam}\ \mathit{Var}\ \mathit{Exp} \mid \mathit{Var}\ \mathit{Var} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lit}\ \mathit{Int}\\&\qquad\quad \mid\ \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Let}\ \mathit{Var}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Err}\end{aligned}$$

We may then straightforwardly define the pretty printer $\mathit{pretty} :: \mathit{Exp} \to \mathit{String}$. However, when we want to implement the desugaring transformation, we need a new algebraic data type:

$$\begin{aligned}&\textbf{data } \mathit{Exp}' = \mathit{Lam}'\ \mathit{Var}\ \mathit{Exp}' \mid \mathit{Var}'\ \mathit{Var} \mid \mathit{App}'\ \mathit{Exp}'\ \mathit{Exp}' \mid \mathit{Lit}'\ \mathit{Int}\\&\qquad\quad \mid\ \mathit{Plus}'\ \mathit{Exp}'\ \mathit{Exp}' \mid \mathit{Err}'\end{aligned}$$

That is, we need to replicate all constructors of *Exp*—except *Let*—into a new type Exp' of core expressions, in order to obtain a properly typed desugaring function $\mathit{desug} :: \mathit{Exp} \to \mathit{Exp}'$. Not only does this mean that we have to replicate the constructors, we also need to replicate common functionality, e.g. in order to obtain a pretty printer for Exp' we must either write a new function, or write an injection function $\mathit{Exp}' \to \mathit{Exp}$.

CDTs provide a solution that allows us to define the ASTs for (core) expressions without having to duplicate common constructors, and without having to give up on statically guaranteed invariants about the structure of the ASTs. CDTs take the viewpoint of data types as fixed points of functors [14], i.e. the definition of the AST data type is separated into non-recursive signatures (functors) on the one hand and the recursive structure on the other hand. For our example, we define the following signatures (omitting the straightforward *Functor* instance declarations):

$$\begin{aligned}&\textbf{data } \mathit{Lam}\ a = \mathit{Lam}\ \mathit{Var}\ a && \textbf{data } \mathit{Plus}\ a = \mathit{Plus}\ a\ a\\&\textbf{data } \mathit{Var}\ a\ = \mathit{Var}\ \mathit{Var} && \textbf{data } \mathit{Let}\ a\ = \mathit{Let}\ \mathit{Var}\ a\ a\\&\textbf{data } \mathit{App}\ a\ = \mathit{App}\ a\ a && \textbf{data } \mathit{Err}\ a = \mathit{Err}\\&\textbf{data } \mathit{Lit}\ a\ \ = \mathit{Lit}\ \mathit{Int}\end{aligned}$$

Signatures can then be combined in a modular fashion by means of a formal sum of functors:

$$\begin{aligned}&\textbf{data } (f :+: g)\ a = \mathit{Inl}\ (f\ a) \mid \mathit{Inr}\ (g\ a)\\&\textbf{instance } (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (f :+: g)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Inl}\ x)\ = \mathit{Inl}\ (\mathit{fmap}\ f\ x)\\&\quad \mathit{fmap}\ f\ (\mathit{Inr}\ x) = \mathit{Inr}\ (\mathit{fmap}\ f\ x)\\&\textbf{type } \mathit{Sig}\ \ = \mathit{Lam} :+: \mathit{Var} :+: \mathit{App} :+: \mathit{Lit} :+: \mathit{Plus} :+: \mathit{Err} :+: \mathit{Let}\\&\textbf{type } \mathit{Sig}' = \mathit{Lam} :+: \mathit{Var} :+: \mathit{App} :+: \mathit{Lit} :+: \mathit{Plus} :+: \mathit{Err}\end{aligned}$$

Finally, the type of terms over a (potentially compound) signature f can be constructed as the (least) fixed point of the signature f:

$$\textbf{data } \mathit{Term}\ f = \mathit{In}\ \{\mathit{out} :: f\ (\mathit{Term}\ f)\}$$

Modulo strictness, *Term Sig* is isomorphic to *Exp*, and *Term Sig′* is isomorphic to *Exp′*.

The use of formal sums entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the right tags automatically, injections are derived using a type class:

```
class sub :≺: sup where
  inj  :: sub a → sup a
  proj :: sup a → Maybe (sub a)
```

Using *overlapping instance* declarations, the subsignature relation $:\prec:$ can be constructively defined [22]. However, due to the limitations of Haskell's type class system, instances are restricted to the form $f :\prec: g$ where f is atomic, i.e. not a sum, and g is a right-associated sum, e.g. $g_1 :+: (g_2 :+: g_3)$ but not $(g_1 :+: g_2) :+: g_3$. With the carefully defined instances for $:\prec:$, injection and projection functions for terms can then be defined as follows:

```
inject :: (g :≺: f) ⇒ g (Term f) → Term f
inject = In . inj
project :: (g :≺: f) ⇒ Term f → Maybe (g (Term f))
project = proj . out
```

Additionally, in order to reduce the syntactic overhead, the CDTs library can automatically derive smart constructors that comprise the injections [4], e.g.

```
iPlus :: (Plus :≺: f) ⇒ Term f → Term f → Term f
iPlus x y = inject (Plus x y)
```

Using the derived smart constructors, we can then write expressions such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ without syntactic overhead:

```
e :: Term Sig
e = iLet "x" (iLit 2) ((iLam "y" (Var "y" `iPlus` Var "x")) `iApp` iLit 3)
```

In fact, the principal type of e is the *open* type:

$$(Lam :\prec: f, Var :\prec: f, App :\prec: f, Lit :\prec: f, Plus :\prec: f, Let :\prec: f) \Rightarrow Term\ f$$

which means that e can be used as a term over any signature containing at least these six signatures!

Next, we want to define the pretty printer, i.e. a function of type $Term\ Sig \to String$. In order to make a recursive function definition modular too, it is defined as the catamorphism of an algebra [14]:

```
type Alg f a = f a → a
cata :: Functor f ⇒ Alg f a → Term f → a
cata φ = φ . fmap (cata φ) . out
```

The advantage of this approach is that algebras can be easily combined over formal sums. A modular algebra definition is obtained by an open family of

algebras indexed by the signature and closed under forming formal sums. This is achieved as a type class:

```
class Pretty f where
  φPretty :: Alg f String
instance (Pretty f, Pretty g) ⇒ Pretty (f :+: g) where
  φPretty (Inl x) = φPretty x
  φPretty (Inr x) = φPretty x
pretty :: (Functor f, Pretty f) ⇒ Term f → String
pretty = cata φPretty
```

The instance declaration that lifts *Pretty* instances to sums is crucial. Yet, the structure of its declaration is independent from the particular algebra class, and the CDTs library provides a mechanism for automatically deriving such instances [4]. What remains in order to implement the pretty printer is to define instances of the *Pretty* algebra class for the six signatures:

```
instance Pretty Lam where
  φPretty (Lam x e) = "(\\" ++ x ++ ". " ++ e ++ ")"
instance Pretty Var where
  φPretty (Var x) = x
instance Pretty App where
  φPretty (App e1 e2) = "(" ++ e1 ++ " " ++ e2 ++ ")"
instance Pretty Lit where
  φPretty (Lit n) = show n
instance Pretty Plus where
  φPretty (Plus e1 e2) = "(" ++ e1 ++ " + " ++ e2 ++ ")"
instance Pretty Let where
  φPretty (Let x e1 e2) = "(let " ++ x ++ " = " ++ e1 ++
                          " in " ++ e2 ++ ")"
instance Pretty Err where
  φPretty Err = "error"
```

With these definitions we then have that *pretty e* evaluates to the string `(let x = 2 in ((\y. (y + x)) 3))`. Moreover, we automatically obtain a pretty printer for the core language as well, cf. the type of *pretty*.

3 Parametric Compositional Data Types

In the previous section we considered a first-order encoding of the language, which means that we have to be careful to ensure that computations are invariant under α-equivalence, e.g. when implementing capture-avoiding substitutions. *Higher-order abstract syntax* (HOAS) [15] remedies this issue, by representing binders and variables of the object language in terms of those of the meta language.

3.1 Higher-Order Abstract Syntax

In a standard Haskell HOAS encoding we replace the signatures *Var* and *Lam* by a revised *Lam* signature:

$$\textbf{data}\ \mathit{Lam}\ a = \mathit{Lam}\ (a \to a)$$

Now, however, *Lam* is no longer an instance of *Functor*, because a occurs both in a contravariant position and a covariant position. We therefore need to generalise functors in order to allow for negative occurrences of the recursive parameter. *Difunctors* [13] provide such a generalisation:

$$\begin{aligned}
&\textbf{class}\ \mathit{Difunctor}\ f\ \textbf{where}\\
&\quad \mathit{dimap} :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d\\
&\textbf{instance}\ \mathit{Difunctor}\ (\to)\ \textbf{where}\\
&\quad \mathit{dimap}\ f\ g\ h = g\ .\ h\ .\ f\\
&\textbf{instance}\ \mathit{Difunctor}\ f \Rightarrow \mathit{Functor}\ (f\ a)\ \textbf{where}\\
&\quad \mathit{fmap} = \mathit{dimap}\ \mathit{id}
\end{aligned}$$

A difunctor must preserve the identity function and distribute over function composition:

$$\mathit{dimap}\ \mathit{id}\ \mathit{id} = \mathit{id} \qquad \text{and} \qquad \mathit{dimap}\ (f\ .\ g)\ (h\ .\ i) = \mathit{dimap}\ g\ h\ .\ \mathit{dimap}\ f\ i$$

The derived *Functor* instance obtained by fixing the contravariant argument will hence satisfy the functor laws, provided that the difunctor laws are satisfied.

Meijer and Hutton [13] showed that it is possible to perform recursion over difunctor terms:

$$\begin{aligned}
&\textbf{data}\ \mathit{Term}_{MH}\ f = \mathit{In}_{MH}\ \{\mathit{out}_{MH} :: f\ (\mathit{Term}_{MH}\ f)\ (\mathit{Term}_{MH}\ f)\}\\
&\mathit{cata}_{MH} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to \mathit{Term}_{MH}\ f \to a\\
&\mathit{cata}_{MH}\ \phi\ \psi = \phi\ .\ \mathit{dimap}\ (\mathit{ana}_{MH}\ \phi\ \psi)\ (\mathit{cata}_{MH}\ \phi\ \psi)\ .\ \mathit{out}_{MH}\\
&\mathit{ana}_{MH} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to b \to \mathit{Term}_{MH}\ f\\
&\mathit{ana}_{MH}\ \phi\ \psi = \mathit{In}_{MH}\ .\ \mathit{dimap}\ (\mathit{cata}_{MH}\ \phi\ \psi)\ (\mathit{ana}_{MH}\ \phi\ \psi)\ .\ \psi
\end{aligned}$$

With Meijer and Hutton's approach, however, in order to lift an algebra $\phi :: f\ b\ a \to a$ to a catamorphism, we also need to supply the *inverse coalgebra* $\psi :: b \to f\ b\ a$. That is, in order to write a pretty printer we must supply a parser, which is not feasible—or perhaps even possible—in practice.

Fortunately, Fegaras and Sheard [8] realised that if the embedded functions within terms are *parametric*, then the inverse coalgebra is only used in order to *undo* computations performed by the algebra, since parametric functions can only "push around their arguments" without examining them. The solution proposed by Fegaras and Sheard is to add a *placeholder* to the structure of terms, which acts as a right-inverse of the catamorphism:[2]

$$\textbf{data}\ \mathit{Term}_{FS}\ f\ a = \mathit{In}_{FS}\ (f\ (\mathit{Term}_{FS}\ f\ a)\ (\mathit{Term}_{FS}\ f\ a)) \mid \mathit{Place}\ a$$

[2] Actually, Fegaras and Sheard do not use difunctors, but the given definition corresponds to their encoding.

$$
\begin{aligned}
&cata_{FS} :: Difunctor\ f \Rightarrow (f\ a\ a \to a) \to Term_{FS}\ f\ a \to a \\
&cata_{FS}\ \phi\ (In_{FS}\ t)\ = \phi\ (dimap\ Place\ (cata_{FS}\ \phi)\ t) \\
&cata_{FS}\ \phi\ (Place\ x) = x
\end{aligned}
$$

We can then define e.g. a signature for lambda terms, and a function that calculates the number of bound variables occurring in a term, as follows (the example is adopted from Washburn and Weirich [25]):

$$
\begin{aligned}
&\textbf{data}\ T\ a\ b = Lam\ (a \to b) \mid App\ b\ b \\
&\quad \text{-- } T \text{ is a difunctor, we omit the instance declaration} \\
&\phi :: T\ Int\ Int \to Int \\
&\phi\ (Lam\ f)\ \ = f\ 1 \\
&\phi\ (App\ x\ y) = x + y \\
&countVar :: Term_{FS}\ T\ Int \to Int \\
&countVar = cata_{FS}\ \phi
\end{aligned}
$$

In the $Term_{FS}$ encoding above, however, parametricity of the embedded functions is not guaranteed. More specifically, the type allows for three kinds of *exotic terms* [25], i.e. values in the meta language that do not correspond to terms in the object language:

$$
\begin{aligned}
&badPlace :: Term_{FS}\ T\ Bool \\
&badPlace = In_{FS}\ (Place\ True) \\
&badCata :: Term_{FS}\ T\ Int \\
&badCata = In_{FS}\ (Lam\ (\lambda x \to \textbf{if}\ countVar\ x \equiv 0\ \textbf{then}\ x\ \textbf{else}\ Place\ 0)) \\
&badCase :: Term_{FS}\ T\ a \\
&badCase = In_{FS}\ (Lam\ (\lambda x \to \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Term_{FS}\ (App\ \_\ \_) \to Term_{FS}\ (App\ x\ x) \\
&\qquad\qquad\qquad\qquad \_ \qquad\qquad\qquad\ \to x))
\end{aligned}
$$

Fegaras and Sheard showed how to avoid exotic terms by means of a custom type system. Washburn and Weirich [25] later showed that exotic terms can be avoided in a Haskell encoding via type parametricity and an abstract type of terms: terms are restricted to the type $\forall\ a\ .\ Term_{FS}\ f\ a$, and the constructors of $Term_{FS}$ are hidden. Parametricity rules out *badPlace* and *badCata*, while the use of an abstract type rules out *badCase*.

3.2 Parametric Higher-Order Abstract Syntax

While the approach of Washburn and Weirich effectively rules out exotic terms in Haskell, we prefer a different encoding that relies on type parametricity only, and not an abstract type of terms. Our solution is inspired by Chlipala's *parametric higher-order abstract syntax* (PHOAS) [6]. PHOAS is similar to the restricted form of HOAS that we saw above; however, Chlipala makes the parametricity explicit in the definition of terms by distinguishing between the type of bound variables and the type of recursive terms. In Chlipala's approach, an algebraic data type encoding of lambda terms *LTerm* can effectively be defined via an auxiliary data type *LTrm* of "preterms" as follows:

```
type LTerm  = ∀ a . LTrm a
data LTrm a = Lam (a → LTrm a) | Var a | App (LTrm a) (LTrm a)
```

The definition of *LTerm* guarantees that all functions embedded via *Lam* are parametric, and likewise that *Var*—Fegaras and Sheard's *Place*—can only be applied to variables bound by an embedded function. Atkey [2] showed that the encoding above adequately captures closed lambda terms modulo α-equivalence, assuming that there is no infinite data and that all embedded functions are total.

3.2.1 Parametric Terms

In order to transfer Chlipala's idea to non-recursive signatures and catamorphisms, we need to distinguish between covariant and contravariant uses of the recursive parameter. But this is exactly what difunctors do! We therefore arrive at the following definition of terms over difunctors:

```
newtype Term f = Term { unTerm :: ∀ a . Trm f a }
data Trm f a   = In (f a (Trm f a)) | Var a   -- "preterm"
```

Note the difference in *Trm* compared to $Term_{FS}$ (besides using the name *Var* rather than *Place*): the contravariant argument to the difunctor f is not the type of terms *Trm f a*, but rather a parametrised type a, which we quantify over at top-level to ensure parametricity. Hence, the only way to use a bound variable is to wrap it in a *Var* constructor—it is not possible to inspect the parameter. This representation more faithfully captures—we believe—the restricted form of HOAS than the representation of Washburn and Weirich: in our encoding it is explicit that bound variables are merely placeholders, and not the same as terms. Moreover, in some cases we actually *need* to inspect the structure of terms in order to define term transformations—we will see such an example in Section 3.2.3. With an abstract type of terms, this is not possible as Washburn and Weirich note [25].

Before we define algebras and catamorphisms, we lift the ideas underlying CDTs to *parametric compositional data types* (PCDTs), namely coproducts and implicit injections. Fortunately, the constructions of Section 2 are straightforwardly generalised (the instance declarations for $:\prec:$ are exactly as in *data types à la carte* [22], so we omit them here):

```
data (f :+: g) a b = Inl (f a b) | Inr (g a b)
instance (Difunctor f, Difunctor g) ⇒ Difunctor (f :+: g) where
  dimap f g (Inl x) = Inl (dimap f g x)
  dimap f g (Inr x) = Inr (dimap f g x)
class sub :≺: sup where
  inj  :: sub a b → sup a b
  proj :: sup a b → Maybe (sub a b)
inject :: (g :≺: f) ⇒ g a (Trm f a) → Trm f a
inject = In . inj
project :: (g :≺: f) ⇒ Trm f a → Maybe (g a (Trm f a))
```

```
project (Term t) =  proj t
project (Var _)  =  Nothing
```

We can then recast our previous signatures from Section 2.1 as difunctors:

```
data Lam a b = Lam (a → b)        data Plus a b = Plus b b
data App a b = App b b            data Let a b  = Let b (a → b)
data Lit a b = Lit Int            data Err a b = Err
type Sig     = Lam :+: App :+: Lit :+: Plus :+: Err :+: Let
type Sig'    = Lam :+: App :+: Lit :+: Plus :+: Err
```

Finally, we can automatically derive instance declarations for *Difunctor* as well as smart constructor definitions that comprise the injections as for CDTs [4]. However, in order to avoid the explicit *Var* constructor, we insert *dimap Var id* into the declarations, e.g.

```
iLam :: (Lam :≺: f) ⇒ (Trm f a → Trm f a) → Trm f a
iLam f = inject (dimap Var id (Lam f))   -- (= inject (Lam (f . Var)))
```

Using *iLam* we then need to be aware, though, that even if it takes a function *Trm f a → Trm f a* as argument, the input to that function will always be of the form *Var x by construction*. We can now again represent terms such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ compactly as follows:

```
e :: Term Sig
e = Term (iLet (iLit 2) (λx → (iLam (λy → y `iPlus` x) `iApp` iLit 3)))
```

3.2.2 Algebras and Catamorphisms

Given the representation of terms as fixed points of difunctors, we can now define algebras and catamorphisms:

```
type Alg f a = f a a → a
cata :: Difunctor f ⇒ Alg f a → Term f → a
cata φ (Term t) = cat t
  where cat (In t)   = φ (fmap cat t)   -- recall: fmap = dimap id
        cat (Var x) = x
```

The definition of *cata* above is essentially the same as $cata_{FS}$. The only difference is that bound variables within terms are already wrapped in a *Var* constructor. Therefore, the contravariant argument to *dimap* is the identity function, and we consequently use the derived function *fmap* instead.

With these definitions in place, we can now recast the modular pretty printer from Section 2.1 to the new difunctor signatures. However, since we now use a higher-order encoding, we need to generate variable names for printing. We therefore arrive at the following definition (the example is adopted from Washburn and Weirich [25], but we use streams rather than lists to represent the sequence of available variable names):

```
data Stream a = Cons a (Stream a)
class Pretty f where
  φPretty :: Alg f (Stream String → String)
  -- instance declaration that lifts Pretty to coproducts omitted
pretty :: (Difunctor f, Pretty f) ⇒ Term f → String
pretty t = cata φPretty t (names 1)
  where names n = Cons ('x' : show n) (names (n + 1))
instance Pretty Lam where
  φPretty (Lam f) (Cons x xs) = "(\\" ++ x ++ ". " ++
                                f (const x) xs ++ ")"
instance Pretty App where
  φPretty (App e1 e2) xs = "(" ++ e1 xs ++ " " ++ e2 xs ++ ")"
instance Pretty Lit where
  φPretty (Lit n) _ = show n
instance Pretty Plus where
  φPretty (Plus e1 e2) xs = "(" ++ e1 xs ++ " + " ++ e2 xs ++ ")"
instance Pretty Let where
  φPretty (Let e1 e2) (Cons x xs) = "(let " ++ x ++ " = " ++ e1 xs ++
                                    " in " ++ e2 (const x) xs ++ ")"
instance Pretty Err where
  φPretty Err _ = "error"
```

With this implementation of *pretty* we then have that *pretty e* evaluates to the string `(let x1 = 2 in ((\x2. (x2 + x1)) 3))`.

3.2.3 Term Transformations

The pretty printer is an example of a modular computation over a PCDT. However, we also want to define computations over PCDTs that *construct* PCDTs, e.g. the desugaring transformation. That is, we want to construct functions of type $Term\ f \to Term\ g$, which means that we must construct functions of type $(\forall\ a\ .\ Trm\ f\ a) \to (\forall\ a\ .\ Trm\ g\ a)$. Following the approach of Section 3.2.2, we construct such functions by forming the catamorphisms of algebras of type $Alg\ f\ (\forall\ a\ .\ Trm\ g\ a)$, i.e. functions of type $f\ (\forall\ a\ .\ Trm\ g\ a)\ (\forall\ a\ .\ Trm\ g\ a) \to \forall\ a\ .\ Trm\ g\ a$. However, in order to avoid the nested quantifiers, we instead use *parametric term algebras* of type $\forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$. From such algebras we then obtain functions of the type $\forall\ a\ .\ (Trm\ f\ a \to Trm\ g\ a)$ as catamorphisms, which finally yield the desired functions of type $(\forall\ a\,.\,Trm\ f\ a) \to (\forall\ a\,.\,Trm\ g\ a)$. With these considerations in mind, we arrive at the following definition of the desugaring algebra type class:

```
class Desug f g where
  φDesug :: ∀ a . Alg f (Trm g a)   -- not Alg f (Term g) !
  -- instance declaration that lifts Desug to coproducts omitted
desug :: (Difunctor f, Desug f g) ⇒ Term f → Term g
desug t = Term (cata φDesug t)
```

The algebra type class above is a *multi-parameter type class*: it is parametrised both by the domain signature f and the codomain signature g. We do this in order to obtain a desugaring function that is also modular in the codomain, similar to the evaluation function for vanilla CDTs [4].

We can now define the instances of *Desug* for the six signatures in order to obtain the desugaring function. However, by utilising overlapping instances we can make do with just two instance declarations:

```
instance (Difunctor f, f :≺: g) ⇒ Desug f g where
  φDesug = inject . dimap Var id   -- default instance for core signatures
instance (App :≺: f, Lam :≺: f) ⇒ Desug Let f where
  φDesug (Let e1 e2) = iLam e2 `iApp` e1
```

Given a term $e :: \mathit{Term\ Sig}$, we then have that $\mathit{desug}\ e :: \mathit{Term\ Sig}'$, i.e. the type shows that indeed all syntactic sugar has been removed.

Whereas the desugaring transformation shows that we can construct PCDTs from PCDTs in a modular fashion, we did not make use of the fact that PCDTs can be inspected. That is, the desugaring transformation does not inspect the recursively computed values, cf. the instance declaration for *Let*. However, in order to implement the constant folding transformation, we actually need to inspect recursively computed PCDTs. We again utilise overlapping instances:

```
class Constf f g where
  φConstf :: ∀ a . Alg f (Trm g a)
  -- instance declaration that lifts Constf to coproducts omitted
constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata φConstf t)
instance (Difunctor f, f :≺: g) ⇒ Constf f g where
  φConstf = inject . dimap Var id   -- default instance
instance (Plus :≺: f, Lit :≺: f) ⇒ Constf Plus f where
  φConstf (Plus e1 e2) = case (project e1, project e2) of
                           (Just (Lit n), Just (Lit m)) → iLit (n + m)
                           _                            → e1 `iPlus` e2
```

Since we provide a default instance, we not only obtain constant folding for the core language, but also for the full language, i.e. *constf* has both the types $\mathit{Term\ Sig}' \to \mathit{Term\ Sig}'$ and $\mathit{Term\ Sig} \to \mathit{Term\ Sig}$.

4 Monadic Computations

In the last section we demonstrated how to extend CDTs with parametric higher-order abstract syntax, and how to perform modular, recursive computations over terms containing binders. In this section we investigate monadic computations over PCDTs.

4.1 Monadic Interpretation

While the previous examples of modular computations did not require effects, the call-by-value interpreter prompts the need for monadic computations: both

in order to handle errors as well as controlling the evaluation order. Ultimately, we want to obtain a function of the type $\mathit{Term}\ \mathit{Sig}' \to m\ (\mathit{Sem}\ m)$, where the semantic domain *Sem* is defined as follows (we use an ordinary algebraic data type for simplicity):

$$\textbf{data}\ \mathit{Sem}\ m = \mathit{Fun}\ (\mathit{Sem}\ m \to m\ (\mathit{Sem}\ m)) \mid \mathit{Int}\ \mathit{Int}$$

Note that the monad only occurs in the codomain of *Fun*—if we want call-by-name semantics rather than call-by-value semantics, we simply add m also to the domain.

We can now implement the modular call-by-value interpreter similar to the previous modular computations but using the monadic algebra carrier $m\ (\mathit{Sem}\ m)$ instead:

```
class Monad m ⇒ Eval m f where
  φEval :: Alg f (m (Sem m))
  -- instance declaration that lifts Eval to coproducts omitted
eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata φEval
instance Monad m ⇒ Eval m Lam where
  φEval (Lam f) = return (Fun (f . return))
instance MonadError String m ⇒ Eval m App where
  φEval (App mx my) = do x ← mx
                         case x of
                           Fun f → my >>= f
                           _     → throwError "stuck"
instance Monad m ⇒ Eval m Lit where
  φEval (Lit n) = return (Int n)
instance MonadError String m ⇒ Eval m Plus where
  φEval (Plus mx my) = do x ← mx
                          y ← my
                          case (x, y) of
                            (Int n, Int m) → return (Int (n + m))
                            _              → throwError "stuck"
instance MonadError String m ⇒ Eval m Err where
  φEval Err = throwError "error"
```

In order to indicate errors in the course of the evaluation, we require the monad to provide a method to throw an error. To this end, we use the type class *MonadError*. Note how the modular design allows us to require the stricter constraint *MonadError String* m only for the cases where it is needed. This modularity of effects will become quite useful when we will rule out `"stuck"` errors in Section 6.

With the interpreter definition above we have that *eval* (*desug* e) evaluates to the value *Right* (*Int* 5) as expected, where e is as of page 94 and m is the *Either String* monad. Moreover, we also have that $0{+}\mathbf{error}$ and $0{+}\lambda x.x$ evaluate to *Left* `"error"` and *Left* `"stuck"`, respectively.

4.2 Monadic Computations with Implicit Sequencing

In the example above we use a monadic algebra carrier for monadic computations. For vanilla CDTs [4], however, we have previously shown how to perform monadic computations with *implicit sequencing*, by utilising the standard type class *Traversable*[3]:

$$\begin{aligned}
&\textbf{type } \mathit{AlgM}\ m\ f\ a = f\ a \to m\ a\\
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Traversable}\ f \textbf{ where}\\
&\quad \mathit{sequence} :: \mathit{Monad}\ m \Rightarrow f\ (m\ a) \to m\ (f\ a)\\
&\mathit{cataM} :: (\mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \mathit{AlgM}\ m\ f\ a \to \mathit{Term}\ f \to m\ a\\
&\mathit{cataM}\ \phi = \phi \mathbin{<\!\!=\!\!<} \mathit{sequence} \mathbin{.} \mathit{fmap}\ (\mathit{cataM}\ \phi) \mathbin{.} \mathit{out}
\end{aligned}$$

AlgM m f a represents the type of monadic algebras [9] over f and m, with carrier a, which is different from *Alg f (m a)* since the monad only occurs in the codomain of the monadic algebra. *cataM* is obtained from *cata* in Section 2 by performing *sequence* after applying *fmap* and replacing function composition with monadic function composition $<\!\!=\!\!<$. That is, the recursion scheme takes care of sequencing the monadic subcomputations. Monadic algebras are useful for instance if we want to recursively project a term over a compound signature to a smaller signature:

$$\begin{aligned}
&\mathit{deepProject} :: (\mathit{Traversable}\ g, f :\prec g) \Rightarrow \mathit{Term}\ f \to \mathit{Maybe}\ (\mathit{Term}\ g)\\
&\mathit{deepProject} = \mathit{cataM}\ (\mathit{liftM}\ \mathit{In} \mathbin{.} \mathit{proj})
\end{aligned}$$

Moreover, in a call-by-value setting we may use a monadic algebra *Alg f m a* rather than an ordinary algebra with a monadic carrier *Alg f (m a)* in order to avoid the explicit sequencing of effects.

Turning back to parametric terms, we can apply the same idea to difunctors yielding the following definition of monadic algebras:

$$\textbf{type } \mathit{AlgM}\ m\ f\ a = f\ a\ a \to m\ a$$

Similarly, we can easily generalise *Traversable* and *cataM* to difunctors:

$$\begin{aligned}
&\textbf{class } \mathit{Difunctor}\ f \Rightarrow \mathit{Ditraversable}\ f \textbf{ where}\\
&\quad \mathit{disequence} :: \mathit{Monad}\ m \Rightarrow f\ a\ (m\ b) \to m\ (f\ a\ b)\\
&\mathit{cataM} :: (\mathit{Ditraversable}\ f, \mathit{Monad}\ m) \Rightarrow \mathit{AlgM}\ m\ f\ a \to \mathit{Term}\ f \to m\ a\\
&\mathit{cataM}\ \phi\ (\mathit{Term}\ t) = \mathit{cat}\ t\\
&\quad \textbf{where } \mathit{cat}\ (\mathit{In}\ t) \quad = \mathit{disequence}\ (\mathit{fmap}\ \mathit{cat}\ t) \gg\!= \phi\\
&\qquad\qquad \mathit{cat}\ (\mathit{Var}\ x) = \mathit{return}\ x
\end{aligned}$$

Unfortunately, *cataM* only works for difunctors that do not use the contravariant argument. To see why this is the case, reconsider the *Lam* constructor; in order to define an instance of *Ditraversable* for *Lam* we must write a function of the type:

$$\mathit{disequence} :: \mathit{Monad}\ m \Rightarrow \mathit{Lam}\ a\ (m\ b) \to m\ (\mathit{Lam}\ a\ b)$$

[3]We have omitted methods from the definition of *Traversable* that are not necessary for our purposes.

Since *Lam* is isomorphic to the function type constructor $\rightarrow$, this is equivalent to a function of the type:

$$\forall\ a\ b\ m\ .\ Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow m\ (a \rightarrow b)$$

We cannot hope to be able to construct a meaningful combinator of that type. Intuitively, in a function of type $a \rightarrow m\ b$, the monadic effect of the result can depend on the input of type a. The monadic effect of a monadic value of type $m\ (a \rightarrow b)$ is not dependent on such input. For example, think of a state transformer monad ST with state S and its put function $put :: S \rightarrow ST\ ()$. What would be the corresponding transformation to a monadic value of type $ST\ (S \rightarrow ())$?

Hence, *cataM* does not extend to terms with binders, but it still works for terms without binders as in vanilla CDTs [4]. In particular, we cannot use *cataM* to define the call-by-value interpreter from Section 4.1.

5 Contexts and Term Homomorphisms

While the generality of catamorphisms makes them a powerful tool for modular function definitions, their generality at the same time inhibits flexibility and reusability. However, the full generality of catamorphisms is not always needed in the case of term transformations, which we discussed in Section 3.2.3. To this end, we have previously studied term homomorphisms [4] as a restricted form of term algebras. In this section we redevelop term homomorphisms for PCDTs.

5.1 From Terms to Contexts and back

The crucial idea behind term homomorphisms is to generalise terms to *contexts*, i.e. terms with *holes*. Following previous work [4] we define the generalisation of terms with holes as a *generalised algebraic data type (GADT)* [18] with *phantom types Hole* and *NoHole*:

$$\begin{array}{l}\textbf{data}\ Cxt :: * \rightarrow (* \rightarrow * \rightarrow *) \rightarrow * \rightarrow * \rightarrow *\ \textbf{where}\\ \quad In \quad :: f\ a\ (Cxt\ h\ f\ a\ b) \rightarrow Cxt\ h \quad\ f\ a\ b\\ \quad Var\ :: a \qquad\qquad\qquad\ \rightarrow Cxt\ h \quad\ f\ a\ b\\ \quad Hole :: b \qquad\qquad\qquad\ \rightarrow Cxt\ Hole\ f\ a\ b\end{array}$$

$$\textbf{data}\ Hole$$

$$\textbf{data}\ NoHole$$

The first argument to *Cxt* is a phantom type indicating whether the term contains holes or not. A context can thus be defined as:

$$\textbf{type}\ Context = Cxt\ Hole$$

That is, contexts *may* contain holes. On the other hand, terms must not contain holes, so we can recover our previous definition of preterms *Trm* as follows:

$$\textbf{type}\ Trm\ f\ a = Cxt\ NoHole\ f\ a\ ()$$

The definition of *Term* remains unchanged. This representation of contexts and preterms allows us to uniformly define functions that work on both types. For example, the function *inject* now has the type:

$$inject :: (g :\prec f) \Rightarrow g\ a\ (Cxt\ h\ f\ a\ b) \to Cxt\ h\ f\ a\ b$$

5.2 Term Homomorphisms

In Section 3.2.3 we have shown that term transformations, i.e. functions of type $Term\ f \to Term\ g$, are obtained as catamorphisms of parametric term algebras of type $\forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$. Spelling out the definition of *Alg*, such algebras are functions of type:

$$\forall\ a\ .\ f\ (Trm\ g\ a)\ (Trm\ g\ a) \to Trm\ g\ a$$

As we have argued previously [4], the fact that the target signature g occurs in both the domain and codomain in the above type prevents us from making use of the structure of the algebra's carrier type $Trm\ g\ a$. In particular, the constructions that we show in Section 5.3 are not possible with the above type.

In order to circumvent this restriction, we remove the occurrences of the algebra's carrier type $Trm\ g\ a$ in the domain by replacing them with type variables:

$$\forall\ a\ b\ .\ f\ a\ b \to Trm\ g\ a$$

However, since we introduce a fresh variable b, functions of the above type are not able to use the corresponding parts of the argument for constructing the result. A value of type b cannot be injected into the type $Trm\ g\ a$.

This is where contexts come into the picture: we enable the use of values of type b in the result by replacing the codomain type $Trm\ g\ a$ with $Context\ g\ a\ b$. The result is the following type of *term homomorphisms*:

$$\textbf{type}\ Hom\ f\ g = \forall\ a\ b\ .\ f\ a\ b \to Context\ g\ a\ b$$

A function $\rho :: Hom\ f\ g$ is a transformation of constructors from f into a context over g, i.e. a term over g that may embed values taken from the arguments of the f-constructor. The parametric polymorphism of the type guarantees that the arguments of the f-constructor cannot be inspected but only embedded into the result context. In order to apply term homomorphisms to terms, we need an auxiliary function that merges nested contexts:

$$\begin{aligned}
&appCxt :: Difunctor\ f \Rightarrow Context\ f\ a\ (Cxt\ h\ f\ a\ b) \to Cxt\ h\ f\ a\ b\\
&appCxt\ (In\ t) \quad\ = In\ (fmap\ appCxt\ t)\\
&appCxt\ (Var\ x)\ = Var\ x\\
&appCxt\ (Hole\ h) = h
\end{aligned}$$

Given a context that has terms embedded in its holes, we obtain a term as a result; given a context with embedded contexts, the result is again a context.

Using the combinator above we can now apply a term homomorphism to a preterm—or more generally, to a context:

$$
\begin{aligned}
&\mathit{appHom} :: (\mathit{Difunctor}\ f, \mathit{Difunctor}\ g) \Rightarrow \\
&\qquad \mathit{Hom}\ f\ g \to \mathit{Cxt}\ h\ f\ a\ b \to \mathit{Cxt}\ h\ g\ a\ b \\
&\mathit{appHom}\ \rho\ (\mathit{In}\ t) \quad = \mathit{appCxt}\ (\rho\ (\mathit{fmap}\ (\mathit{appHom}\ \rho)\ t)) \\
&\mathit{appHom}\ \rho\ (\mathit{Var}\ x) \ = \mathit{Var}\ x \\
&\mathit{appHom}\ \rho\ (\mathit{Hole}\ h) = \mathit{Hole}\ h
\end{aligned}
$$

From *appHom* we can then obtain the actual transformation on terms as follows:

$$
\begin{aligned}
&\mathit{appTHom} :: (\mathit{Difunctor}\ f, \mathit{Difunctor}\ g) \Rightarrow \mathit{Hom}\ f\ g \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{appTHom}\ \rho\ (\mathit{Term}\ t) = \mathit{Term}\ (\mathit{appHom}\ \rho\ t)
\end{aligned}
$$

Before we describe the benefits of term homomorphisms over term algebras, we reconsider the desugaring transformation from Section 3.2.3, but as a term homomorphism rather than a term algebra:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Desug}\ f\ g\ \textbf{where} \\
&\quad \rho_{\mathrm{Desug}} :: \mathit{Hom}\ f\ g \\
&\quad \text{-- instance declaration that lifts } \mathit{Desug} \text{ to coproducts omitted} \\
&\mathit{desug} :: (\mathit{Difunctor}\ f, \mathit{Difunctor}\ g, \mathit{Desug}\ f\ g) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{desug} = \mathit{appTHom}\ \rho_{\mathrm{Desug}} \\
&\textbf{instance}\ (\mathit{Difunctor}\ f, \mathit{Difunctor}\ g, f :\prec g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
&\quad \rho_{\mathrm{Desug}} = \mathit{In}\ .\ \mathit{fmap}\ \mathit{Hole}\ .\ \mathit{inj} \quad \text{-- default instance for core signatures} \\
&\textbf{instance}\ (\mathit{App} :\prec f, \mathit{Lam} :\prec f) \Rightarrow \mathit{Desug}\ \mathit{Let}\ f\ \textbf{where} \\
&\quad \rho_{\mathrm{Desug}}\ (\mathit{Let}\ e_1\ e_2) = \mathit{inject}\ (\mathit{Lam}\ (\mathit{Hole}\ .\ e_2))\ `\mathit{iApp}`\ \mathit{Hole}\ e_1
\end{aligned}
$$

Note how, in the instance declaration for *Let*, the constructor *Hole* is used to embed arguments of the constructor *Let*, viz. e_1 and e_2, into the context that is constructed as the result.

As for the desugaring function in Section 3.2.3, we utilise overlapping instances to provide a default translation for the signatures that need not be translated. The definitions above yield the desired desugaring function $\mathit{desug} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Term}\ \mathit{Sig}'$.

5.3 Transforming and Combining Term Homomorphisms

In the following we shall shortly describe what we actually gain by adopting the term homomorphism approach. First, term homomorphisms enable automatic propagation of annotations, where annotations are added via a restricted difunctor product, namely a product of a difunctor f and a constant c:

$$
\textbf{data}\ (f :\&: c)\ a\ b = f\ a\ b :\&: c
$$

For instance, the type of ASTs of our language where each node is annotated with source positions is captured by the type $\mathit{Term}\ (\mathit{Sig} :\&: \mathit{SrcPos})$. With a term homomorphism $\mathit{Hom}\ f\ g$ we automatically get a lifted version $\mathit{Hom}\ (f :\&: c)\ (g :\&: c)$, which propagates annotations from the input to the output. Hence, from our desugaring function in the previous section we automatically get a lifted function on parse trees $\mathit{Term}\ (\mathit{Sig} :\&: \mathit{SrcPos}) \to \mathit{Term}\ (\mathit{Sig}' :\&: \mathit{SrcPos})$, which propagates source positions from the syntactic sugar to the core constructs. We

omit the details here, but note that the constructions for CDTs [4] carry over straightforwardly to PCDTs.

The second motivation for introducing term homomorphisms is deforestation [23]. As we have shown previously [4], it is not possible to fuse two term algebras in order to traverse the term only once. That is, we do not find a composition operator $\circledcirc$ on algebras that satisfies the following equation for all $\phi_1 :: Alg\ g\ a$ and $\phi_2 :: \forall\ a\ .\ Alg\ f\ (Trm\ g\ a)$:

$$cata\ \phi_1\ .\ cata\ \phi_2 = cata\ (\phi_1 \circledcirc \phi_2)$$

With term homomorphism, however, we do have such a composition operator $\circledcirc$:

$$\begin{aligned}&(\circledcirc) :: (Difunctor\ g, Difunctor\ h) \Rightarrow Hom\ g\ h \to Hom\ f\ g \to Hom\ f\ h\\ &\rho_1 \circledcirc \rho_2 = appHom\ \rho_1\ .\ \rho_2\end{aligned}$$

For this composition, we then obtain the desired equation for all $\rho_1 :: Hom\ g\ h$ and $\rho_2 :: Hom\ f\ g$:

$$appHom\ \rho_1\ .\ appHom\ \rho_2 = appHom\ (\rho_1 \circledcirc \rho_2)$$

In fact, we can also compose an arbitrary algebra with a term homomorphism:

$$\begin{aligned}&(\boxdot) :: Difunctor\ g \Rightarrow Alg\ g\ a \to Hom\ f\ g \to Alg\ f\ a\\ &\phi \boxdot \rho = free\ \phi\ id\ .\ \rho\end{aligned}$$

where

$$\begin{aligned}&free :: Difunctor\ f \Rightarrow Alg\ f\ a \to (b \to a) \to Cxt\ h\ f\ a\ b \to a\\ &free\ \phi\ f\ (In\ t) \quad = \phi\ (fmap\ (free\ \phi\ f)\ t)\\ &free\ \_\ \_\ (Var\ x) \ = x\\ &free\ \_\ f\ (Hole\ h) = f\ h\end{aligned}$$

The composition of algebras and homomorphisms satisfies the following equation:

$$cata\ \phi\ .\ appHom\ \rho = cata\ (\phi \boxdot \rho) \quad \text{for all } \phi :: Alg\ g\ a \text{ and } \rho :: Hom\ f\ g$$

For example, in order to evaluate a term with syntactic sugar, rather than composing *eval* and *desug*, we can use the function $cata\ (\phi_{\mathrm{Eval}} \boxdot \rho_{\mathrm{Desug}})$, which only traverses the term once. This transformation can be automated using GHC's rewrite mechanism [11] and our experimental results for CDTs show that the thus obtained speedup is significant [4].

6 Generalised Parametric Compositional Data Types

In this section we briefly describe how to lift the construction of mutually recursive data types and—more generally—GADTs from CDTs to PCDTs. The construction is based on the work of Johann and Ghani [10]. For CDTs the generalisation, roughly speaking, amounts to lifting functors to (generalised) *higher-order functors* [10], and functions on terms to *natural transformations*, as shown earlier [4]:

$$\textbf{type}\ a \mathrel{\dot{\to}} b = \forall\ i\ .\ a\ i \to b\ i$$

$$\textbf{class}\ \mathit{HFunctor}\ f\ \textbf{where}$$
$$\quad \mathit{hfmap} :: a \mathrel{\dot{\to}} b \to f\ a \mathrel{\dot{\to}} f\ b$$

Now, signatures are of the kind $(* \to *) \to * \to *$, rather than $* \to *$, which reflects the fact that signatures are now *indexed types*, and so are terms (or contexts in general). Consequently, the carrier of an algebra is a type constructor of kind $* \to *$:

$$\textbf{type}\ \mathit{Alg}\ f\ a = f\ a \mathrel{\dot{\to}} a$$

Since signatures will be defined as GADTs, we effectively deal with *many-sorted algebras*. If a subterm has the type index i, then the value computed recursively by a catamorphism will have the type $a\ i$. The coproduct $:+:$ and the automatic injections $:\prec:$ carry over straightforwardly from functors to higher-order functors [4].

In order to lift the ideas from CDTs to PCDTs, we need to consider indexed difunctors. This prompts the notion of *higher-order difunctors*:

$$\textbf{class}\ \mathit{HDifunctor}\ f\ \textbf{where}$$
$$\quad \mathit{hdimap} :: (a \mathrel{\dot{\to}} b) \to (c \mathrel{\dot{\to}} d) \to f\ b\ c \mathrel{\dot{\to}} f\ a\ d$$
$$\textbf{instance}\ \mathit{HDifunctor}\ f \Rightarrow \mathit{HFunctor}\ (f\ a)\ \textbf{where}$$
$$\quad \mathit{hfmap} = \mathit{hdimap}\ \mathit{id}$$

Note the familiar pattern from ordinary PCDTs: a higher-order difunctor gives rise to a higher-order functor when the contravariant argument is fixed.

To illustrate higher-order difunctors, consider a modular GADT encoding of our core language:

$$\textbf{data}\ \mathit{TArrow}\ i\ j$$
$$\textbf{data}\ \mathit{TInt}$$
$$\textbf{data}\ \mathit{Lam} :: (* \to *) \to (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathit{Lam} :: (a\ i \to b\ j) \to \mathit{Lam}\ a\ b\ (i\ `\mathit{TArrow}`\ j)$$
$$\textbf{data}\ \mathit{App} :: (* \to *) \to (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathit{App} :: b\ (i\ `\mathit{TArrow}`\ j) \to b\ i \to \mathit{App}\ a\ b\ j$$
$$\textbf{data}\ \mathit{Lit} :: (* \to *) \to (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathit{Lit} :: \mathit{Int} \to \mathit{Lit}\ a\ b\ \mathit{TInt}$$
$$\textbf{data}\ \mathit{Plus} :: (* \to *) \to (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathit{Plus} :: b\ \mathit{TInt} \to b\ \mathit{TInt} \to \mathit{Plus}\ a\ b\ \mathit{TInt}$$
$$\textbf{data}\ \mathit{Err} :: (* \to *) \to (* \to *) \to * \to *\ \textbf{where}$$
$$\quad \mathit{Err} :: \mathit{Err}\ a\ b\ i$$
$$\textbf{type}\ \mathit{Sig}' = \mathit{Lam} :+: \mathit{App} :+: \mathit{Lit} :+: \mathit{Plus} :+: \mathit{Err}$$

Note, in particular, the type of *Lam*: now the bound variable is typed!

We use *TArrow* and *TInt* as type indices for the GADT definitions above. The preference of these fresh types over Haskell's $\to$ and *Int* is meant to emphasise that these phantom types are only labels that represent the type constructors of our object language.

We use the coproduct $:+:$ of higher-order difunctors above to combine signatures, which is easily defined, and as for CDTs it is straightforward to lift instances of *HDifunctor* for f and g to an instance for $f :+: g$. Similarly, we can generalise the relation $:\prec:$ from difunctors to higher-order difunctors, so we omit its definition here.

The type of generalised parametric (pre)terms can now be constructed as an indexed type:

```
newtype Term f i = Term {unTerm :: ∀ a . Trm f a i}
data Trm f a i   = In (f a (Trm f a) i) | Var (a i)
```

Moreover, we use smart constructors as for PCDTs to compactly construct terms, for instance:

```
e :: Term Sig' TInt
e = Term (iLam (λx → x `iPlus` x) `iApp` iLit 2)
```

Finally, we can lift algebras and their induced catamorphisms by lifting the definitions in Section 3.2.2 via natural transformations and higher-order difunctors:

```
type Alg f a = f a a :-> a
cata :: HDifunctor f ⇒ Alg f a → Term f :-> a
cata φ (Term t) = cat t
  where cat (In t)  = φ (hfmap cat t)   -- recall: hfmap = hdimap id
        cat (Var x) = x
```

With the definitions above we can now define a call-by-value interpreter for our typed example language. To this end, we must provide a type-level function that, for a given object language type constructed from *TArrow* and *TInt*, selects the corresponding subset of the semantic domain *Sem m* from Section 4.1. This can be achieved via Haskell's *type families* [17]:

```
type family Sem (m :: * → *) i
type instance Sem m (i `TArrow` j) = Sem m i → m (Sem m j)
type instance Sem m TInt           = Int
```

The type *Sem m t* is obtained from an object language type t by replacing each function type t_1 `` `TArrow` `` t_2 occurring in t with *Sem m* $t_1 \to$ *m* (*Sem m* t_2) and each *TInt* with *Int*.

To make *Sem* into a proper type—as opposed to a mere type synonym—and simultaneously add the monad m at the top level, we define a **newtype** M:

```
newtype M m i = M {unM :: m (Sem m i)}
class Monad m ⇒ Eval m f where
  φEval :: f (M m) (M m) i → m (Sem m i)
    -- M . φEval :: Alg f (M m) is the actual algebra
eval :: (Monad m, HDifunctor f, Eval m f) ⇒ Term f i → m (Sem m i)
eval = unM . cata (M . φEval)
```

We can then provide the instance declarations for the signatures of the core language, and effectively obtain a tagless, modular, and extendable monadic interpreter:

```
instance Monad m ⇒ Eval m Lam where
  φEval (Lam f) = return (unM . f . M . return)
instance Monad m ⇒ Eval m App where
  φEval (App (M mf) (M mx)) = do f ← mf
                                 mx >>= f
instance Monad m ⇒ Eval m Lit where
  φEval (Lit n) = return n
instance Monad m ⇒ Eval m Plus where
  φEval (Plus (M mx) (M my)) = do x ← mx
                                  y ← my
                                  return (x + y)
instance MonadError String m ⇒ Eval m Err where
  φEval Err = throwError "error"
```

With the above definition of *eval* we have, for instance, that the expression *eval e*:: *Either String Int* evaluates to the value *Right* 4. Due to the fact that we now have a typed language, the *Err* constructor is the only source of an erroneous computation—the interpreter cannot get stuck. Moreover, since the modular specification of the interpreter only enforces the constraint *MonadError String m* for the signature *Err*, the term *e* can in fact be interpreted in the identity monad, rather than the *Either String* monad, as it does not contain **error**. Consequently, we know statically that the evaluation of *e* cannot fail!

Note that computations over generalised PCDTs are not limited to the tagless approach that we have illustrated above. We could have easily reformulated the semantic domain *Sem m* from Section 4.1 as a GADT to use it as the carrier of a many-sorted algebra. Other natural carriers for many-sorted algebras are the type families of terms *Term f*, of course.

Other concepts that we have introduced for vanilla PCDTs before can be transferred straightforwardly to generalised PCDTs in the same fashion. This includes contexts and term homomorphisms.

7 Practical Considerations

The motivation for introducing CDTs was to make Swierstra's *data types à la carte* [22] readily useful in practice. Besides extending *data types à la carte* with various aspects, such as monadic computations and term homomorphisms, the CDTs library provides all the generic functionality as well as automatic derivation of boilerplate code. With (generalised) PCDTs we have followed that path. Our library provides Template Haskell [20] code to automatically derive instances of the required type classes, such as *Difunctor* and *Ditraversable*, as well as smart constructors and lifting of algebra type classes to coproducts. Moreover, our library supports automatic derivation of standard type classes *Show*, *Eq*, and *Ord* for terms, similar to Haskell's **deriving** mechanism. We show how to derive

instances of *Eq* in the following subsection. *Ord* follows in the same fashion, and *Show* follows an approach similar to the pretty printer in Section 3.2.2, but using the monad *FreshM* that is also used to determine equality, as we shall see below.

Figure 1 provides the complete source code needed to implement our example language from Section 2.1. Note that we have derived *Show*, *Eq*, and *Ord* instances for terms of the language—in particular the term e is printed as `Let (Lit 2) (\a -> App (Lam (\b -> Plus b a)) (Lit 3))`.

7.1 Equality

A common pattern when programming in Haskell is to derive instances of the type class *Eq*, for instance in order to test the desugaring transformation in Section 3.2.3. While the use of PHOAS ensures that all functions are invariant under α-renaming, we still have to devise an algorithm that decides α-equivalence. To this end, we will turn the rather elusive representation of bound variables via functions into a concrete form.

In order to obtain concrete representations of bound variables, we provide a method for generating fresh variable names. This is achieved via a monad *FreshM* offering the following operations:

$$\begin{aligned}&\mathit{withName} :: (\mathit{Name} \to \mathit{FreshM}\ a) \to \mathit{FreshM}\ a\\&\mathit{evalFreshM} :: \mathit{FreshM}\ a \to a\end{aligned}$$

FreshM is an abstraction of an infinite sequence of fresh names. The function *withName* provides a fresh name. Names are represented by the abstract type *Name*, which implements instances of *Show*, *Eq*, and *Ord*.

We first introduce a variant of the type class *Eq* that uses the *FreshM* monad:

$$\begin{aligned}&\textbf{class}\ \mathit{PEq}\ a\ \textbf{where}\\&\quad \mathit{peq} :: a \to a \to \mathit{FreshM}\ \mathit{Bool}\end{aligned}$$

This type class is used to define the type class *EqD* of equatable difunctors, which lifts to coproducts:

$$\begin{aligned}&\textbf{class}\ \mathit{EqD}\ f\ \textbf{where}\\&\quad \mathit{eqD} :: \mathit{PEq}\ a \Rightarrow f\ \mathit{Name}\ a \to f\ \mathit{Name}\ a \to \mathit{FreshM}\ \mathit{Bool}\\&\textbf{instance}\ (\mathit{EqD}\ f, \mathit{EqD}\ g) \Rightarrow \mathit{EqD}\ (f :+: g)\ \textbf{where}\\&\quad \mathit{eqD}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ y) = x\ `\mathit{eqD}`\ y\\&\quad \mathit{eqD}\ (\mathit{Inr}\ x)\ (\mathit{Inr}\ y) = x\ `\mathit{eqD}`\ y\\&\quad \mathit{eqD}\ \_\ \ \ \_\ \ \ = \mathit{return}\ \mathit{False}\end{aligned}$$

We then obtain equality of terms as follows (we do not consider contexts here for simplicity):

$$\begin{aligned}&\textbf{instance}\ \mathit{EqD}\ f \Rightarrow \mathit{PEq}\ (\mathit{Trm}\ f\ \mathit{Name})\ \textbf{where}\\&\quad \mathit{peq}\ (\mathit{In}\ t_1)\ \ (\mathit{In}\ t_2)\ \ = t_1\ `\mathit{eqD}`\ t_2\\&\quad \mathit{peq}\ (\mathit{Var}\ x_1)\ (\mathit{Var}\ x_2) = \mathit{return}\ (x_1 \equiv x_2)\\&\quad \mathit{peq}\ \_\ \ \ \_\ \ \ = \mathit{return}\ \mathit{False}\end{aligned}$$

```
instance (Difunctor f, EqD f) ⇒ Eq (Term f) where
  (≡) (Term x) (Term y) = evalFreshM ((x :: Trm f Name) `peq` y)
```

Note that we need to explicitly instantiate the parametric type in x to *Name* in the last instance declaration, in order to trigger the instance for *Trm f Name* defined above.

Equality of terms, i.e. α-equivalence, has thus been reduced to providing instances of *EqD* for the difunctors comprising the signature of the term, which for *Lam* can be defined as follows:

```
instance EqD Lam where
  eqD (Lam f) (Lam g) = withName (λx → f x `peq` g x)
```

That is, f and g are considered equal if they are equal when applied to the same fresh name x.

8 Discussion and Related Work

Implementing languages with binders can be a difficult task. Using explicit variable names, we have to be careful in order to make sure that functions on ASTs are invariant under α-renaming. HOAS [15] is one way of tackling this problem, by reusing the binding mechanisms of the implementation language to define those of the object language. The challenge with HOAS, however, is that it is difficult to perform recursive computations over ASTs with binders [8, 13, 19, 25]. Besides what is documented in this paper, we have also lifted (generalised) parametric compositional data types to other (co)recursion schemes, such as anamorphisms and histomorphisms. Moreover, term homomorphisms can be straightforwardly extended with a state space: depending on how the state is propagated, this yields bottom-up resp. top-down tree transducers [7].

Our approach of using PHOAS [6] amounts to the same restriction on embedded functions as Fegeras and Sheard [8], and Washburn and Weirich [25]. However, unlike Washburn and Weirich's Haskell implementation, our approach does not rely on making the type of terms abstract. Not only is it interesting to see that we can do without type abstraction, in fact, we sometimes need to inspect terms in order to write functions that produce terms, such as our constant folding algorithm. With Washburn and Weirich's encoding this is not possible.

Ahn and Sheard [1] recently showed how to generalise the recursion schemes of Washburn and Weirich to Mendler-style recursion schemes, using the same representation for terms as Washburn and Weirich. Hence their approach also suffers from the inability to inspect terms. Although we could easily adopt Mendler-style recursion schemes in our setting, their generality does not make a difference in a non-strict language such as Haskell. Additionally, Ahn and Sheard pose the open question whether there is a safe (i.e., terminating) way to apply histomorphisms to terms with negative recursive occurrences: although we have not investigated termination properties of our histomorphisms, we conjecture that the use of our parametric terms—which are purely inductive—may provide one solution.

The *finally tagless* approach of Carette et al. [5] has been proposed as an alternative solution to the expression problem [24]. While the approach is very

simple and elegant, and also supports (typed) higher-order encodings, the approach falls short when we want to define recursive, modular computations that construct modular terms too. Atkey et al. [3], for instance, use the finally tagless approach to build a modular interpreter. However, the interpreter cannot be made modular in the return type, i.e. the language defining values. Hence, when Atkey et al. extend their expression language they need to also change the data type that represents values, which means that the approach is not fully modular. Although our interpreter in Section 4.1 also uses a fixed domain of values *Sem*, we can make the interpreter fully modular by also using a PCDT for the return type, and using a multi-parameter type class definition similar to the desugaring transformation in Section 3.2.3.

Nominal sets [16] is another approach for dealing with binders, in which variables are explicit, but recursively defined functions are guaranteed to be invariant with respect to α-equivalence of terms. Implementations of this approach, however, require extensions of the metalanguage [21], and the approach is therefore not immediately usable in Haskell.

Acknowledgement

The authors wish to thank Andrzej Filinski for his insightful comments on an earlier version of this paper.

Bibliography

[1] K. Y. Ahn and T. Sheard. A hierarchy of Mendler style recursion combinators: taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 234–246, New York, NY, USA, 2011. ACM. doi: 10.1145/2034773.2034807.

[2] R. Atkey. Syntax for Free: Representing Syntax with Binding Using Parametricity. In P.-L. Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-02273-9˙5.

[3] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 37–48, New York, NY, USA, 2009. ACM. doi: 10.1145/1596638.1596644.

[4] P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, New York, NY, USA, 2011. ACM. doi: 10.1145/2036918.2036930.

[5] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009. doi: 10.1017/S0956796809007205.

[6] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 143–156, New York, NY, USA, 2008. ACM. doi: 10.1145/1411204.1411226.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on `http://www.grappa.univ-lille3.fr/tata`, 2008.

[8] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, New York, NY, USA, 1996. ACM. doi: 10.1145/237721.237792.

[9] M. M. Fokkinga. A Gentle Introduction to Category Theory: the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, 1992.

[10] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *POPL '08*, pages 297–308, New York, New York, USA, 2008. ACM Press. doi: 10.1145/1328438.1328475.

[11] S. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, page 203, 2001.

[12] S. Marlow. Haskell 2010 Language Report, 2010.

[13] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the seventh International Conference on Functional Programming languages and computer architecture*, pages 324–333, New York, NY, USA, 1995. ACM. doi: 10.1145/224164.224225.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961˙7.

[15] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM. doi: 10.1145/53990.54010.

[16] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006. doi: 10.1145/1147954.1147961.

[17] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62, New York, NY, USA, 2008. ACM. doi: 10.1145/1411204.1411215.

[18] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596599.

[19] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00418-7.

[20] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, volume 37 of *SIGPLAN Notices*, pages 60–75, New York, NY, USA, 2002. ACM. doi: 10.1145/636517.636528.

[21] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, New York, NY, USA, 2003. ACM. doi: 10.1145/944705.944729.

[22] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758.

[23] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990. doi: 10.1016/0304-3975(90)90147-A.

[24] P. Wadler. The Expression Problem. Available on `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[25] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(1):87–140, 2008. doi: 10.1017/S0956796807006557.

```
import Data.Comp.Param
import Data.Comp.Param.Show ()
import Data.Comp.Param.Equality ()
import Data.Comp.Param.Ordering ()
import Data.Comp.Param.Derive
import Control.Monad.Error (MonadError, throwError)

data Lam a b  = Lam (a → b)
data App a b  = App b b
data Lit a b  = Lit Int
data Plus a b = Plus b b
data Let a b  = Let b (a → b)
data Err a b  = Err

$(derive [smartConstructors, makeDifunctor, makeShowD, makeEqD, makeOrdD]
         [''Lam, ''App, ''Lit, ''Plus, ''Let, ''Err])

e :: Term (Lam :+: App :+: Lit :+: Plus :+: Let :+: Err)
e = Term (iLet (iLit 2) (λx → (iLam (λy → y `iPlus` x) `iApp` iLit 3)))

-- * Desugaring
class Desug f g where
  desugHom :: Hom f g

$(derive [liftSum] [''Desug]) -- lift Desug to coproducts

desug :: (Difunctor f, Difunctor g, Desug f g) ⇒ Term f → Term g
desug (Term t) = Term (appHom desugHom t)

instance (Difunctor f, Difunctor g, f :<: g) ⇒ Desug f g where
  desugHom = In . fmap Hole . inj -- default instance for core signatures

instance (App :<: f, Lam :<: f) ⇒ Desug Let f where
  desugHom (Let e1 e2) = inject (Lam (Hole . e2)) `iApp` Hole e1

-- * Constant folding
class Constf f g where
  constfAlg :: forall a. Alg f (Trm g a)

$(derive [liftSum] [''Constf]) -- lift Constf to coproducts

constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata constfAlg t)

instance (Difunctor f, f :<: g) ⇒ Constf f g where
  constfAlg = inject . dimap Var id -- default instance

instance (Plus :<: f, Lit :<: f) ⇒ Constf Plus f where
  constfAlg (Plus e1 e2) = case (project e1, project e2) of
                             (Just (Lit n),Just (Lit m)) → iLit (n + m)
                             _                           → e1 `iPlus` e2

-- * Call-by-value evaluation
data Sem m = Fun (Sem m → m (Sem m)) | Int Int

class Monad m ⇒ Eval m f where
  evalAlg :: Alg f (m (Sem m))

$(derive [liftSum] [''Eval]) -- lift Eval to coproducts

eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata evalAlg

instance Monad m ⇒ Eval m Lam where
  evalAlg (Lam f) = return (Fun (f . return))

instance MonadError String m ⇒ Eval m App where
  evalAlg (App mx my) = do x ← mx
                           case x of Fun f → my >>= f
                                     _     → throwError "stuck"

instance Monad m ⇒ Eval m Lit where
  evalAlg (Lit n) = return (Int n)

instance MonadError String m ⇒ Eval m Plus where
  evalAlg (Plus mx my) = do x ← mx
                            y ← my
                            case (x,y) of (Int n,Int m) → return (Int (n + m))
                                          _             → throwError "stuck"

instance MonadError String m ⇒ Eval m Err where
  evalAlg Err = throwError "error"
```

Figure 1: Complete example using the parametric compositional data types library.

Modular Tree Automata

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

Tree automata are traditionally used to study properties of tree languages and tree transformations. In this paper, we consider tree automata as the basis for modular and extensible recursion schemes. We show, using well-known techniques, how to derive from standard tree automata highly modular recursion schemes. Functions that are defined in terms of these recursion schemes can be combined, reused and transformed in many ways. This flexibility facilitates the specification of complex transformations in a concise manner, which is illustrated with a number of examples.

Contents

1 Introduction

Functional programming languages are an excellent tool for specifying abstract syntax trees (ASTs) and defining syntax-directed transformations on them: algebraic data types provide a compact notation for both defining types of ASTs as well as constructing and manipulating ASTs. As a complement to that, recursively defined functions on algebraic data types allow us to traverse ASTs defined by algebraic data types.

For example, writing an evaluation function for a small expression language is easily achieved in Haskell [19] as follows:

$$\begin{aligned}
&\textbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp}\\
&\mathit{eval} :: \mathit{Exp} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ i) \quad\ = i\\
&\mathit{eval}\ (\mathit{Plus}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Unfortunately, this simple approach does not scale very well. As soon as we have to implement more complex transformations that work on more than just a few types of ASTs, simple recursive function definitions become too inflexible and complicated.

Specifying and implementing such transformations is an everyday issue for compiler construction and thus has prompted a lot of research in this area. One notable approach to address both sides is the use of attribute grammars [15, 22]. These systems facilitate compact specification and efficient implementation of syntax-directed transformations.

In this paper, we take a different but not unrelated approach. We still want to implement the transformations in a functional language. But instead of writing transformation functions as general recursive functions as the one above, our goal is to devise *recursion schemes*, which can then be used to define the desired transformations. The use of these recursion schemes will allow us reuse, combine and reshape the syntax-directed transformations that we write. In addition, the embedding into a functional language will give us a lot of flexibility and expressive power such as a powerful type system and generic programming techniques.

As a starting point for our recursion schemes we consider various kinds of tree automata [3]. For each such kind we show how to implement them in

Haskell. From the resulting recursion schemes we then derive more sophisticated and highly modular recursion schemes. In particular, our contributions are the following:

- We implement bottom-up tree acceptors (Section 2), bottom-up tree transducers (Section 4) and top-down tree transducers (Section 5) as recursion schemes in Haskell. While the implementation of the first two is well-known, the implementation of the last one is new but entirely straightforward.

- From the thus obtained recursion schemes, we derive more modular variants (Section 3) using a variation of the well-know product automaton construction (Section 3.1) and Swierstra's *data types à la carte* [23] (Section 3.2).

- We decompose the recursion schemes derived from bottom-up and from top-down tree transducer into a homomorphism part and a state transition part (Section 4.5 and Section 5.3). This makes it possible to specify these two parts independently and to modify and combine them in a flexible manner.

- We derive a recursion scheme that combines both bottom-up and top-down state propagation (Section 6).

- We illustrate the merit of our recursion schemes by a running example in which we develop a simple compiler for a simple expression language. Utilising the modularity of our approach, we extend the expression language throughout the paper in order to show how the more advanced recursion schemes help us in devising an increasingly more complex compiler. In addition to that, the high degree of modularity of our approach not only simplifies the construction of the compiler but also allows us to reuse earlier iterations of the compiler.

Apart from the abovementioned running example, we also include a number of independent examples illustrating the mechanics of the presented tree automata.

The remainder of this paper is structured as follows: we start in Section 2 with bottom-up tree acceptors and their implementation in Haskell. In Section 3, we introduce two dimensions of modularity that can be exploited in the recursion scheme obtained from bottom-up tree acceptors. In Section 4, we will turn to bottom-up tree transducers, which, based on a state that is propagated upwards, perform a transformation of an input term to an output term. In Section 4.5 we will then introduce yet another dimension of modularity by separating the state propagation in tree transducers from the tree transformation. This will also allow us to adopt the modularity techniques from Section 3. In Section 5, we will do the same thing again, however, for top-down tree transducers in which the state is propagated top-down rather than bottom-up. Finally, in Section 6, we will combine both bottom-up and top-down state transitions.

The library of recursion schemes that we develop in this paper is available as part of the `compdata` package [2]. Additionally, this paper is written as a literate Haskell file[1], which can be directly loaded into the GHCi Haskell interpreter.

[1] Available from the author's web site.

2 Bottom-Up Tree Acceptors

The tree automata that we consider in this paper operate on terms over some signature $\mathcal{F}$. In the setting of tree automata, a signature $\mathcal{F}$ is simply a set of function symbols with a fixed arity and we write $f/n \in \mathcal{F}$ to indicate that f is a function symbol in $\mathcal{F}$ of arity n. Given a signature $\mathcal{F}$ and some set $\mathcal{X}$, the set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set T such that $\mathcal{X} \subseteq T$ and if $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in T$ then $f(t_1, \ldots, t_n) \in T$. Instead of $\mathcal{T}(\mathcal{F}, \emptyset)$ we also write $\mathcal{T}(\mathcal{F})$ and call elements of $\mathcal{T}(\mathcal{F})$ terms over $\mathcal{F}$. Tree automata run on terms in $\mathcal{T}(\mathcal{F})$.

Each of the tree automata that we describe in this paper consists at least of a finite set Q of states and a set of rules according to which an input term is transformed into an output term. While performing such a transformation, these automata maintain state information, which is stored in the intermediate results of the transformation. To this end each state $q \in Q$ is considered as a unary function symbol and a subterm t is annotated with state q by writing $q(t)$. For example, $f(q_0(a), q_1(b))$ represents the term $f(a, b)$, where the two subterms a and b are annotated with states q_0 and q_1, respectively.

The rules of the tree automata in this paper will all be of the form $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}', \mathcal{X})$, where $\mathcal{F}' = \mathcal{F} \uplus \{q/1 \mid q \in Q\}$. The rules can be read as term rewrite rules, i.e. the variables in l and t are placeholders that are instantiated with terms when the rule is applied. Running an automaton is then simply a matter of applying these term rewrite rules to a term. The different kinds of tree automata only differ in the set of rules they allow.

2.1 Deterministic Bottom-Up Tree Acceptors

A *deterministic bottom-up tree acceptor* (DUTA) over a signature $\mathcal{F}$ consists of a (finite) set of states Q, a set of accepting states $Q_a \subseteq Q$, and a set of transition rules of the form

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)), \quad \text{with } f/n \in \mathcal{F} \text{ and } q, q_1, \ldots, q_n \in Q$$

The variable symbols $x_1, \ldots, x_n$ serve as placeholders in these rules and states in Q are considered as function symbols of arity 1. The set of transition rules must be deterministic – i.e. there are no two different rules with the same left-hand side – and complete – i.e. for each $f/n \in \mathcal{F}$ and $q_1, \ldots, q_n \in Q$, there is a rule with the left-hand side $f(q_1(x_1), \ldots, q_n(x_n))$. The state q on the right-hand side of the transition rule is also called the *successor state* of the transition.

By repeatedly applying the transition rules to a term t over $\mathcal{F}$, initial states are created at the leaves which then get propagated upwards through function symbols. Eventually, we obtain a final state q_f at the root of the term. That is, an input term t is transformed into $q_f(t)$. The term t is *accepted* by the DUTA iff $q_f \in Q_a$. In this way, a DUTA defines a term language.

Example 1. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0\}$ and the DUTA

over $\mathcal{F}$ with $Q = \{q_0, q_1\}$, $Q_a = \{q_1\}$ and the following transition rules:

$$\begin{array}{lll} \mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{not}(x)) & \mathsf{and}(q_0(x), q_1(y)) \to q_0(\mathsf{and}(x, y)) \\ \mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{not}(x)) & \mathsf{and}(q_1(x), q_0(y)) \to q_0(\mathsf{and}(x, y)) \\ \multicolumn{2}{c}{\mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{and}(x, y))} & \mathsf{and}(q_0(x), q_0(y)) \to q_0(\mathsf{and}(x, y)) \end{array}$$

Terms over signature $\mathcal{F}$ are Boolean expressions and the automaton accepts such an expression iff it evaluates to true.

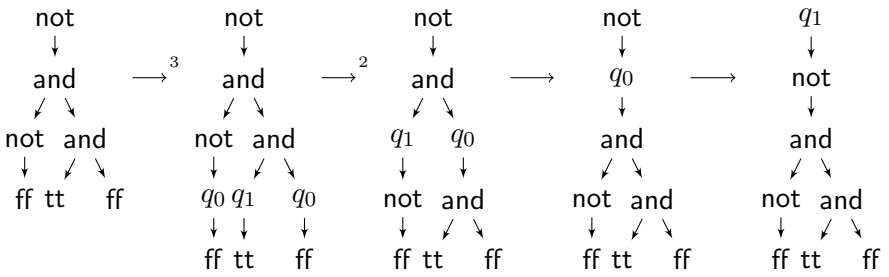
Note that the rules are complete – for each function symbol, every combination of input states occurs in the left-hand side of some rule – and deterministic – there are no two rules with the same left-hand side.

The transition rules are applied by interpreting them as rules in a term rewriting system, where variables are placeholders for terms. For the term $\mathsf{and}(\mathsf{tt}, \mathsf{ff})$, we get the following derivation:

$$\mathsf{and}(\mathsf{tt}, \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), q_0(\mathsf{ff})) \to q_0(\mathsf{and}(\mathsf{tt}, \mathsf{ff}))$$

The result of this derivation is the final state q_0; the term is rejected.

The following picture illustrates a run of the automaton on the bigger term $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{ff}), \mathsf{and}(\mathsf{tt}, \mathsf{ff})))$:



For the sake of conciseness, we applied rules in parallel where possible. At first we apply the rules to the leaves of the term, performing three rewrite steps in parallel. This effectively produces the initial states of the run. Subsequent rule applications propagate the states according to the rules until we obtain the final state at the root of the term.

Note that in both runs, apart from the final state at the root, the result term is the same as the one we started with. This is expected. The only significant output of a DUTA run is the final state.

The rules of a DUTA contain some syntactic overhead as they explicitly copy the function symbol from the left-hand side to the right-hand side. This formulation serves two purposes: first, it makes it possible to describe the run of a DUTA as a term reduction as in the above example. Secondly, we will see that the more sophisticated automata that we will consider later are simply generalisations of the rules of a DUTA, which for example do not require copying the function symbol but allow arbitrary transformations.

2.2 Algebras and Catamorphisms

For the representation of recursion schemes in Haskell, we consider data types as fixed points of polynomial functors:

$$\textbf{data}\ \mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$$

Given a functor f that represents some signature, *Term* f constructs its fixed point, which represents the terms over f. For example, the data type *Exp* from the introduction may be instead defined as *Term Sig* with[2]

$$\textbf{data}\ \mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$$

The functoriality of *Sig* is given by an instance of the type class *Functor*:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{Sig}\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Val}\ i) \quad\ = \mathit{Val}\ i\\ &\quad \mathit{fmap}\ f\ (\mathit{Plus}\ x\ y) = \mathit{Plus}\ (f\ x)\ (f\ y)\end{aligned}$$

The function *eval* from the introduction is defined by a simple recursion scheme: its recursive definition closely follows the recursive definition of the data type *Exp*. This recursion scheme is known as *catamorphism* (or also *fold*). Given an *algebra*, i.e. a functor f and type a together with a function of type $f\ a \to a$, its catamorphism is a function of type *Term* $f \to a$ constructed as follows:

$$\begin{aligned}&\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to (\mathit{Term}\ f \to a)\\ &\mathit{cata}\ \phi\ (\mathit{In}\ t) = \phi\ (\mathit{fmap}\ (\mathit{cata}\ \phi)\ t)\end{aligned}$$

In the definition of the algebra for the evaluation function, we make use of the fact that the arguments of the *Plus* constructor are already the results of evaluating the corresponding subexpressions:

$$\begin{aligned}&\mathit{evalAlg} :: \mathit{Sig}\ \mathit{Int} \to \mathit{Int}\\ &\mathit{evalAlg}\ (\mathit{Val}\ i) \quad\ = i\\ &\mathit{evalAlg}\ (\mathit{Plus}\ x\ y) = x + y\end{aligned}$$

$$\begin{aligned}&\mathit{eval} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Int}\\ &\mathit{eval} = \mathit{cata}\ \mathit{evalAlg}\end{aligned}$$

Programming in algebras and catamorphisms or other algebraic or coalgebraic recursion schemes is a well-known technique in functional programming [20]. We shall use this representation in order to implement the recursion schemes that we derive from the tree automata.

2.3 Bottom-Up State Transition Functions

If we omit the syntactic overhead of the state transition rules of DUTAs, we see that DUTAs are algebras – in fact, they were originally defined as such [5]. For instance, the algebra of the automaton in Example 1 is an algebra that evaluates Boolean expressions. Speaking in Haskell terms, a DUTA over a signature functor F is given by a type of states Q, a state transition function in the form of an

[2] *Term Sig* is "almost" isomorphic to *Exp*. The only difference stems from the fact that the constructor *In* is non-strict.

F-algebra $trans :: F\ Q \to Q$, and a predicate $acc :: Q \to Bool$. A term over F is an element of type $Term\ F$. When running a DUTA on a term t of type $Term\ F$, we obtain the final state $cata\ trans\ t$ of the run. Afterwards, the predicate acc checks whether the final state is accepting:

```
runDUTA :: Functor f ⇒ (f q → q) → (q → Bool) → Term f → Bool
runDUTA trans acc = acc . cata trans
```

Example 2. We implement the DUTA from Example 1 in Haskell as follows:

```
data F a = And a a
         | Not a
         | TT | FF
data Q = Q0 | Q1

acc :: Q → Bool
acc Q1 = True
acc Q0 = False

trans :: F Q → Q
trans FF          = Q0
trans TT          = Q1
trans (Not Q0)    = Q1
trans (Not Q1)    = Q0
trans (And Q1 Q1) = Q1
trans (And _  _ ) = Q0
```

The automaton is run on a term of type $Term\ F$ as follows:

```
evalBool :: Term F → Bool
evalBool = runDUTA trans acc
```

The restriction to a finite state space is not crucial for our purposes as we are not interested in deciding properties of automata. Instead, we want to use automata as powerful recursion schemes that allow for modular definitions of functions on terms. Since we are only interested in the traversal of the term that an automaton provides, we also drop the predicate and consider the final state as the output of a run of the automaton. We, therefore, consider only the transition function of a DUTA:

```
type UpState f q = f q → q
runUpState :: Functor f ⇒ UpState f q → Term f → q
runUpState = cata
```

With the functions *evalAlg* from Section 2.2 and *trans* from Example 2, we have already seen two simple examples of bottom-up state transition functions. In practice, only few state transitions of interest are that simple, of course.

In the following, we want to write a simple compiler for our expression language that generates code for a simple virtual machine with a single accumulator register and a random access memory indexed by non-negative integers. At first, we devise the instructions of the virtual machine:

```
type Addr = Int
data Instr = Acc Int | Load Addr | Store Addr | Add Addr
type Code = [Instr]
```

For simplicity, we use integers to represent addresses for the random access memory. The four instructions listed above write an integer constant to the accumulator, load the contents of a memory cell into the accumulator, store the contents

of the accumulator into a memory cell, and add the contents of a memory cell to the contents of the accumulator, respectively.

The code that we want to produce for an expression e of type *Term Sig* should evaluate e, i.e. after executing the code, the virtual machine's accumulator is supposed to contain the integer value *eval e*:

$$
\begin{aligned}
&\mathit{codeSt} :: \mathit{UpState\ Sig\ Code}\\
&\mathit{codeSt}\ (\mathit{Val}\ i) \quad\ = [\mathit{Acc}\ i]\\
&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a]\\
&\quad \textbf{where}\ a = \ldots
\end{aligned}
$$

In order to perform addition, the result of the computation for the first summand has to be stored into a temporary memory cell at some address a. However, we also have to make sure that this memory cell is not overwritten by the computation for the second summand. To this end, we maintain a counter that tells us which address is safe to use:

$$
\begin{aligned}
&\mathit{codeAddrSt} :: \mathit{UpState\ Sig}\ (\mathit{Code}, \mathit{Addr})\\
&\mathit{codeAddrSt}\ (\mathit{Val}\ i) = ([\mathit{Acc}\ i], 0)\\
&\mathit{codeAddrSt}\ (\mathit{Plus}\ (x, a')\ (y, a)) = (x +\!\!+ [\mathit{Store}\ a] +\!\!+ y +\!\!+ [\mathit{Add}\ a],\\
&\qquad\qquad\qquad\qquad 1 + \mathit{max}\ a\ a')\\
&\mathit{code} :: \mathit{Term\ Sig} \to \mathit{Code}\\
&\mathit{code} = \mathit{fst}\ .\ \mathit{runUpState\ codeAddrSt}
\end{aligned}
$$

While this definition yields the desired code generator, it is not very elegant as it mixes the desired output state – the code – with an auxiliary state – the fresh address. This flaw can be mitigated by using a state monad to carry around the auxiliary state. In this way we can still benefit from computing both states side by side has, which means that the input term is only traversed once.

This however still leaves the specification of two computations uncomfortably entangled, which is not only more prone to errors but also inhibits reuse and flexibility: the second component of the state, which we use as a fresh address, is in fact the height of the expression and might be useful for other computations:

$$
\begin{aligned}
&\mathit{heightSt} :: \mathit{UpState\ Sig\ Int}\\
&\mathit{heightSt}\ (\mathit{Val}\ \_) \quad\ = 0\\
&\mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y
\end{aligned}
$$

Moreover, as we extend the expression language with new language features, we might have to change the way we allocate memory locations for intermediate results. Thus, separating the two components of the computation is highly desirable since it would then allow us to replace the *heightSt* component with a different one while reusing the rest of the code generator.

The next section addresses this concern.

3 Making Tree Automata Modular

Our goal is to devise modular recursion schemes. In this section, we show how to leverage two dimensions of modularity inherent in tree automata, viz. the state

space and the signature. For each dimension, we present a well-know technique to make use of the modularity in the specification of automata. In particular, we shall demonstrate these techniques on bottom-up state transitions. However, due to their generality, both techniques are applicable also to the more advanced tree automata that we consider in later sections.

3.1 Product Automata

A common construction in automata theory combines two automata by simply forming the cartesian product of their state spaces and defining the state transition componentwise according to the state transitions of the original automata. The resulting automaton runs the original automata in parallel. We shall follow the same idea to construct the state transition *codeAddrSt* from Section 2.3 by combining the state transition *heightSt* with a state transition that computes the machine code using the state maintained by *heightSt*.

However, in contrast to the standard product automaton construction, the two computations in our example are not independent from each other – the code generator depends on the height in order to allocate memory addresses. Therefore, we need a means of communication between the constituent automata.

In order to allow access to components of a compound state space, we define a binary type class $\in$ that tells us if a type is a component of a product type and provides a projection for that component:

$$\begin{array}{l}\textbf{class } a \in b \textbf{ where}\\ \quad pr :: b \to a\end{array}$$

Using *overlapping instance declarations*, we define the relation $a \in b$ as follows:

$$\begin{array}{lll}\textbf{instance} & a \in a & \textbf{where } pr = id\\ \textbf{instance} & a \in (a, b) & \textbf{where } pr = fst\\ \textbf{instance } (c \in b) \Rightarrow & c \in (a, b) & \textbf{where } pr = pr\ .\ snd\end{array}$$

That is, we have $a \in b$ if b is of the form $(b_1, (b_2, ...))$ and $a = b_i$ for some i.

We generalise bottom-up state transitions by allowing the successor state of a transition to be dependent on a potentially larger state space:

$$\textbf{type } DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

The result state of type q for the state transition of the above type may depend on the states that are propagated from below. However, in contrast to ordinary bottom-up state transitions, these states – of type p – may contain more components in addition to the component of type q.

Every ordinary bottom-up state transition such as *heightSt* can be readily converted into such a *dependent bottom-up state transition function* by precomposing the projection pr:

$$\begin{array}{l}dUpState :: Functor\ f \Rightarrow UpState\ f\ q \to DUpState\ f\ p\ q\\ dUpState\ st = st\ .\ fmap\ pr\end{array}$$

A dependent state transition function is the same as an ordinary state transition function if the state spaces p and q coincide. Hence, we can run such a dependent state transition function in the same way:

$$\begin{aligned}&\mathit{runDUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{DUpState}\ f\ q\ q \to \mathit{Term}\ f \to q\\&\mathit{runDUpState}\ f = \mathit{runUpState}\ f\end{aligned}$$

When defining a dependent state transition function, we can make use of the fact that the state propagated from below may contain additional components. For the definition of the state transition function generating the code, we declare that we expect an additional state component of type Int.

$$\begin{aligned}&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\&\mathit{codeSt}\ (\mathit{Val}\ i) \quad\ = [\mathit{Acc}\ i]\\&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\&\quad \textbf{where}\ a = \mathit{pr}\ y\end{aligned}$$

Using the method pr of the type class $\in$, we project to the desired components of the state: $\mathit{pr}\ x$ and the first occurrence of $\mathit{pr}\ y$ are of type Code whereas the second occurrence of $\mathit{pr}\ y$ is of type Int.

The product construction that combines two dependent state transition functions is simple: it takes two state transition functions depending on the same (compound) state space and combines them by forming the product of their respective outcomes:

$$\begin{aligned}(\otimes) :: (p \in c, q \in c) \Rightarrow\ & \mathit{DUpState}\ f\ c\ p \to \mathit{DUpState}\ f\ c\ q\\ & \to \mathit{DUpState}\ f\ c\ (p, q)\\(\mathit{sp} \otimes \mathit{sq})\ t = (\mathit{sp}\ t, \mathit{sq}\ t)\end{aligned}$$

We obtain the desired code generator from Section 2.3 by combining our two (dependent) state transition functions and running the resulting state transition function:

$$\begin{aligned}&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Code}\\&\mathit{code} = \mathit{fst}\ .\ \mathit{runDUpState}\ (\mathit{codeSt} \otimes \mathit{dUpState}\ \mathit{heightSt})\end{aligned}$$

Note that combining state transition functions in this way is not restricted to such simple dependencies. State transition functions may depend on each other. The construction that we have seen in this section makes it possible to decompose state spaces into isolated modules with a typed interface to access them. This practice of decomposing state spaces is not different from the abstraction and reuse that we perform when writing mutual recursive functions. Functions which can be defined in this way are also known as *mutumorphisms* [6].

There are still two minor shortcomings, which we shall address when we consider other types of automata below. First, the extraction of components from compound states is purely based on the type information, which can easily result in confusion of distinct state components that happen to have the same type. This can be seen in the instance declarations for the type class $\in$, which are overlapping and will simply select the left-most occurrence of a type. Secondly, we

only allow access to the state of the children of the current node. In principle, this restriction is no problem as we can use the states of the children nodes to compute the state of the current node. For example, if, in the code generation, we needed the height of the current expression instead of the height of the right summand, we could have computed it from the height of both summands. However, this means that code as well as the corresponding computations are duplicated since the state of the current node is already computed by the corresponding state transition.

3.2 Compositional Data Types

We also want to leverage the modularity that stems from the data types on which we want to define functions. This modularity is based on the ability to combine functors by forming coproducts:

$$\begin{aligned}
&\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)\\
&\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (f \oplus g)\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{Inl}\ e) = \mathit{Inl}\ (\mathit{fmap}\ f\ e)\\
&\quad \mathit{fmap}\ f\ (\mathit{Inr}\ e) = \mathit{Inr}\ (\mathit{fmap}\ f\ e)
\end{aligned}$$

Using the $\oplus$ operator, we can extend the signature functor *Sig* with an increment operation, for example:

$$\begin{aligned}
&\textbf{data}\ \mathit{Inc}\ e = \mathit{Inc}\ e\\
&\textbf{type}\ \mathit{Sig}' \ = \mathit{Inc} \oplus \mathit{Sig}
\end{aligned}$$

In order to make use of this composition of functors for defining automata on functors in a modular fashion, we will follow Swierstra's *data types à la carte* [23] approach, which we will summarise briefly below.

The use of coproducts entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the correct tags automatically, injections are derived using a type class:

$$\begin{aligned}
&\textbf{class}\ \mathit{sub} \preceq \mathit{sup}\ \textbf{where}\\
&\quad \mathit{inj} :: \mathit{sub}\ a \to \mathit{sup}\ a
\end{aligned}$$

Similarly to the type class $\in$, we define the subsignature relation $\preceq$ as follows:

$$\begin{aligned}
&\textbf{instance} && f \preceq f && \textbf{where}\ \mathit{inj} = \mathit{id}\\
&\textbf{instance} && f \preceq (f \oplus g) && \textbf{where}\ \mathit{inj} = \mathit{Inl}\\
&\textbf{instance}\ (f \preceq g) \Rightarrow && f \preceq (h \oplus g) && \textbf{where}\ \mathit{inj} = \mathit{Inr}\ .\ \mathit{inj}
\end{aligned}$$

That is, we have $f \preceq g$ if g is of the form $g_1 \oplus (g_2 \oplus ...)$ and $f = g_i$ for some i. From the injection function *inj*, we derive an injection function for terms:

$$\begin{aligned}
&\mathit{inject} :: (g \preceq f) \Rightarrow g\ (\mathit{Term}\ f) \to \mathit{Term}\ f\\
&\mathit{inject} = \mathit{In}\ .\ \mathit{inj}
\end{aligned}$$

Additionally, in order to reduce syntactic overhead, we assume, for each signature functor such as *Sig* or *Inc*, smart constructors that comprise the injection, e.g.:

$$
\begin{aligned}
&\mathit{plus} :: (\mathit{Sig} \preceq f) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ f \to \mathit{Term}\ f \\
&\mathit{plus}\ x\ y = \mathit{inject}\ (\mathit{Plus}\ x\ y) \\
&\mathit{inc} :: (\mathit{Inc} \preceq f) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ f \\
&\mathit{inc}\ x = \mathit{inject}\ (\mathit{Inc}\ x)
\end{aligned}
$$

Using these smart constructors, we can write, for example, $\mathit{inc}\ (\mathit{val}\ 3\ `\mathit{plus}`\ \mathit{val}\ 4)$ to denote the expression $\mathit{inc}(3 + 4)$.

For writing modular functions on compositional data types, we use type classes. For example, for recasting the definition of the *heightSt* state transition function, we introduce a new type class and make it propagate over coproducts:

$$
\begin{aligned}
&\textbf{class}\ \mathit{HeightSt}\ f\ \textbf{where} \\
&\quad \mathit{heightSt} :: \mathit{UpState}\ f\ \mathit{Int} \\
&\textbf{instance}\ (\mathit{HeightSt}\ f, \mathit{HeightSt}\ g) \Rightarrow \mathit{HeightSt}\ (f \oplus g)\ \textbf{where} \\
&\quad \mathit{heightSt}\ (\mathit{Inl}\ x) = \mathit{heightSt}\ x \\
&\quad \mathit{heightSt}\ (\mathit{Inr}\ x) = \mathit{heightSt}\ x
\end{aligned}
$$

The above instance declaration lifts instances of *HeightSt* over coproducts in a straightforward manner. Subsequently, we will omit these instance declarations as they always follow the same pattern and thus can be generated automatically like instances declarations for *Functor*.

We then instantiate this class for each (atomic) signature functor separately:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{HeightSt}\ \mathit{Sig}\ \textbf{where} \\
&\quad \mathit{heightSt}\ (\mathit{Val}\ \_) \quad = 0 \\
&\quad \mathit{heightSt}\ (\mathit{Plus}\ x\ y) = 1 + \mathit{max}\ x\ y \\
&\textbf{instance}\ \mathit{HeightSt}\ \mathit{Inc}\ \textbf{where} \\
&\quad \mathit{heightSt}\ (\mathit{Inc}\ x) \quad = 1 + x
\end{aligned}
$$

Due to the propagation of instances over coproducts, we obtain an instance of *HeightSt* for Sig' for free.

With the help of the type class *HeightSt*, we eventually obtain an extensible definition of the height function.

$$
\begin{aligned}
&\mathit{height} :: (\mathit{Functor}\ f, \mathit{HeightSt}\ f) \Rightarrow \mathit{Term}\ f \to \mathit{Int} \\
&\mathit{height} = \mathit{runUpState}\ \mathit{heightSt}
\end{aligned}
$$

Since we have instantiated *HeightSt* for the signature Sig' and all its subsignatures, the function *height* may be given any argument of type $\mathit{Term}\ f$, where f is the Sig' or any of its subsignatures. Moreover, by simply providing further instance declarations for *HeightSt*, we can extend the domain of *height* to further signatures.

4 Bottom-Up Tree Transducers

A compiler usually consists of several stages that perform diverse kinds of transformations on the abstract syntax tree, e.g. renaming variables or removing syntactic sugar. Representing syntax trees as terms, i.e. values of type $\mathit{Term}\ f$,

such transformations are functions of type *Term* $f \to$ *Term* g that map terms over some signature to terms over a potentially different signature. Tree transducers are a well-established technique for specifying such transformations [3, 7]. Moreover, there are a number of composition theorems that permit the composition of certain tree transducers such that the transformation function denoted by the composition is equal to the composition of the transformation functions denoted by the original tree transducers [7]. These composition theorems permit us to perform deforestation [26], i.e. eliminating intermediate results by fusing several stages of a compiler to a single tree transducer [16, 25], thus making tree transducers an attractive recursion scheme.

4.1 Deterministic Bottom-Up Tree Transducers

A *deterministic bottom-up tree transducer* (*DUTT*) defines – like a DUTA – for each function symbol a successor state. But, additionally, it also defines an expression that should replace the original function symbol. More formally, a DUTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q and a set of transduction rules of the form

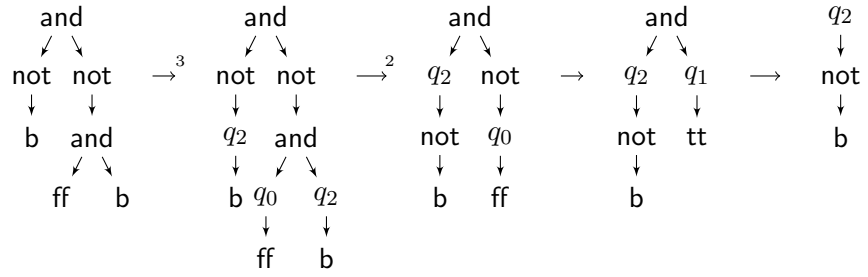
$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(u), \quad \text{with } f \in \mathcal{F} \text{ and } q, q_1, \ldots, q_n \in Q$$

where $u \in \mathcal{T}(\mathcal{G}, \mathcal{X})$ is a term over signature $\mathcal{G}$ and the set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$. Compare this to the state transition rules of DUTAs, which are simply a restriction of the transduction rules above with $u = f(x_1, \ldots, x_n)$, thus only allowing the identity transformation. By repeatedly applying its transduction rules in a bottom-up fashion, a run of a DUTT transforms an input term over $\mathcal{F}$ into an output term over $\mathcal{G}$ plus – similarly to DUTAs – a final state at the root.

Example 3. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{ff}/0, \mathsf{tt}/0, \mathsf{b}/0\}$ and the DUTT from $\mathcal{F}$ to $\mathcal{F}$ with $Q = \{q_0, q_1, q_2\}$ and the following transduction rules:

$$\begin{array}{lll}
\mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{tt}) & \mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{tt}) \\
\mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{ff}) & \mathsf{and}(q_1(x), q_2(y)) \to q_2(y) \\
\mathsf{b} \to q_2(\mathsf{b}) & \mathsf{not}(q_2(x)) \to q_2(\mathsf{not}(x)) & \mathsf{and}(q_2(x), q_1(y)) \to q_2(x) \\
\mathsf{and}(q(x), p(y)) \to q_0(\mathsf{ff}) \quad \text{if } q_0 \in \{p, q\} & & \mathsf{and}(q_2(x), q_2(y)) \to q_2(\mathsf{and}(x, y))
\end{array}$$

The signature $\mathcal{F}$ allows us to express Boolean expression containing a single Boolean variable b. When applied to such an expression, the automaton performs constant folding, i.e. it evaluates subexpression if possible. With the states q_0 and q_1 it signals that a subexpression is false respectively true; q_2 indicates uncertainty. For example, applying the automaton to the expression $\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{not}(\mathsf{and}(\mathsf{ff}, \mathsf{b})))$ yields the following derivation:



The rules for the constant symbols do not perform any transformation in this example and simply provide initial states. Then the first real transformation is performed, which collapses the subterm rooted in and to $q_0(\mathsf{ff})$. The run of the automaton is completed as soon as a state appears at the root, the final state of the run.

4.2 Contexts in Haskell

In order to, represent transduction rules in Haskell, we need a representation of the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of terms over signature $\mathcal{F}$ and variables $\mathcal{X}$. We call such extended terms *contexts*. These contexts appear on the right-hand side of transduction rules of DUTTs. We obtain a representation of contexts by simply extending the definition of the data type *Term* by an additional constructor:

```
data Context f a = In (f (Context f a)) | Hole a
```

We call this additional constructor *Hole* as we will use it also for things other than variables. For example, the holes in a context may be filled by other contexts over the same signature. The following function substitutes the contexts in the holes into the surrounding context.

```
appCxt :: Functor f ⇒ Context f (Context f a) → Context f a
appCxt (Hole x) = x
appCxt (In t)   = In (fmap appCxt t)
```

Context f is in fact the *free monad* of the functor f with *Hole* and *appCxt* as unit and multiplication operation, respectively. The functoriality of *Context f* is given as follows:

```
instance Functor f ⇒ Functor (Context f) where
  fmap f (Hole v) = Hole (f v)
  fmap f (In t)   = In (fmap (fmap f) t)
```

Recall that the set of terms $\mathcal{T}(\mathcal{F})$ is defined as the set $\mathcal{T}(\mathcal{F}, \emptyset)$ of terms without variables. We can do the same in the Haskell representation and replace our definition of the type *Term* with the following:

```
data Empty
type Term f = Context f Empty
```

Here, *Empty* is simply an empty type.[3] This definition of *Term* allows us to use terms and context in a uniform manner. For example, the function *appCxt* defined above can also be given the type $Context\ f\ (Term\ f) \to Term\ f$. Moreover, this encoding allows us to give a more general type for the injection function:

$$inject :: (g \preceq f) \Rightarrow g\ (Context\ f\ a) \to Context\ f\ a$$

The definition of *inject* remains the same. The same also applies to smart constructors; for example, the smart constructor *plus* has now the more general type

$$plus :: (Sig \preceq f) \Rightarrow Context\ f\ a \to Context\ f\ a \to Context\ f\ a$$

Most of the time we are using very simple contexts that only consist of a single functor application as constructed by the following function:

$$simpCxt :: Functor\ f \Rightarrow f\ a \to Context\ f\ a$$
$$simpCxt\ t = In\ (fmap\ Hole\ t)$$

4.3 Bottom-Up Transduction Functions

The transduction rules of a DUTT use placeholder variables x_1, x_2, etc. in order to refer to arguments of function symbols. These placeholder variables can then be used on the right-hand side of a transduction rule. This mechanism makes it possible to rearrange, remove and duplicate the terms that are matched against these placeholder variables. On the other hand, it is not possible to inspect them. For instance, in Example 3, $\mathsf{not}(q_0(\mathsf{ff})) \to q_1(\mathsf{tt})$ would not be a valid transduction rule as we are not allowed to pattern match on the arguments of not. We can only observe the state.

When representing transduction rules as Haskell functions, we have to be careful in order to maintain this restriction on DUTTs. In their categorical representation, Hasuo et al. [11] recognised that the restriction due to placeholder variables in the transduction rules can be enforced by a *naturality* condition. Naturality, in turn, can be represented in Haskell's type system as *parametric polymorphism*. Following this approach, we represent DUTTs from signature functor f to signature functor g with state space q by the following type:

$$\textbf{type}\ UpTrans\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to (q, Context\ g\ a)$$

In the definition of tree automata, states are used syntactically as a unary function symbol – an argument with state q is written as $q(x)$ in the left-hand side. In the Haskell representation, we use pairs and simply write (q, x).

In the type *UpTrans*, the type variable a represents the type of the placeholder variables. The universal quantification over a makes sure that placeholders can only be used if they appear on the left-hand side and that they cannot be inspected.

[3] Note that in Haskell, every data type – including *Empty* – is inhabited by $\bot$. Thus the definition of *Term* is not entirely accurate. However, for the sake of simplicity, we prefer this definition over a more precise one such as in [1].

Example 4. We implement the DUTT from Example 3 in Haskell. At first we define the signature and the state space.

```
data F a = And a a | Not a | TT | FF | B
data Q   = Q0 | Q1 | Q2
```

For the definition of the transduction function, we use the smart constructors *and*, *not*, *tt*, *ff* and *b* for the constructors of the signature F. These smart constructors are defined as before, e.g.

```
and :: (F ≼ f) ⇒ Context f a → Context f a → Context f a
and x y = inject (And x y)
```

The definition of the transduction function is a one-to-one translation of the transduction rules of the DUTT from Example 3.

```
trans :: UpTrans F Q F
trans TT = (Q1, tt);          trans (Not (Q0, x)) = (Q1, tt)
trans FF = (Q0, ff);          trans (Not (Q1, x)) = (Q0, ff)
trans B  = (Q2, b);           trans (Not (Q2, x)) = (Q2, not (Hole x))
trans (And (q, x) (p, y))
  | q ≡ Q0 ∨ p ≡ Q0       = (Q0, ff)
trans (And (Q1, x) (Q1, y)) = (Q1, tt)
trans (And (Q1, x) (Q2, y)) = (Q2, Hole y)
trans (And (Q2, x) (Q1, y)) = (Q2, Hole x)
trans (And (Q2, x) (Q2, y)) = (Q2, and (Hole x) (Hole y))
```

Since we do not constrain ourselves to finite state spaces, DUTTs do not add any expressive power to the state transition functions of DUTAs. Each DUTT can be transformed into an algebra whose catamorphism is the transformation denoted by the DUTT:

```
runUpTrans :: (Functor f, Functor g) ⇒ UpTrans f q g
                                     → Term f → (q, Term g)
runUpTrans trans = cata (appCxt′ . trans)
  where appCxt′ (x, y) = (x, appCxt y)
```

For instance, we run the DUTT from Example 4 as follows:

```
foldBool :: Term F → (Q, Term F)
foldBool = runUpTrans trans
```

As we have seen in Section 3.1, a tree acceptor with a compound state space comprises several computations which may be disentangled in order to increase modularity. A tree transducer intrinsically combines two computations: the state transition and the actual transformation of the term. We will see in Section 4.5 how to disentangle these two components. Before that, we shall look at a special case of DUTTs.

4.4 Tree Homomorphisms

To simplify matters, Bahr and Hvitved [1] focused on tree transducers with a singleton state space, also known as *tree homomorphisms* [3]:

```
type Hom f g = ∀ a . f a → Context g a
runHom :: (Functor f, Functor g) ⇒ Hom f g → Term f → Term g
runHom hom = cata (appCxt . hom)
```

Tree homomorphisms can only transform the tree structure uniformly without the ability to maintain a state. Nonetheless, tree homomorphisms provide a useful recursion scheme. For example, desugaring, i.e. transforming syntactic sugar of a language to the language's core operations, can in many cases be implemented as a tree homomorphism. Reconsider the signature $Sig' = Inc \oplus Sig$ that extends Sig with an increment operator. The increment operator is only syntactic sugar for adding the value 1. The corresponding desugaring transformation can be implemented as a tree homomorphism:

```
class DesugHom f g where
  desugHom :: Hom f g
  -- instance declaration lifting DesugHom to coproducts omitted
desugar :: (Functor f, Functor g, DesugHom f g) ⇒ Term f → Term g
desugar = runHom desugHom
instance (Sig ≼ g) ⇒ DesugHom Inc g where
  desugHom (Inc x) = Hole x `plus` val 1
instance (Functor g, f ≼ g) ⇒ DesugHom f g where
  desugHom = simpCxt . inj
```

The first instance declaration states that as long as the target signature g contains Sig, we can desugar the signature Inc to g by mapping $inc(x)$ to $x + 1$. Using overlapping instances, the second instance declaration then defines the desugaring for all other signatures f – provided f is contained in the target signature – by leaving the input untouched.

The above instance declarations make it now possible to use the *desugar* function with type $Term\ Sig' \to Term\ Sig$. That is, *desugar* transforms a term over signature Sig' to a term over signature Sig.

As an ordinary recursive Haskell function we would implement desugaring as follows:

```
data Exp  = Val  Int | Plus  Exp  Exp
data Exp' = Val' Int | Plus' Exp' Exp' | Inc' Exp'
desugExp :: Exp' → Exp
desugExp (Val' i)    = Val i
desugExp (Plus' e f) = desugExp e `Plus` desugExp f
desugExp (Inc' e)    = desugExp e `Plus` Val 1
```

Note that we have to provide two separate data types for the input and output types of the function instead of using the compositionality of signatures.

Moreover, the function *desugar* is applicable more broadly. It can be used as a function of type $\mathit{Term}\ (f \oplus \mathit{Inc}) \to \mathit{Term}\ f$ for any signature f that contains Sig, i.e. for which we have $\mathit{Sig} \preceq f$. Apart from these advantages in modularity and extensibility we also obtain all the advantages of using a transducer, which we shall discuss in more detail in Section 7.

4.5 Combining Tree Homomorphisms with State Transitions

We aim to combine the simplicity of tree homomorphisms and the expressivity of bottom-up tree transducers. To this end, we shall devise a method to combine a tree homomorphism and a state transition function to form a DUTT. This construction will be complete in the sense that any DUTT can be constructed in this way.

At first, compare the types of automata that we have considered so far:

$$\begin{aligned}
&\textbf{type}\ \mathit{Hom}\ \ f\ \ g = \forall\ a\ .\ f\ \ a\ \ \to\ \ \mathit{Context}\ g\ a\\
&\textbf{type}\ \mathit{UpState}\ f\ q\ \ =\ \ \ f\ \ q\ \ \to\ \ q\\
&\textbf{type}\ \mathit{UpTrans}\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to (q, \mathit{Context}\ g\ a)
\end{aligned}$$

We can observe from this – admittedly suggestive – comparison that a bottom-up tree transducer is roughly a combination of a tree homomorphism and a state transition function. Our aim is to make use of this observation by decomposing the specification of a bottom-up tree transducer into a tree homomorphism and a bottom-up state transition function. Like for the product construction of state transition functions from Section 3.1, we have to provide a mechanism to deal with dependencies between the two components. Since the state transition is independent from the tree transformation, we only need to allow the tree homomorphism to access the state information that is produced by the bottom-up state transition.

A *stateful tree homomorphism* can thus be (tentatively) defined as follows:

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to \mathit{Context}\ g\ a$$

Since q appears to the left of the function arrow but not to the right, functions of the above type have access to the states of the arguments, but do not transform the state themselves. However, we want to make it easy to ignore the state if it is not needed as the state is often only needed for a small number of cases. This goal can be achieved by replacing the pairing with the state space q by an additional argument of type $a \to q$.

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (a \to q) \to f\ a \to \mathit{Context}\ g\ a$$

We can still push this interface even more to the original tree homomorphism type *Hom* by turning the function argument into an implicit parameter [17]:

$$\textbf{type}\ \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (?\mathit{state} :: a \to q) \Rightarrow f\ a \to \mathit{Context}\ g\ a$$

In a last refinement step, we add an implicit parameter that provides access to the state of the current node as well:

$$\begin{aligned}&\textbf{type } \mathit{QHom}\ f\ q\ g = \forall\ a\ .\ (?\mathit{above} :: q, ?\mathit{below} :: a \to q)\\ &\qquad\qquad \Rightarrow f\ a \to \mathit{Context}\ g\ a\end{aligned}$$

Functions with implicit parameters have to be invoked in the scope of appropriate bindings. For functions of the above type this means that $?\mathit{below}$ has to be bound to a function of type $a \to q$ and $?\mathit{above}$ to a value of type q. We shall use the following function to make implicit parameters explicit:

$$\begin{aligned}&\mathit{explicit} :: ((?\mathit{above} :: q, ?\mathit{below} :: a \to q) \Rightarrow b) \to q \to (a \to q) \to b\\ &\mathit{explicit}\ x\ ab\ be = x\ \textbf{where}\ ?\ \mathit{above} = ab; ?\mathit{below} = be\end{aligned}$$

In particular, given a stateful tree homomorphism h of type $\mathit{QHom}\ f\ q\ g$, we thus obtain a function $\mathit{explicit}\ h$ of type $q \to (a \to q) \to f\ a \to \mathit{Context}\ g\ a$.

The use of implicit parameters is solely for reasons of syntactic appearance and convenience. One can think of implicit parameters as reader monads without the syntactic overhead of monads. If, in the definition of a stateful tree homomorphism, the state is not needed, it can be easily ignored. Hence, tree homomorphisms are, in fact, also syntactic special cases of stateful tree homomorphisms.

The following construction combines a stateful tree homomorphism of type $\mathit{QHom}\ f\ q\ g$ and a state transition function of type $\mathit{UpState}\ f\ q$ into a tree transducer of type $\mathit{UpTrans}\ f\ q\ g$, which can then be used to perform the desired transformation:

$$\begin{aligned}&\mathit{upTrans} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\ &\qquad \mathit{UpState}\ f\ q \to \mathit{QHom}\ f\ q\ g \to \mathit{UpTrans}\ f\ q\ g\\ &\mathit{upTrans}\ st\ hom\ t = (q, c)\ \textbf{where}\\ &\quad q = st\ (\mathit{fmap}\ \mathit{fst}\ t)\\ &\quad c = \mathit{fmap}\ \mathit{snd}\ (\mathit{explicit}\ \mathit{hom}\ q\ \mathit{fst}\ t)\\ &\mathit{runUpHom} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\ &\qquad \mathit{UpState}\ f\ q \to \mathit{QHom}\ f\ q\ g \to \mathit{Term}\ f \to (q, \mathit{Term}\ g)\\ &\mathit{runUpHom}\ st\ hom = \mathit{runUpTrans}\ (\mathit{upTrans}\ st\ hom)\end{aligned}$$

Often the state space accessed by a stateful tree homomorphism is compound. Therefore, it is convenient to have the projection function pr built into the interface to the state space:

$$\begin{aligned}&\mathit{above} :: (?\mathit{above} :: q, p \in q) \Rightarrow p\\ &\mathit{above} = pr\ ?\ \mathit{above}\\ &\mathit{below} :: (?\mathit{below} :: a \to q, p \in q) \Rightarrow a \to p\\ &\mathit{below} = pr\ .\ ?\mathit{below}\end{aligned}$$

In order to illustrate how stateful tree homomorphisms are programmed, we extend the signature Sig with variables and let bindings:

$$\begin{aligned}&\textbf{type } \mathit{Name} = \mathit{String}\\ &\textbf{data } \mathit{Let}\ e\ \ = \mathit{LetIn}\ \mathit{Name}\ e\ e \mid \mathit{Var}\ \mathit{Name}\\ &\textbf{type } \mathit{LetSig} = \mathit{Let} \oplus \mathit{Sig}\end{aligned}$$

We shall implement a simple optimisation that removes let bindings whenever the variable that is bound is not used in the scope of the let binding. To this end, we define a state transition that computes the set of free variables:

```
type Vars = Set Name
class FreeVarsSt f where
  freeVarsSt :: UpState f Vars
instance FreeVarsSt Sig where
  freeVarsSt (Plus x y) = x `union` y
  freeVarsSt (Val _)    = empty
instance FreeVarsSt Let where
  freeVarsSt (Var v)      = singleton v
  freeVarsSt (LetIn v e s) = if v `member` s then delete v (e `union` s)
                                             else s
```

Note that the free variables occurring in the right-hand side of a binding are only included if the bound variable occurs in the scope of the let binding. The transformation itself is simple:

```
class RemLetHom f q g where
  remLetHom :: QHom f q g
instance (Vars ∈ q, Let ≼ g, Functor g) ⇒ RemLetHom Let q g where
  remLetHom (LetIn v _ s) | ¬ (v `member` below s) = Hole s
  remLetHom t                                      = simpCxt (inj t)
instance (Functor f, Functor g, f ≼ g) ⇒ RemLetHom f q g where
  remLetHom = simpCxt . inj
```

The homomorphism removes a let binding whenever the bound variable is not found in the set of free variables. Otherwise, no transformation is performed. Notice that the type specifies that the transformation depends on a state space that at least contains a set of variables. In addition, we make use of overlapping instances to define the transformation for all signatures different from *Let*. We then obtain the desired transformation function by combining the stateful tree homomorphism with the state transition computing the free variables:

```
remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
       ⇒ Term f → Term f
remLet = snd . runUpHom freeVarsSt remLetHom
```

In particular, we can give *remLet* the type $Term\ LetSig \to Term\ LetSig$ but also $Term\ (Inc \oplus LetSig) \to Term\ (Inc \oplus LetSig)$.

4.6 Refining Dependent Bottom-Up State Transition Functions

The implicit parameters ?*below* and ?*above* of stateful tree homomorphisms provide an interface to the states of the children of the current node as well as the state of the current node itself. The same interface can be given to dependent bottom-up state transition functions as well. We therefore redefine the type of these state transitions from Section 3.1 as follows:

$$\textbf{type}\ DUpState\ f\ p\ q = \forall\ a\ .\ (?below :: a \to p, ?above :: p, q \in p) \Rightarrow f\ a \to q$$

While the definition of the product operator $\otimes$ remains the same, we have to change the other functions slightly to accommodate this change:

```
dUpState :: Functor f ⇒ UpState f q → DUpState f p q
dUpState st = st . fmap below

upState :: DUpState f q q → UpState f q
upState st s = res where
  res = explicit st res id s

runDUpState :: Functor f ⇒ DUpState f q q → Term f → q
runDUpState = runUpState . upState
```

Note that definition of *res* in *upState* is cyclic and thus crucially depends on Haskell's non-strict semantics. This also means that dependent state transition functions do not necessarily yield a terminating run since one can create a cyclic dependency by defining a state transition that depends on its own result such as the following:

```
loopSt :: DUpState f p q
loopSt _ = above
```

The definition of the code generator from Section 3.1 is easily adjusted to the slightly altered interface of dependent state transitions. Since we intend to extend the code generator in Section 6, we also turn it into a type class:

```
class CodeSt f q where
  codeSt :: DUpState f q Code

code :: (Functor f, CodeSt f (Code, Int), HeightSt f)
     ⇒ Term f → (Code, Addr)
code = runDUpState (codeSt ⊗ dUpState heightSt)

instance (Int ∈ q) ⇒ CodeSt Sig q where
  codeSt (Val i)     = [Acc i]
  codeSt (Plus x y) = below x ++ [Store a] ++ below y ++ [Add a]
    where a = below y
```

Note that the access to the state of the current node – via *above* – solves one of the minor issues we have identified at the end of Section 3.1. In order to obtain the state of the current node, we do not have to duplicate the corresponding state transition anymore. Moreover, we can use the same interface when we move to top-down state transitions in the next section.

5 Top-Down Automata

Operations on abstract syntax trees are often dependent on a state that is propagated top-down rather than bottom-up, e.g. typing environments and variable

bindings. For such operations, recursion schemes derived from bottom-up automata are not sufficient. Hence, we shall consider top-down automata as a complementary paradigm to overcome this restriction.

Unlike the bottom-up case, we will not start with acceptors but with transducers. Our interest for bottom-up acceptors was based on the fact that such automata produce an output state. For top-down acceptors this application vanishes since such automata rather consume an input state than produce an output state. We will however come back to top-down state transition in order to make the state transition of top-down transducer modular – using the same stateful tree homomorphisms that we introduced in Section 4.5.

5.1 Deterministic Top-Down Tree Transducers

Deterministic top-down tree transducers (DDTTs) are able to produce transformations that depend on a top-down flow of information. They work in a fashion similar to bottom-up tree transducers but propagate their state downwards rather than upwards. More formally, a DDTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q, an initial state $q_0 \in Q$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n)) \to u \quad \text{with } f \in \mathcal{F} \text{ and } q \in Q$$

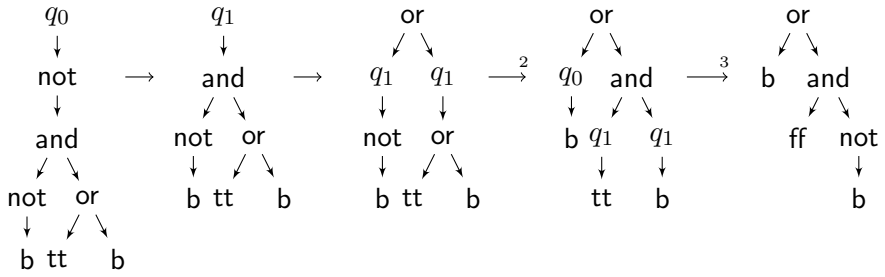
where $u \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ is a term over $\mathcal{G}$ and $Q(\mathcal{X}) = \{p(x_i) \,|\, p \in Q, 1 \leq i \leq n\}$. That is, the right-hand side is a term that may have subterms of the form $p(x_i)$ with x_i a variable from the left-hand side and p a state in Q. In other words, each *occurrence* of a variable on the right-hand side is given a successor state.

In order to run a DDTT on a term $t \in \mathcal{T}(\mathcal{F})$, we have to provide an initial state q_0 and then apply the transduction rules to $q_0(t)$ in a top-down fashion. Eventually, this yields a result term $t' \in \mathcal{T}(\mathcal{G})$.

Example 5. Consider the signature $\mathcal{F} = \{\mathsf{or}/2, \mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0, \mathsf{b}/0\}$ and the DDTT from $\mathcal{F}$ to $\mathcal{F}$ with the set of states $Q = \{q_0, q_1\}$, initial state q_0 and the following transduction rules:

$$\begin{array}{llll} q_0(\mathsf{b}) \to \mathsf{b} & q_0(\mathsf{tt}) \to \mathsf{tt} & q_0(\mathsf{ff}) \to \mathsf{ff} & q_0(\mathsf{not}(x)) \to q_1(x) \\ q_1(\mathsf{b}) \to \mathsf{not}(\mathsf{b}) & q_1(\mathsf{tt}) \to \mathsf{ff} & q_1(\mathsf{ff}) \to \mathsf{tt} & q_1(\mathsf{not}(x)) \to q_0(x) \\ q_0(\mathsf{and}(x,y)) \to \mathsf{and}(q_0(x), q_0(y)) & & q_0(\mathsf{or}(x,y)) \to \mathsf{or}(q_0(x), q_0(y)) & \\ q_1(\mathsf{and}(x,y)) \to \mathsf{or}(q_1(x), q_1(y)) & & q_1(\mathsf{or}(x,y)) \to \mathsf{and}(q_1(x), q_1(y)) & \end{array}$$

Terms over $\mathcal{F}$ are Boolean expressions with a single Boolean variable b. The above DDTT transforms such an expression into negation normal form by moving the operator not inwards. For instance, applied to the Boolean expression $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{or}(\mathsf{tt}, \mathsf{b})))$, the automaton yields the following derivation:



In order to start the run of a DDTT, the initial state q_0 has to be explicitly inserted at the root of the input term. The run of the automaton is completed as soon as all states in the term have vanished; there is no final state.

5.2 Top-Down Transduction Functions

Similar to bottom-up tree transducers, we follow the *placeholders-via-naturality* principle of Hasuo et al. [11] in order to represent top-down transduction functions:

$$\textbf{type}\ \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to \mathit{Context}\ g\ (q, a)$$

Now the state comes from above and is propagated downwards to the holes of the context, which defines the actual transformation that the transducer performs.

Running a top-down tree transducer on a term is a straightforward affair:

$$\begin{aligned}
&\mathit{runDownTrans} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{DownTrans}\ f\ q\ g \to q \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{runDownTrans}\ tr\ q\ t = \mathit{run}\ (q, t)\ \textbf{where} \\
&\quad \mathit{run}\ (q, \mathit{In}\ t) = \mathit{appCxt}\ (\mathit{fmap}\ \mathit{run}\ (tr\ (q, t)))
\end{aligned}$$

A top-down transducer is run by applying its transduction function – $tr\ (q, t)$ – then recursively running the transformation in the holes of the produced context – $\mathit{fmap}\ \mathit{run}$ – and finally joining the context with the thus produced embedded terms – appCxt.

Example 6. We implement the DDTT from Example 5 in Haskell as follows:

$$\begin{aligned}
&\textbf{data}\ F\ a = \mathit{Or}\ a\ a \mid \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B \\
&\textbf{data}\ Q = \mathit{Q0} \mid \mathit{Q1} \\
&\\
&\mathit{trans} :: \mathit{DownTrans}\ F\ Q\ F \\
&\mathit{trans}\ (\mathit{Q0}, \mathit{TT}) = \mathit{tt}; \qquad \mathit{trans}\ (\mathit{Q0}, B) = b \\
&\mathit{trans}\ (\mathit{Q1}, \mathit{TT}) = \mathit{ff}; \qquad \mathit{trans}\ (\mathit{Q1}, B) = \mathit{not}\ b \\
&\mathit{trans}\ (\mathit{Q0}, \mathit{FF}) = \mathit{ff}; \qquad \mathit{trans}\ (\mathit{Q0}, \mathit{Not}\ x) = \mathit{Hole}\ (\mathit{Q1}, x) \\
&\mathit{trans}\ (\mathit{Q1}, \mathit{FF}) = \mathit{tt}; \qquad \mathit{trans}\ (\mathit{Q1}, \mathit{Not}\ x) = \mathit{Hole}\ (\mathit{Q0}, x) \\
&\mathit{trans}\ (\mathit{Q0}, \mathit{And}\ x\ y) = \mathit{Hole}\ (\mathit{Q0}, x)\ `\mathit{and}`\ \mathit{Hole}\ (\mathit{Q0}, y) \\
&\mathit{trans}\ (\mathit{Q1}, \mathit{And}\ x\ y) = \mathit{Hole}\ (\mathit{Q1}, x)\ `\mathit{or}`\ \ \mathit{Hole}\ (\mathit{Q1}, y) \\
&\mathit{trans}\ (\mathit{Q0}, \mathit{Or}\ \ x\ y) = \mathit{Hole}\ (\mathit{Q0}, x)\ `\mathit{or}`\ \ \mathit{Hole}\ (\mathit{Q0}, y) \\
&\mathit{trans}\ (\mathit{Q1}, \mathit{Or}\ \ x\ y) = \mathit{Hole}\ (\mathit{Q1}, x)\ `\mathit{and}`\ \mathit{Hole}\ (\mathit{Q1}, y)
\end{aligned}$$

The definition of the transduction function *trans* is a one-to-one translation of the transduction rules of the DDTT from Example 5. Note, that we use the smart constructors *or*, *and*, *not*, *tt*, *ff* and *b* on the right-hand side of the definitions. We apply the thus defined DDTT to a term of type *Term F* as follows:

$$\begin{aligned}&\mathit{negNorm} :: \mathit{Term}\ F \to \mathit{Term}\ F\\&\mathit{negNorm} = \mathit{runDownTrans}\ \mathit{trans}\ \mathit{Q0}\end{aligned}$$

5.3 Top-Down State Transition Functions

Unfortunately, we cannot provide a full decomposition of DDTTs into a state transition and a homomorphism part in the way we did for DUTTs in Section 4.5. Unlike in DUTTs, the state transition in a DDTT is inherently dependent on the transformation: since a placeholder variable may be copied on the right-hand side, each copy may be given a different successor state! For example, a DDTT may have a transduction rule

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

which transforms a function symbol f into g and copies the argument of f. However, the two copies are given different successor states, viz. q_1 and q_2.

In order to avoid this dependency of state transitions on the transformation, we restrict ourselves to DDTTs in which successor states are given to placeholder variables and not their occurrences. That is, for each two occurrences of subterms $q_1(x)$ and $q_2(x)$ on the right-hand side of a transduction rule, we require that $q_1 = q_2$. The DDTT given in Example 5 is, in fact, of this form.

The top-down state transitions we are aiming for are dual to bottom-up state transitions. The run of a bottom-up state transition function assigns a state to each node by an upwards state propagation, performing the same computation as an upwards accumulation [8]. The run of a top-down state transition function, on the other hand, should do the same by a downwards state propagation and thus perform the same computation as a downwards accumulation [9, 10].

However, representing top-down state transitions is known to be challenging [8–10]. A first attempt yields the type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$. This type, however, allows apart from the state transition also a transformation. The result is not required to have the same shape as the input. For example, the following equation (partially) defines a function *bad* of type $\forall\ a\ .\ (Q, \mathit{Sig}\ a) \to \mathit{Sig}\ Q$:

$$\mathit{bad}\ (q, \mathit{Plus}\ x\ y) = \mathit{Val}\ 1$$

In order to assign a successor state to each child of the input node without permitting changes to its structure, we use explicit placeholders to which we can assign the successor states:

$$\textbf{type}\ \mathit{DownState}\ f\ q = \forall\ i\ .\ \mathit{Ord}\ i \Rightarrow (q, f\ i) \to \mathit{Map}\ i\ q$$

The type *Map i q* represents finite mappings from type i to type q. Since such finite mappings are implemented by search trees, we require that the domain type i is of class *Ord*, which provides a total ordering.

The idea is to produce, from a state transition function of the above type, a function of type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$ that does preserve the structure of the input and only produces the successor states. This is achieved by injecting unique placeholders of type i into a value of type $f\ \ a$ – one for each child node. We can then produce the desired value of type $f\ \ q$ from the mapping of type $Map\ \ i\ \ q$ given by the state transition function. A placeholder that is not mapped to a state explicitly is assumed to keep the state of the current node by default.

To work with finite mappings, we assume an interface with $\emptyset$ denoting the empty mapping, $x \mapsto y$ the singleton mapping that maps x to y, $m \cup n$ the left-biased union of two mappings m and n, and a lookup function $lookup :: Ord\ \ i \Rightarrow i \to Map\ \ i\ \ q \to Maybe\ \ q$. Moreover, we define the lookup with default as follows:

$$
\begin{aligned}
&findWithDefault :: Ord\ \ i \Rightarrow q \to i \to Map\ \ i\ \ q \to q\\
&findWithDefault\ \ def\ \ i\ \ m = \mathbf{case}\ lookup\ \ i\ \ m\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad Nothing \to def\\
&\qquad\qquad\qquad\qquad\qquad\quad Just\ \ q\quad \to q
\end{aligned}
$$

At first, we need a mechanism to introduce unique placeholders into the structure of a functorial value. To this end, we will use the standard Haskell type class *Traversable* that provides the method

$$mapM :: (Traversable\ f, Monad\ m) \Rightarrow (a \to m\ \ b) \to f\ \ a \to m\ (f\ \ b)$$

which allows us to apply a monadic function to the components of a functorial value and then sequence the resulting monadic effects. Every polynomial functor can be made an instance of *Traversable*. Declarations to that effect can be derived automatically.

Ultimately, we want to number the elements in a functorial value to make them unique placeholders. To this end, we introduce a type of numbered values.

$$
\begin{aligned}
&\mathbf{newtype}\ Numbered\ \ a = Numbered\ (Int, a)\\
&unNumbered :: Numbered\ \ a \to a\\
&unNumbered\ (Numbered\ (\_, x)) = x\\
&\mathbf{instance}\ Eq\ (Numbered\ \ a)\ \mathbf{where}\\
&\quad Numbered\ (i, \_) \equiv Numbered\ (j, \_) = i \equiv j\\
&\mathbf{instance}\ Ord\ (Numbered\ \ a)\ \mathbf{where}\\
&\quad compare\ (Numbered\ (i, \_))\ (Numbered\ (j, \_)) = compare\ \ i\ \ j
\end{aligned}
$$

The instance declarations allow us to use elements of the type $Numbered\ \ a$ as placeholders.

With the help of the *mapM* combinator, we define a function that numbers the components in a functorial value by counting up using a state monad:

$$
\begin{aligned}
&number :: Traversable\ f \Rightarrow f\ \ a \to f\ (Numbered\ \ a)\\
&number\ \ x = fst\ (runState\ (mapM\ \ run\ \ x)\ 0)\ \mathbf{where}\\
&\quad run\ \ b = \mathbf{do}\ n \leftarrow get\\
&\qquad\qquad\quad put\ (n + 1)\\
&\qquad\qquad\quad return\ (Numbered\ (n, b))
\end{aligned}
$$

where $runState :: State\ s\ a \to s \to (a, s)$ runs a state monad with state type s, $put :: s \to State\ s\ m\ ()$ sets the state and $get :: State\ s\ s$ queries the state inside a state monad.

Using the above numbering combinator to create unique placeholders, we construct the explicit top-down propagation of states from a mapping of placeholders to successor states. Since the mapping of placeholders to successor states is partial, we also have to give a default state:

$$
\begin{aligned}
&appMap :: Traversable\ f \Rightarrow (\forall\ i\ .\ Ord\ i \Rightarrow f\ i \to Map\ i\ q) \\
&\qquad\qquad \to q \to f\ a \to f\ (q, a) \\
&appMap\ qmap\ q\ s = fmap\ qfun\ s'\ \textbf{where} \\
&\quad s' \quad\ = number\ s \\
&\quad qfun\ k = (findWithDefault\ q\ k\ (qmap\ s'), unNumbered\ k)
\end{aligned}
$$

Finally, we can combine a top-down state transition function with a stateful tree homomorphism by propagating the successor states using the *appMap* combinator. As the default state, we take the state of the current node, i.e. by default the state remains unchanged.

$$
\begin{aligned}
&downTrans :: Traversable\ f \Rightarrow DownState\ f\ q \to QHom\ f\ q\ g \\
&\qquad\qquad \to DownTrans\ f\ q\ g \\
&downTrans\ st\ f\ (q, s) = explicit\ f\ q\ fst\ (appMap\ (curry\ st\ q)\ q\ s) \\
&runDownHom :: (Traversable\ f, Functor\ g) \Rightarrow DownState\ f\ q \\
&\qquad\qquad \to QHom\ f\ q\ g \to q \to Term\ f \to Term\ g \\
&runDownHom\ st\ h = runDownTrans\ (downTrans\ st\ h)
\end{aligned}
$$

Note that we use the same type of stateful tree homomorphisms that we introduced for bottom-up state transitions. The roles of $?above$ and $?below$ are simply swapped: $?above$ refers to the state propagated from above whereas $?below$ provides the successor states of the current subterm. Stateful tree homomorphisms are ignorant of the direction in which the state is propagated.

Example 7. We reconstruct the DDTT from Example 6 by defining the state transition and the transformation separately:

$$
\begin{aligned}
&state :: DownState\ F\ Q \\
&state\ (Q0, Not\ x) = x \mapsto Q1 \\
&state\ (Q1, Not\ x) = x \mapsto Q0 \\
&state\ \_ \qquad\quad = \emptyset \\
&hom :: QHom\ F\ Q\ F \\
&hom\ TT \qquad = \textbf{if}\ above \equiv Q0\ \textbf{then}\ tt\ \textbf{else}\ ff \\
&hom\ FF \qquad = \textbf{if}\ above \equiv Q0\ \textbf{then}\ ff\ \textbf{else}\ tt \\
&hom\ B \qquad = \textbf{if}\ above \equiv Q0\ \textbf{then}\ b\ \ \textbf{else}\ not\ b \\
&hom\ (Not\ x) \quad = Hole\ x \\
&hom\ (And\ x\ y) = \textbf{if}\ above \equiv Q0\ \textbf{then}\ Hole\ x\ `and`\ Hole\ y \\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ Hole\ x\ `or`\ \ Hole\ y \\
&hom\ (Or\ x\ y) \quad = \textbf{if}\ above \equiv Q0\ \textbf{then}\ Hole\ x\ `or`\ \ Hole\ y \\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ Hole\ x\ `and`\ Hole\ y
\end{aligned}
$$

Note that in the definition of the state transition function, we return the empty mapping for all constructors different from *Not*. Consequently, the input state for these constructors is propagated unchanged by default.

By combining the state transition function and the stateful homomorphism, we obtain the same transformation function as in Example 6.

$$
\begin{aligned}
&\mathit{negNorm}' :: \mathit{Term}\ F \to \mathit{Term}\ F \\
&\mathit{negNorm}' = \mathit{runDownHom}\ \mathit{state}\ \mathit{hom}\ \mathit{Q0}
\end{aligned}
$$

Instead of introducing explicit placeholders in order to distribute the successor state, we could have also simply taken the encoding we first suggested, i.e. via a function ρ of type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$, and required as (an unchecked) side condition that ρ must preserve the shape of the input. This approach was taken in Gibbons' generic downwards accumulations [10] in which he requires the accumulation operation to be shape preserving.

Alternatively, we could have also adopted Gibbons' earlier approach to downwards accumulations [9], which instead represents the downward flow of information as a fold over a separately constructed data type called *path*. This path data type is constructed as the fixed point of a functor that is constructed from the signature functor. Unfortunately, this functor is quite intricate and not easy to program with in practice. Apart from that, it would be difficult to construct this path functor for each signature functor in Haskell.

In the end, our approach yields a straightforward representation of downward state transitions that is easy to work with in practise. Moreover, the ability to have a default behaviour for unspecified transitions makes for compact specifications as we have seen in Example 7. However, this default behaviour may also lead to errors more easily due to forgotten transitions.

5.4 Making Top-Down State Transition Functions Modular

Analogously to bottom-up state transition functions, we also define a variant of top-down state transition functions that has access to a bigger state space whose components are defined separately.

$$
\begin{aligned}
\textbf{type}\ \mathit{DDownState}\ f\ p\ q = \forall\ i\ .\ (&\mathit{Ord}\ i, ?\mathit{below} :: i \to p, ?\mathit{above} :: p, q \in p) \\
&\Rightarrow f\ i \to \mathit{Map}\ i\ q
\end{aligned}
$$

Translations between ordinary top-down state transitions and their generalised variants are produced as follows:

$$
\begin{aligned}
&\mathit{dDownState} :: \mathit{DownState}\ f\ q \to \mathit{DDownState}\ f\ p\ q \\
&\mathit{dDownState}\ f\ t = f\ (\mathit{above}, t) \\
&\mathit{downState} :: \mathit{DDownState}\ f\ q\ q \to \mathit{DownState}\ f\ q \\
&\mathit{downState}\ f\ (q, s) = \mathit{res}\ \textbf{where} \\
&\quad \mathit{res}\ \ = \mathit{explicit}\ f\ q\ \mathit{bel}\ s \\
&\quad \mathit{bel}\ k = \mathit{findWithDefault}\ q\ k\ \mathit{res}
\end{aligned}
$$

Similarly to their bottom-up counterparts, dependent top-down state transition functions that depend on the same state space can be combined to form a product state transition:

$$
\begin{aligned}
&(\circledast) :: (p \in c, q \in c) \Rightarrow \mathit{DDownState}\ f\ c\ p \to \mathit{DDownState}\ f\ c\ q \\
&\qquad\qquad \to \mathit{DDownState}\ f\ c\ (p, q) \\
&(\mathit{sp} \circledast \mathit{sq})\ t = \mathit{prodMap}\ \mathit{above}\ \mathit{above}\ (\mathit{sp}\ t)\ (\mathit{sq}\ t) \\
&\mathit{prodMap} :: \mathit{Ord}\ i \Rightarrow p \to q \to \mathit{Map}\ i\ p \to \mathit{Map}\ i\ q \to \mathit{Map}\ i\ (p, q)
\end{aligned}
$$

This construction is based on the pointwise product of mappings defined by *prodMap*, which we do not give in detail here. Since the mappings are partial, we have to provide a default state that is used in case only one of the mappings has a value for a given index. In accordance with the default behaviour of top-down state transition functions, this default state is the state from above.

As an example, we will define a transformation that replaces variables bound by let expressions with de Bruijn indices. For the sake of demonstration, we will implement this transformation using two states: the scope level, i.e. the number of let-bindings that are in scope, and a mapping from bound variables to the scope level of their respective binding site.

The scope level state simply counts the nesting of let bindings:

$$
\begin{aligned}
&\textbf{class}\ \mathit{ScopeLvlSt}\ f\ \textbf{where} \\
&\quad \mathit{scopeLvlSt} :: \mathit{DownState}\ f\ \mathit{Int} \\
&\textbf{instance}\ \mathit{ScopeLvlSt}\ \mathit{Let}\ \textbf{where} \\
&\quad \mathit{scopeLvlSt}\ (d, \mathit{LetIn}\ \_\ \_\ b) = b \mapsto (d + 1) \\
&\quad \mathit{scopeLvlSt}\ \_ \qquad\qquad\quad = \emptyset \\
&\textbf{instance}\ \mathit{ScopeLvlSt}\ f\ \textbf{where} \\
&\quad \mathit{scopeLvlSt}\ \_ \qquad\qquad\quad = \emptyset
\end{aligned}
$$

Here we use the fact that if a successor state is not defined for a subexpression, then the current state is propagated by default.

The state that maintains a mapping from variables to the scope level of their respective binding site is dependent on the scope level state:

$$
\begin{aligned}
&\textbf{type}\ \mathit{VarLvl} = \mathit{Map}\ \mathit{Name}\ \mathit{Int} \\
&\textbf{class}\ \mathit{VarLvlSt}\ f\ q\ \textbf{where} \\
&\quad \mathit{varLvlSt} :: \mathit{DDownState}\ f\ q\ \mathit{VarLvl} \\
&\textbf{instance}\ (\mathit{Int} \in q) \Rightarrow \mathit{VarLvlSt}\ \mathit{Let}\ q\ \textbf{where} \\
&\quad \mathit{varLvlSt}\ (\mathit{LetIn}\ v\ \_\ b) = b \mapsto ((v \mapsto \mathit{above}) \cup \mathit{above}) \\
&\quad \mathit{varLvlSt}\ \_ \qquad\qquad\quad = \emptyset \\
&\textbf{instance}\ \mathit{VarLvlSt}\ f\ q\ \textbf{where} \\
&\quad \mathit{varLvlSt}\ \_ = \emptyset
\end{aligned}
$$

Note that the first occurrence of *above* is of type *Int* – derived from the type constraint $\mathit{Int} \in q$ – whereas the second occurrence is of type *VarLvl* – derived from the type constraint $\mathit{VarLvl} \in q$ in the type $\mathit{DDownState}\ f\ q\ \mathit{VarLvl}$.

Since we want to replace explicit variables with de Bruijn indices, we have to replace the signature *Let* with the following signature in the output term:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Let}'\ e\ \ = \mathit{LetIn}'\ e\ e \mid \mathit{Var}'\ \mathit{Int} \\
&\textbf{type}\ \mathit{LetSig}' = \mathit{Let}' \oplus \mathit{Sig}
\end{aligned}
$$

The actual transformation is defined as a stateful tree homomorphism:

```
class DeBruijnHom f q g where
  deBruijnHom :: QHom f q g
instance (VarLvl ∈ q, Int ∈ q, Let' ≼ g) ⇒ DeBruijnHom Let q g where
  deBruijnHom (LetIn _ a b) = letIn' (Hole a) (Hole b)
  deBruijnHom (Var v)       = case lookup v above of
                                Nothing → error "free variable"
                                Just i → var' (above − i)
instance (Functor f, Functor g, f ≼ g) ⇒ DeBruijnHom f q g where
  deBruijnHom = simpCxt . inj
```

Note that we issue an error if we encounter a variable that is not bound by a let expression. Otherwise, we create the de Bruijn index by subtracting the variable's scope level from the current scope level.

Finally, we have to tie the components together by forming the product state transition and providing an initial state:

```
deBruijn :: Term LetSig → Term LetSig'
deBruijn = runDownHom stateTrans deBruijnHom init
  where init = (∅, 0) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

Due to its open definition, we can give the function *deBruijn* also the type $Term\ (Inc \oplus LetSig) \to Term\ (Inc \oplus LetSig')$, for example.

6 Bidirectional State Transitions

We have seen recursion schemes that use an upwards flow of information as well as recursion schemes that use a downwards flow of information. Some computations, however, require the combination of both. For example, if we want to extend the code generator from Section 4.6 to also work on let bindings, we need to propagate the generated code *upwards* but the symbol table for bound variables *downwards*.

In this section, we show two ways of achieving this combination.

6.1 Avoiding the Problem

The issue of combining two directions of information flow is usually circumvented by splitting up the computation in several runs instead. For the code generator, for instance, we can introduce a preprocessing step that translates let bindings into explicit assignments to memory addresses and variables into corresponding references to memory addresses.

This preprocessing step is easily implemented by modifying the stateful tree homomorphism from Section 5.4 that transforms variables into de Bruijn indices. Instead of de Bruijn indices we generate memory addresses.

At first, we define the signature that contains explicit addresses for bound variables:

```
data LetAddr e = LetAddr Addr e e | VarAddr Addr
type AddrSig   = LetAddr ⊕ Sig
```

The following stateful homomorphism then transforms a term over a signature containing *Let* into a signature containing *LetAddr* instead. The homomorphism depends on the same state as the de Bruijn homomorphism from Section 5.4:

```
class AddrHom f q g where
  addrHom :: QHom f q g
instance (VarLvl ∈ q, Int ∈ q, LetAddr ≼ g) ⇒ AddrHom Let q g where
  addrHom (LetIn _ x y) = letAddr above (Hole x) (Hole y)
  addrHom (Var v)       = case lookup v above of
                            Nothing → error "free variable"
                            Just a → varAddr a
instance (Functor f, Functor g, f ≼ g) ⇒ AddrHom f q g where
  addrHom = simpCxt . inj
```

By combining all components of the computation including the state transition functions *varLvlSt* and *scopeLvlSt* from Section 5.4, we obtain the desired transformation:

```
toAddr :: Addr → Term LetSig → Term AddrSig
toAddr startAddr = runDownHom stateTrans addrHom init
  where init = (∅, startAddr) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

The additional argument of type *Addr* allows us to control from which address we should start when assigning addresses to variables.

The actual code generation can then proceed on the signature *LetAddr* instead of *Let*:

```
instance CodeSt LetAddr q where
  codeSt (LetAddr a s e) = below s ++ [Store a] ++ below e
  codeSt (VarAddr a)     = [Load a]
```

To this end, we must also extend the *HeightSt* type class, which is used by the code generator:

```
instance HeightSt LetAddr where
  heightSt (LetAddr _ x y) = 1 + max x y
  heightSt (VarAddr _)     = 0
```

Now, we can use the function *code* from Section 4.6 with the type

```
code :: Term AddrSig → (Code, Addr)
```

Combining this function with the above defined transformation *toAddr*, yields the desired code generator:

```
codeLet :: Term LetSig → Code
codeLet t = c
  where t'          = toAddr (addr + 1) t
        (c, addr)   = code t'
```

When combining the two functions *toAddr* and *code*, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. To this end, we use the result *addr* of the code generator function *code*, which is the highest address used for intermediate results, to initialise the address counter for the transformation *toAddr*. This makes sure that we use different addresses for intermediate results and bound variables.

6.2 A Direct Implementation

An alternative approach performs the bottom-up and the top-down computations side-by-side, taking advantage of the non-strict semantics of Haskell. This approach avoids the construction of an intermediate syntax tree that contains the required information.

For implementing a suitable recursion scheme, we make use of the fact that both bottom-up as well as top-down state transition functions in their dependent form share the same interface to access other components of the state space via the implicit parameters ?*above* and ?*below*.

The following combinator runs a bottom-up and a top-down state transition function that both depend on the product of the state spaces they define:

```
runDState :: Traversable f ⇒ DUpState f (u, d) u
                           → DDownState f (u, d) d → d → Term f → u
runDState up down d (In t) = u where
  bel (Numbered (i, s)) =
    let d' = findWithDefault d (Numbered (i, ⊥)) qmap
    in Numbered (i, (runDState up down d' s, d'))
  t'    = fmap bel (number t)
  qmap  = explicit down (u, d) unNumbered t'
  u     = explicit up (u, d) unNumbered t'
```

The definition of *runDState* looks convoluted but follows a simple structure: the two lines at the bottom apply both state transition functions at the current node. To this end, the state from above and the state from below is given as (u, d) and *unNumbered*, respectively. The latter works as t' is computed by first numbering the child nodes and then using the numbering to lookup the successor states from *qmap* as well as recursively applying *runDState* at the child nodes.

Note that the definition of *runDState* is cyclic in several different ways and thus essentially depends on Haskell's non-strict semantics: the result u of the bottom-up state transition function is used also as input for the bottom-up state transition function. Likewise the result *qmap* of the top-down state transition function is fed into the construction of t', which is given as argument to the top-down state transition function. Moreover, the definition of both u and *qmap* depend on each other.

The above combinator allows us to write a code generator for the signature *LetSig* without resorting to an intermediate syntax tree. However, we have to be careful as this requires combining state transition functions with the same state space type: both *heightSt* and *scopeLvlSt* use the type *Int*.

However, the ambiguity can be easily resolved by "tagging" the types using *newtype* type synonyms. For the *scopeLvlSt* state transition, we define such a type like this:

```
newtype ScopeLvl = ScopeLvl { scopeLvl :: Int }
```

The tagging itself is a straightforward construction given the isomorphism between the type and its synonym in the form of a forward and a backward function:

```
tagDownState :: (q → p) → (p → q) → DownState f q → DownState f p
tagDownState i o t (q, s) = fmap i (t (o q, s))
```

We thus obtain a tagged variant of *scopeLvlSt*:

```
scopeLvlSt' :: ScopeLvlSt f ⇒ DownState f ScopeLvl
scopeLvlSt' = tagDownState ScopeLvl scopeLvl scopeLvlSt
```

The state maintained by *scopeLvlSt'* can now be accessed via the function *scopeLvl* in any compound state space containing *ScopeLvl*. A similar combinator can be defined for bottom-up state transitions.

Using the above state, we define a state transition function that assigns a memory address to each bound variable.

```
type VarAddr = Map Name Addr

class VarAddrSt f q where
  varAddrSt :: DDownState f q VarAddr

instance (ScopeLvl ∈ q) ⇒ VarAddrSt Let q where
  varAddrSt (LetIn v _ e) = e ↦ ((v ↦ scopeLvl above) ∪ above)
  varAddrSt _             = ∅

instance VarAddrSt f q where
  varAddrSt _ = ∅
```

Here, we use again overlapping instance declarations to give a uniform instance of *VarAddrSt* for all signatures different from *Let*.

We can now extend the type class *CodeSt* for the signature *Let*:

```
instance HeightSt Let where
  heightSt (LetIn _ x y) = 1 + max x y
  heightSt (Var _)       = 0

instance (ScopeLvl ∈ q, VarAddr ∈ q) ⇒ CodeSt Let q where
  codeSt (LetIn _ b e) = below b ++ [Store a] ++ below e
    where a = scopeLvl above
  codeSt (Var v)       = case lookup v above of
```

```
                                  Nothing → error "unbound variable"
                                  Just i  → [Load i]
```

Again, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. Similar to our implementation in Section 6.1, we use the output of the bottom-up state transition to obtain the maximum address used for storing intermediate results.

Thus, we tie the different components of the computation together as follows:

```
codeLet' :: Term LetSig → Code
codeLet' t = c
  where (c, addr) = runDState (codeSt ⊗ dUpState heightSt)
                              (varAddrSt ⊛ dDownState scopeLvlSt')
                              (∅ :: VarLvl, ScopeLvl (addr + 1)) t
```

Note that in both implementations, we could have avoided the use of the result of the state transition function *heightSt* to initialise the address counter for bound variables. The modularity of our recursion schemes makes it possible to replace the *heightSt* state transition function with a different one. In this way, we could avoid clashes by using even address numbers for intermediate results and odd address numbers for variables.

We already observed that stateful tree homomorphisms cannot discern the direction in which the state is propagated. Thus we can supply them with a state using either bottom-up or top-down state transitions. In fact, following the bidirectional state transitions we considered above, we can provide a stateful tree homomorphism with a combined state given by both a bottom-up and a top-down state transition function. Such a transformation can for example be used to rename apart all bound variables or inline simple let bindings.

7 Discussion

We have seen that with some adjustments tree automata can be turned into highly modular recursion schemes. These recursion schemes allow us to take advantage of two orthogonal dimensions of modularity: modularity in the state that is propagated and – courtesy of Swierstra's [23] *data types à la carte* – modularity in the structure of terms. In addition to that, we also showed how to decompose transducers into a homomorphism and into a state transition part. This high level of modularity makes our automata-based recursion schemes especially valuable for constructing modular compilers as we have illustrated in our running example. However, we should point out that there are many more aspects to consider when constructing compilers in a modular fashion [4].

The dependent forms of bottom-up and top-down state transitions that we have developed in this paper are nothing else than the *synthesised* and *inherited attributes* known from *attribute grammars* [22]. In fact, the combinator *runDState* that runs both a bottom-up and a top-down state transition can be seen as a run of an attribute grammar with corresponding synthesised and inherited attributes. Viera et al. [24] have developed a Haskell library that allows to specify such attribute grammars in Haskell in a very concise way.

We also obtain an added value by using a powerful functional language for the embedding of our recursion schemes. One immediate benefit that we obtain is the use of further generic programming techniques. For example, the *heightSt* state transition function could have been defined entirely generically, without having to extend the definition for every new signature.

7.1 Why Tree Transducers?

In principle, tree transducers offer no increase in expressiveness over (dependent) bottom-up state transition functions since we allow for infinite state spaces anyway. However, due to their additional structure they provide at least two advantages.

First of all, tree transducers are very flexible in the way they can be manipulated in order to form new transformations. For example, we can extend a given signature functor f with annotations of some type a by using the construction

$$\textbf{data}\ (f :\&: a)\ e = f\ e :\&: a$$

A term over the signature $(f :\&: a)$ is similar to a term over f but it additionally contains annotations of type a at every subterm. We can provide a combinator that modifies a tree transducer from F to G into one from $F :\&: A$ to $G :\&: A$ that propagates the annotations from the input term to the output term [1].

Secondly, tree transducers can be composed. That is, given two bottom-up (respectively top-down) tree transducers – one from F to G, the other one from G to H, we can generically construct a bottom-up (respectively top-down) transducer from F to H whose transformation is equal to the composition of the transformations denoted by the original transducers [7]. The resulting transducer then only has to traverse the input term once and avoids the construction of the intermediate term [26]. Note that tree homomorphisms can be considered both a special case of bottom-up and of top-down tree transducers and can thus be composed with either kind.

The two abovementioned features also set tree transducers apart from other generic programming approaches such as *Scrap your Boilerplate* [12, 13, 18] or *Uniplate* [21]. We do not give the full technical details of the two features here but the implementation can be found in the `compdata` package [2].

7.2 Extensions & Future Work

While we only considered single recursive data types, this restriction is not essential: following the construction of Yakushev et al. [27] and Bahr and Hvitved [1], our recursion schemes can be readily extended to work on mutually recursive data types as well.

Note that the *runDState* combinator of Section 6.2 constructs the product of the two state spaces u and d. Consequently, if u is a compound state space, we obtain a product type that is not a right-associative nesting of pairs which we require for the type class $\in$ to work properly. However, this can be remedied by a more clever encoding of compound state spaces as *heterogeneous lists* [14] or generating instance declarations for products of a limited number of components via *Template Haskell*.

The transducers that we have considered here have one severe limitation. This limitation can be seen when looking at the implementation of these transducers in Haskell: the parametric polymorphism of the type for placeholder variables prevents us from using these placeholder variables in the state transition. This would allow us to store and retrieve subterms that the placeholder variables are instantiated with. The ability to do that is necessary in order to perform "non-local" transformations such as inlining of arbitrary let bindings or applying substitutions. However, we can remedy this issue by making the state a functor. The type of bottom-up respectively top-down transducers would then look as follows:

$$\begin{aligned}&\textbf{type } \mathit{UpTrans} \quad f\ q\ g = \forall\ a\ .\ f\ (q\ a, a) \to (q\ (\mathit{Context}\ g\ a), \mathit{Context}\ g\ a)\\ &\textbf{type } \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to \mathit{Context}\ g\ (q\ (\mathit{Context}\ g\ a), a)\end{aligned}$$

We can then, for example, instantiate q with $\mathit{Map}\ \mathit{Var}$ such that the state is a substitution, i.e. a mapping from variables to terms (respectively term placeholders).

The above types represent a limited form of macro tree transducers [7]. Wile the decomposition of such an extended bottom-up transducer into a homomorphism and a state transition function is again straightforward, the decomposition of an extended top-down transducer is trickier: at least the representation with explicit placeholders that we used for dependent top-down state transition functions does not straightforwardly generalise to polymorphic states.

Note that the abovementioned limitation only affects transducers, not state transition functions. We can, of course, implement inlining and substitution as a bidirectional state transition. However, if we want to make use of the nice properties of transducers, we have to move to the extended tree transducers illustrated above.

Acknowledgements

The author would like to thank Tom Hvitved, Jeremy Gibbons, Wouter Swierstra, Doaitse Swierstra and the anonymous referees for valuable comments, corrections, suggestions for improvements and pointers to the literature.

Bibliography

[1] P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, New York, NY, USA, 2011. ACM. doi: 10.1145/2036918.2036930.

[2] P. Bahr and T. Hvitved. Compdata Haskell library, 2012. module `Data.Comp.Automata`, available from `http://hackage.haskell.org/package/compdata`.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on `http://www.grappa.univ-lille3.fr/tata`, 2008.

[4] L. Day and G. Hutton. Towards Modular Compilers For Effects. In *Proceedings of the Symposium on Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, Madrid, Spain, 2011. Springer-Verlag. doi: 10.1007/978-3-642-32037-8˙4.

[5] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11(4):452–470, 1967. ISSN 0019-9958. doi: 10.1016/S0019-9958(67)90670-5.

[6] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 7500 AE Enschede, Netherlands, 1992.

[7] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer-Verlag New York, Inc., 1998. ISBN 3540646078.

[8] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56625-2˙11.

[9] J. Gibbons. Polytypic downwards accumulations. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 207–233. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64591-7. doi: 10.1007/BFb0054292.

[10] J. Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37(1-3):37–65, 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00022-2.

[11] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical Views on Computations on Trees (Extended Abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 619–630. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73420-8˙54.

[12] R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In *Proceedings of the Eighth International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208, Berlin, Heidelberg, 2006. Springer-Verlag. doi: 10.1007/11783596˙13.

[13] R. Hinze, A. Löh, and B. C. d. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin Heidelberg, 2006. doi: 10.1007/11737414˙3.

[14] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004. doi: 10.1145/1017472.1017488.

[15] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968. ISSN 1432-4350. doi: 10.1007/BF01692511.

[16] A. Kühnemann. Benefits of Tree Transducers for Optimizing Functional Programs. In V. Arvind and S. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, page 1046. Springer Berlin / Heidelberg, 1998. doi: 10.1007/978-3-540-49382-2˙13.

[17] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–118, New York, NY, USA, 2000. ACM. doi: 10.1145/325694.325708.

[18] R. Lämmel and S. P. Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 204–215, New York, NY, USA, 2005. ACM. doi: 10.1145/1086365.1086391.

[19] S. Marlow. Haskell 2010 Language Report, 2010.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961˙7.

[21] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 49–60, New York, NY, USA, 2007. ACM. doi: 10.1145/1291201.1291208.

[22] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995. ISSN 0360-0300. doi: 10.1145/210376.197409.

[23] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758.

[24] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming - ICFP '09*, page 245, New York, New York, USA, 2009. ACM Press. doi: 10.1145/1596550.1596586.

[25] J. Voigtländer. Formal Efficiency Analysis for Tree Transducer Composition. *Theory of Computing Systems*, 41(4):619–689, 2007. ISSN 1432-4350. doi: 10.1007/s00224-006-1235-9.

[26] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990. doi: 10.1016/0304-3975(90)90147-A.

[27] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596585.

Domain-Specific Languages for Enterprise Systems

Tom Hvitved Patrick Bahr Jesper Andersen

Department of Computer Science, University of Copenhagen

Abstract

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. is a novel software architecture for enterprise resource planning (ERP) systems. POETS employs a pragmatic separation between (i) transactional data, that is what has happened; (ii) reports, that is what can be derived from the transactional data; and (iii) contracts, that is which transactions are expected in the future. Moreover, POETS applies domain-specific languages (DSLs) for specifying reports and contracts, in order to enable succinct declarative specifications as well as rapid adaptability and customisation. In this report we document an implementation of a generalised and extended variant of the POETS architecture. The generalisation is manifested in a detachment from the ERP domain, which is rather an instantiation of the system than a built-in assumption. The extensions amount to a customisable data model based on nominal subtyping; support for run-time changes to the data model, reports and contracts, while retaining full auditability; and support for referable data that may evolve over time, also while retaining full auditability as well as referential integrity. Besides the revised architecture, we present the DSLs used to specify data definitions, reports, and contracts respectively, and we provide the complete specification for a use case scenario, which demonstrates the conciseness and validity of our approach. Lastly, we describe technical aspects of our implementation, with focus on the techniques used to implement the tightly coupled DSLs.

Contents

1 Introduction

Enterprise resource planning (ERP) systems are comprehensive software systems used to manage daily activities in enterprises. Such activities include—but are not limited to—financial management (accounting), production planning, supply chain management and customer relationship management. ERP systems emerged as a remedy to heterogeneous systems, in which data and functionality are spread out—and duplicated—amongst dedicated subsystems. Instead, an ERP system it built around a central database, which stores all information in one place.

Traditional ERP systems such as Microsoft Dynamics NAV[1], Microsoft Dynamics AX[2], and SAP[3] are three-tier architectures with a client, an application server, and a centralised relational database system. The central database stores information in tables, and the application server provides the business logic, typically coded in a general purpose, imperative programming language. A shortcoming to this approach is that the state of the system is decoupled from the business logic, which means that business processes—that is, the daily activities—are not represented explicitly in the system. Rather, business processes are encoded implicitly as transition systems, where the state is maintained by tables in the database, and transitions are encoded in the application server, possibly spread out across several different logical modules.

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. [6] is a qualitatively different approach to ERP systems. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between transactional data, which is persisted in an *event log*, and *contracts*, which are explicit representations of business processes, stored in a separate module. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises a declarative domain-specific language (DSL) [1]. The use of a DSL not only enables explicit formalisation of business processes, it also minimises the gap between requirements and a running system. In fact, Henglein et al. take it as a goal of POETS that "[...] the formalized requirements *are* the system" [6, page 382].

The bird's-eye view of the POETS architecture is presented in Figure 1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent "things that take place" such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items in an inventory. The append-only restriction serves two purposes. First, it is a legal requirement in ERP systems that transactions, which are relevant for auditing, are retained. Second, the report engine utilises monotonicity of the event log for optimisation, as shown by Nissen and Larsen [12].

[1] http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx.
[2] http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx.
[3] http://www.sap.com.

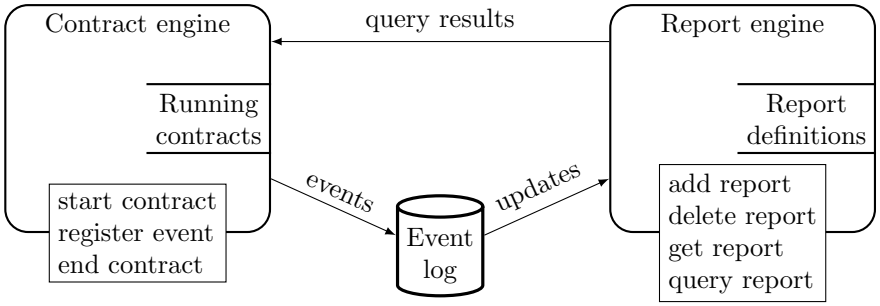


Figure 1: Bird's-eye view of the POETS architecture (diagram copied from [6]).

Whereas the event log stores historical data, contracts play the role of describing which events are expected in the future. For instance, a yearly payment of value-added tax (VAT) to the tax authorities is an example of a (recurring) business process. The amount to be paid to the tax authorities depends, of course, on the financial transactions that have taken place. Therefore, information has to be derived from previous transactions in the event log, which is realised as a *report*. A report provides structured data derived from the transactions in the event log. Like contracts, reports are written in a declarative domain-specific language—not only in order to minimise the semantic gap between requirements and the running system, but also in order to perform automatic optimisations.

Besides the radically different software architecture, POETS distinguishes itself from existing ERP systems by abandoning the double-entry bookkeeping (DEB) accounting principle [17] in favour of the resources, events, and agents (REA) accounting model of McCarthy [10].

In double-entry bookkeeping, each transaction is recorded as two postings in a *ledger*—a *debit* and a *credit*. When, for instance, a customer pays an amount x to a company, then a debit of x is posted in a cash account, and a credit of x is posted in a sales account, which reflects the flow of cash from the customer to the company. The central invariant of DEB is that the total credit equals the total debit—if not, resources have either vanished or spontaneously appeared. DEB fits naturally in the relational database oriented architectures, since each ledger is similar in structure to a table. Moreover, DEB is the de facto standard accounting method, and therefore used by current ERP systems.

In REA, transactions are not registered in accounts, but rather as the events that take place. An event in REA is of the form (a_1, a_2, r) meaning that agent a_1 transfers resource r to agent a_2. Hence, when a customer pays an amount x to a company, then it is represented by a single event $(\text{customer}, \text{company}, x)$. Since events are atomic, REA does not have the same redundancy[4] as DEB, and inconsistency is not a possibility: resources always have an origin and a destination. The POETS architecture not only fits with the REA ontology, it is based on it. Events are stored as first-class objects in the event log, and contracts describe the expected future flow of resources.[5] Reports enable computation of

[4]In traditional DEB, redundancy is a feature to check for consistency. However, in a computer system such redundancy is superfluous.

[5]Structured contracts are not part of the original REA ontology but instead due to Andersen et al. [1].

derived information that is inherent in DEB, and which may be a legal requirement for auditing. For instance, a sales account—which summarises (pending) sales payments—can be reconstructed from information about initiated sales and payments made by customers. Such a computation will yield the same *derived* information as in DEB, and the principles of DEB consistency will be fulfilled simply by construction.

1.1 Outline and Contributions

The motivation for our work is to assess the POETS architecture in terms of a prototype implementation. During the implementation process we have added features to the architecture that were painfully missing. Moreover, in the process we found that the architecture need not be tailored to the REA ontology—indeed to ERP systems—but the applicability of our generalised architecture to other domains remains future research. Our contributions are as follows:

- We present a generalised and extended POETS architecture (Section 2) that has been fully implemented.
- We present domain-specific languages for data modelling (Section 2.1), report specification (Section 2.4), and contract specification (Section 2.5).
- We demonstrate how to implement a small use case, from scratch, in our implemented system (Section 3). We provide the complete specification of the system, which demonstrates both the conciseness and domain-orientation[6] of our approach. We conclude that the extended architecture is indeed well-suited for implementing ERP systems—although the DSLs and the data model may require additions for larger systems. Most notably, the amount of code needed to implement the system is but a fraction of what would be have to be implemented in a standard ERP system.
- We describe how we have utilised state-of-the art software development tools in our implementation, especially how the tightly coupled DSLs are implemented (Section 4).

2 Revised POETS Architecture

Our generalised and extended architecture is presented in Figure 2. Compared to the original architecture in Figure 1, the revised architecture sees the addition of three new components: a *data model*, an *entity store*, and a *rule engine*. The rule engine is currently not implemented, and we will therefore not return to this module until Section 5.1.

As in the original POETS architecture, the event log is at the heart of the system. However, in the revised architecture the event log plays an even greater role, as it is the *only* persistent state of the system. This means that the states of all other modules are also persisted in the event log, hence the flow of information from all other modules to the event log in Figure 2. For example, whenever a

[6] Compare the motto: "[...] the formalized requirements *are* the system" [6, page 382].

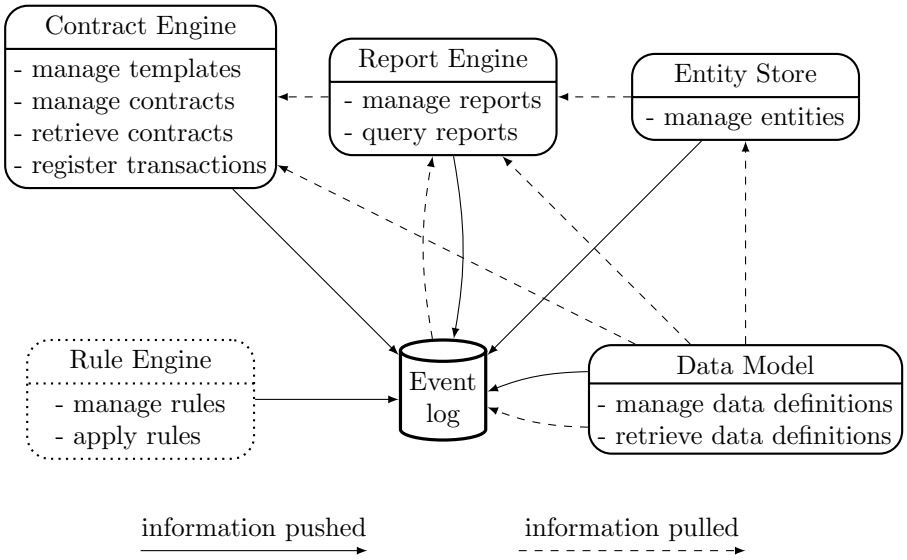


Figure 2: Bird's-eye view of the generalised and extended POETS architecture.

contract is started or a new report is added to the system, then an event reflecting this operation is persisted in the event log. This, in turn, means that the state of each module can—in principle—be derived from the event log. However, for performance reasons each module—including the event log—maintains its own state in memory.

The addition of a data model constitutes the generalisation of the new architecture over the old architecture. In the data model, data definitions can be added to the system—at run-time—such as data defining customers, resources, or payments. Therefore, the system is not a priori tailored to ERP systems or the REA ontology, but it can be instantiated to that, as we shall see in Section 3.

The entity store is added to the architecture in order to support *entities*—unique "objects" with associated data that may evolve over time. For instance, a concrete customer can suitably be modelled as an entity: Although information attributed to that customer—such as address, or even name—are likely to change over time, it is still the same customer. Moreover, we do not want a copy of the customer data in for instance a sale, but rather a reference to that customer. Hence by modelling customers as entities, we are able to derive, for instance, all transactions in which that customer has participated—even if the customer attributes have changed over time.

We describe each module of the revised architecture in the following subsections. Since we will focus on the revised architecture in the remainder of the text, we will refer to said architecture simply as POETS.

2.1 Data Model

The data model is a core component of the extended architecture, and the interface it provides is summarised in Figure 3. The data model defines the *types* of data that are used throughout the system, and it includes predefined types such as events. Custom types such as invoices can be added to the data model at run-

Data Model		
Function	**Input**	**Output**
addDataDefs	ontology specification	
getRecordDef	record name	type definition
getSubTypes	record name	list of record names

Figure 3: Data model interface.

time via *addDataDefs*—for simplicity, we currently only allow addition of types, not updates and deletions. Types define the structure of the data in a running POETS instance manifested as *values*. A value—such as a concrete invoice—is an instance of the data specified by a type. Values are not only communicated between the system and its environment but they are also stored in the event log, which is simply a list of values of a certain type.

2.1.1 Types

Structural data such as payments and invoices are represented as *records*, that is typed finite mappings from field labels to values. Record types define the structure of such records by listing the constituent field labels and their associated types. In order to form a hierarchical ontology of record types, we use a nominal subtyping system [14]. That is, each record type has a unique name, and one type is a subtype of another if and only if stated so explicitly or by transitivity. For instance, a customer can be defined as a subtype of a person, which means that a customer contains all the data of a person, similar to inheritance in object oriented programming.

The choice of nominal types over structural types [14] is justified by the domain: the nominal type associated with a record may have a semantic impact. For instance, the type of customers and premium customers may be structurally equal, but a value of one type is considered different from the other, and clients of the system may for example choose to render them differently. Moreover, the purpose of the rule engine, which we return to in Section 5.1, is to define rules for values of a particular semantic domain, such as invoices. Hence it is wrong to apply these rules to data that happens to have the same structure as invoices. Although we use nominal types to classify data, the DSLs support full record polymorphism [13] in order to minimise code duplication. That is, it is possible for instance to use the same piece of code with customers and premium customers, even if they are not related in the subtyping hierarchy.

The grammar for types is as follows:

$$
\begin{aligned}
T ::= {} & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \mathbf{String} \mid \mathbf{Timestamp} \mid \mathbf{Duration} && \text{(type constants)} \\
\mid {} & \mathit{RecordName} && \text{(record type)} \\
\mid {} & [T] && \text{(list type)} \\
\mid {} & \langle \mathit{RecordName} \rangle && \text{(entity type)}
\end{aligned}
$$

Type constants are standard types Booleans, integers, reals, and strings, and less standard types timestamps (absolute time) and durations (relative time). Record types are named types, and the record typing environment—which we will

describe shortly—defines the structure of records. For record types we assume a set $\mathit{RecordName} = \{\mathsf{Customer}, \mathsf{Address}, \mathsf{Invoice}, \dots\}$ of record names ranged over by r. Concrete record types are typeset in sans-serif, and they always begin with a capital letter. Likewise, we assume a set *FieldName* of all field names ranged over by f. Concrete field names are typeset in sans-serif beginning with a lower-case letter.

List types $[\tau]$ represent lists of values, where each element has type τ, and it is the only collection type currently supported. Entity types $\langle r \rangle$ represent entity values that have associated data of type r. For instance, if the record type Customer describes the data of a customer, then a value of type $\langle \mathsf{Customer} \rangle$ is a (unique) customer entity, whose associated Customer data may evolve over time. The type system ensures that a value of an entity type in the system will have associated data of the given type, similar to referential integrity in database systems [3]. We will return to how entities are created and modified in Section 2.3.

A *record typing environment* provides the record types that are available, their subtype relation, and the fields they define.

Definition 2.1. A record typing environment is a tuple $(R, A, F, \rho, \leq)$ consisting of finite sets $R \subseteq \mathit{RecordName}$ and $F \subseteq \mathit{FieldName}$, a set $A \subseteq R$, a mapping $\rho : R \to \mathcal{P}_{\text{fin}}(F \times T)$, and a relation $\leq \subseteq R \times R$, where $\mathcal{P}_{\text{fin}}(X)$ denotes the set of all finite subsets of a set X.

Intuitively, R is the set of defined record types, ρ gives for each defined record type its fields and their types, $\leq$ gives the subtyping relation between record types, and record types in A are considered to be abstract. Abstract record types are not supposed to be instantiated, they are only used to structure the record type hierarchy. The functions *getRecordDef* and *getSubTypes* from Figure 3 provide the means to retrieve the record typing environment from a running system.

Record types can depend on other record types by having them as part of the type of a constituent field:

Definition 2.2. The *immediate dependency relation* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$, denoted $\to_{\mathcal{R}}$, is the binary relation on R such that $r_1 \to_{\mathcal{R}} r_2$ iff there is some $(f, \tau) \in \rho(r_1)$ such that a record name r occurs in τ with $r_2 \leq r$. The *dependency relation* $\to_{\mathcal{R}}^{+}$ of $\mathcal{R}$ is the transitive closure of $\to_{\mathcal{R}}$.

We do not permit all record typing environments. Firstly, we do not allow the subtyping to be cyclic, that is a record type r cannot have a proper subtype which has r as a subtype. Secondly, the definition of field types must be unique and must follow the subtyping, that is a subtype must define at least the fields of its supertypes. Lastly, we do not allow recursive record type definitions, that is a cycle in the dependency relation. The two first restrictions are sanity checks, but the last restriction makes a qualitative difference: the restriction is imposed for simplicity, and moreover we have not encountered practical situations where recursive types were needed.

Definition 2.3. A record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is *well-formed*, whenever the following is satisfied:

- $\leq$ is a partial order, (acyclic inheritance)
- each $\rho(r)$ is the graph of a partial function $F \rightharpoonup T$, (unique typing)
- $r_1 \leq r_2$ implies $\rho(r_1) \supseteq \rho(r_2)$, and (consistent typing)
- $\to_{\mathcal{R}}^{+}$ is irreflexive, that is $r_1 \to_{\mathcal{R}}^{+} r_2$ implies $r_1 \neq r_2$. (non-recursive)

Well-formedness of a record typing environment combines both conditions for making it easy to reason about them—for instance, transitivity of $\leq$ and inclusion of fields of supertypes—and hard restrictions such as non-recursiveness and unique typing. If a record typing environment fails to be well-formed due to the former only, it can be uniquely closed to a well-formed one:

Definition 2.4. The *closure* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is the record typing environment $\mathrm{Cl}(\mathcal{R}) = (R, A, F, \rho', \leq')$ such that $\leq'$ is the transitive, reflexive closure of $\leq$ and ρ' is the consistent closure of ρ with respect to $\leq'$, that is $\rho'(r) = \bigcup_{r \leq' r'} \rho(r')$.

The definition of closure allows us to easily build a well-formed record typing environment from an incomplete specification.

Example 2.5. As an example, we may define a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ for persons and customers as follows:

$$\begin{aligned} R &= \{\mathsf{Person}, \mathsf{Customer}, \mathsf{Address}\} & \rho(\mathsf{Person}) &= \{(\mathsf{name}, \mathbf{String})\} \\ A &= \{\mathsf{Person}\} & \rho(\mathsf{Customer}) &= \{(\mathsf{address}, \mathsf{Address})\} \\ F &= \{\mathsf{name}, \mathsf{address}, \mathsf{road}, \mathsf{no}\} & \rho(\mathsf{Address}) &= \{(\mathsf{road}, \mathbf{String}), (\mathsf{no}, \mathbf{Int})\}, \end{aligned}$$

with $\mathsf{Customer} \leq \mathsf{Person}$. The only properties that prevent $\mathcal{R}$ from being well-formed are the missing field typing $(\mathsf{name}, \mathbf{String})$ that $\mathsf{Customer}$ should inherit from Person and the missing reflexivity of $\leq$. Hence, the closure $\mathrm{Cl}(\mathcal{R})$ of $\mathcal{R}$ is indeed a well-formed record typing environment.

In order to combine record typing environments we define the union $\mathcal{R}_1 \cup \mathcal{R}_2$ of two record typing environments $\mathcal{R}_i = (R_i, A_i, F_i, \rho_i, \leq_i)$ as the pointwise union:

$$\mathcal{R}_1 \cup \mathcal{R}_2 = (R_1 \cup R_2, A_1 \cup A_2, F_1 \cup F_2, \rho_1 \cup \rho_2, \leq_1 \cup \leq_2),$$

where $(\rho_1 \cup \rho_2)(r) = \rho_1(r) \cup \rho_2(r)$ for all $r \in R_1 \cup R_2$. Note that the union of two well-formed record typing environments need not be well-formed—either due to incompleteness, which can be resolved by forming the closure of the union, or due to inconsistencies respectively cyclic dependencies, which cannot be resolved.

2.1.2 Values

The set of values *Value* supplementing the types from the previous section is defined inductively as the following disjoint union:

$$\mathit{Value} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Record} \uplus \mathit{List} \uplus \mathit{Ent},$$

with

$$\begin{aligned} \mathit{Bool} &= \{\mathit{true}, \mathit{false}\} & \mathit{String} &= \mathit{Char}^* & \mathit{Record} &= \mathit{RecordName} \times \mathit{Fields} \\ \mathit{Int} &= \mathbb{Z} & \mathit{Timestamp} &= \mathbb{N} & \mathit{Fields} &= \mathit{FieldName} \rightharpoonup_{\text{fin}} \mathit{Value} \\ \mathit{Real} &= \mathbb{R} & \mathit{Duration} &= \mathbb{Z} & \mathit{List} &= \mathit{Value}^*, \end{aligned}$$

where X^* denotes the set of all finite sequences over a set X; *Char* is a set of characters; *Ent* is an abstract, potentially infinite set of entity values; and $A \rightharpoonup_{\text{fin}} B$ denotes the set of finite partial mappings from a set A to a set B.

Timestamps are modelled using UNIX time[7] and durations are measured in seconds. A record $(r, m) \in \mathit{Record}$ consists of a record name $r \in \mathit{RecordName}$ together with a finite set of named values $m \in \mathit{Fields}$. Entity values $e \in \mathit{Ent}$ are abstract values that only permit equality testing and dereferencing—the latter takes place only in the report engine (Section 2.4), and the type system ensures that dereferencing cannot get stuck, as we shall see in the following subsection.

Example 2.6. As an example, a customer record value $c \in \mathit{Record}$ may be as follows:

$$\begin{aligned} c &= (\mathsf{Customer}, m) & m'(\mathsf{road}) &= \texttt{Universitetsparken} \\ m(\mathsf{name}) &= \texttt{John Doe} & m'(\mathsf{no}) &= 1, \\ m(\mathsf{address}) &= (\mathsf{Address}, m') \end{aligned}$$

where $m, m' \in \mathit{Fields}$.

2.1.3 Type Checking

All values are type checked before they enter the system, both in order to check that record values conform with the record typing environment, but also to check that entity values have valid associated data. In particular, events—which are values—are type checked before they are persisted in the event log. In order to type check entities, we assume an *entity typing environment* $\mathcal{E} : \mathit{Ent} \rightharpoonup_{\text{fin}} \mathit{RecordName}$, that is a finite partial mapping from entities to record names. Intuitively, an entity typing environment maps an entity to the record type that it has been declared to have upon creation.

The typing judgement has the form $\mathcal{R}, \mathcal{E} \vdash v : \tau$, where $\mathcal{R}$ is a well-formed record typing environment, $\mathcal{E}$ is an entity typing environment, $v \in \mathit{Value}$ is a value, and $\tau \in T$ is a type. The typing judgment uses the auxiliary subtyping judgement $\mathcal{R} \vdash \tau_1 <: \tau_2$, which is a generalisation of the subtyping relation from Section 2.1.1 to arbitrary types.

The typing rules are given in Figure 4. The typing rules for base types and lists are standard. In order to type check a record, we need to verify that the record contains all and only those fields that the record typing environment prescribes, and that the values have the right type. The typing rule for entities uses the entity typing environment to check that each entity has associated data, and that the data has the required type. The last typing rule enables values to be coerced to a supertype in accordance with the subtyping judgement, which is

[7] http://en.wikipedia.org/wiki/Unix_time.

$$\boxed{\mathcal{R}, \mathcal{E} \vdash v : \tau}$$

$$\frac{b \in \mathit{Bool}}{\mathcal{R}, \mathcal{E} \vdash b : \mathbf{Bool}} \qquad \frac{n \in \mathit{Int}}{\mathcal{R}, \mathcal{E} \vdash n : \mathbf{Int}} \qquad \frac{r \in \mathit{Real}}{\mathcal{R}, \mathcal{E} \vdash r : \mathbf{Real}}$$

$$\frac{s \in \mathit{String}}{\mathcal{R}, \mathcal{E} \vdash s : \mathbf{String}} \qquad \frac{t \in \mathit{Timestamp}}{\mathcal{R}, \mathcal{E} \vdash t : \mathbf{Timestamp}} \qquad \frac{d \in \mathit{Duration}}{\mathcal{R}, \mathcal{E} \vdash d : \mathbf{Duration}}$$

$$\frac{\begin{array}{cc} \mathcal{R} = (R, A, F, \rho, \leq) & \mathrm{dom}(\rho(r)) = \mathrm{dom}(m) \\ (r, m) \in \mathit{Record} \qquad r \in R \setminus A & \forall f \in \mathrm{dom}(m) \colon \mathcal{R}, \mathcal{E} \vdash m(f) : \rho(r)(f) \end{array}}{\mathcal{R}, \mathcal{E} \vdash (r, m) : r}$$

$$\frac{(v_1, \ldots, v_n) \in \mathit{List} \qquad \forall i \in \{1, \ldots, n\}. \mathcal{R}, \mathcal{E} \vdash v_i : \tau}{\mathcal{R}, \mathcal{E} \vdash (v_1, \ldots, v_n) : [\tau]} \qquad \frac{e \in \mathit{Ent} \qquad \mathcal{E}(e) = r}{\mathcal{R}, \mathcal{E} \vdash e : \langle r \rangle}$$

$$\frac{\mathcal{R}, \mathcal{E} \vdash v : \tau' \qquad \mathcal{R} \vdash \tau' <: \tau}{\mathcal{R}, \mathcal{E} \vdash v : \tau}$$

$$\boxed{\mathcal{R} \vdash \tau_1 <: \tau_2}$$

$$\frac{}{\mathcal{R} \vdash \tau <: \tau} \qquad \frac{\mathcal{R} \vdash \tau_1 <: \tau_2 \qquad \mathcal{R} \vdash \tau_2 <: \tau_3}{\mathcal{R} \vdash \tau_1 <: \tau_3}$$

$$\frac{}{\mathcal{R} \vdash \mathbf{Int} <: \mathbf{Real}} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash r_1 <: r_2}$$

$$\frac{\mathcal{R} \vdash \tau_1 <: \tau_2}{\mathcal{R} \vdash [\tau_1] <: [\tau_2]} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash \langle r_1 \rangle <: \langle r_2 \rangle}$$

Figure 4: Type checking of values $\mathcal{R}, \mathcal{E} \vdash v : \tau$ and subtyping $\mathcal{R} \vdash \tau_1 <: \tau_2$.

also given in Figure 4. The rules for the subtyping relation extend the relation from Section 2.1.1 to include subtyping of base types, and contextual rules for lists and entities. We remark that the type system in Figure 4 is declarative: in our implementation, an equivalent algorithmic type system is used.

Example 2.7. Reconsider the record typing environment $\mathcal{R}$ and its closure $\mathrm{Cl}(\mathcal{R})$ from Example 2.5, and the record value c from Example 2.6. Using the typing rules in Figure 4, we can derive the typing judgement $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Customer}$ for any entity typing environment $\mathcal{E}$. Moreover, since Customer is a subtype of Person we also have that $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Person}$.

In the following, we want to detail how the typing rules guarantee the integrity of entities, which involves reasoning about the evolution of the system over time. To this end, we use $\mathcal{R}_t = (R_t, A_t, F_t, \rho_t, \leq_t)$ and $\mathcal{E}_t$ to indicate the record typing environment and the entity typing environment respectively, at a point in time $t \in \mathit{Timestamp}$. In order to reason about the data associated with an entity, we assume for each point in time $t \in \mathit{Timestamp}$ an *entity environment* $\epsilon_t : \mathit{Ent} \rightharpoonup_{\mathrm{fin}} \mathit{Record}$ that maps an entity to its associated data. Entity (typing) environments form the basis of the entity store, which we will describe in detail in Section 2.3.

Given $T \subseteq \mathit{Timestamp}$ and sequences $(\mathcal{R}_t)_{t \in T}$, $(\mathcal{E}_t)_{t \in T}$ and $(\epsilon_t)_{t \in T}$ of record typing environments, entity typing environments, and entity environments respectively, which represent the evolution of the system over time, we require

the following invariants to hold for all $t, t' \in \mathit{Timestamp}$, $r, r' \in \mathit{RecordName}$, $e \in \mathit{Ent}$, and $v \in \mathit{Record}$:

$$\text{if } \mathcal{E}_t(e) = r \text{ and } \mathcal{E}_{t'}(e) = r', \text{ then } r = r', \qquad \text{(stable type)}$$
$$\text{if } \mathcal{E}_t(e) \text{ is defined, then so is } \epsilon_t(e), \text{ and} \qquad \text{(well-definedness)}$$
$$\text{if } \epsilon_t(e) = v, \text{ then } \mathcal{E}_t(e) = r \text{ and } \mathcal{R}_{t'}, \mathcal{E}_{t'} \vdash v : r \text{ for some } t' \leq t. \qquad \text{(well-typing)}$$

We refer to the three invariants above collectively as the *entity integrity invariants*. The *stable type* invariant states that each entity can have at most one declared type throughout its lifetime. The *well-definedness* invariant guarantees that every entity that is given a type also has an associated record value. Finally, the *well-typing* invariant guarantees that the record value associated with an entity *was* well-typed at some earlier point in time t'.

The well-typing invariant is, of course, not strong enough. What we need is that the value v associated with an entity e *remains* well-typed throughout the lifetime of the system. This is, however, dependant on the record typing environment and the entity typing environment, which both may change over time. Therefore, we need to impose restrictions on the possible evolution of the record typing environment, and we need to take into account that entities used in the value v may have been deleted. We return to these issues in Section 2.2 and Section 2.3, and in the latter we will see that the entity integrity invariants are indeed satisfied by the system.

2.1.4 Ontology Language

Section 2.1.1 provides the semantic account of record types, and in order to specify record types, we use a variant of Attempto Controlled English [4] due to Jønsson Thomsen [8], referred to as the *ontology language*. The approach is to define data types in near-English text, in order to minimise the gap between requirements and specification. As an example, the record typing environment from Example 2.5 is specified in the ontology language as follows:

Person is abstract.
Person has a String called name.

Address has a String called road.
Address has an Int called no.

Customer is a Person.
Customer has an Address.

An ontology definition consists of a sequence of sentences as defined by the grammar below (where $[\cdot]$ denotes optionality):

$$
\begin{array}{lll}
\textit{Ontology} & ::= \textit{Sentence}^* & \text{(ontology)} \\
\textit{Sentence} & ::= \textit{RecordName}\ \textbf{is}\ [\textbf{a} \mid \textbf{an}]\ \textit{RecordName}. & \text{(supertype declaration)} \\
& \mid \textit{RecordName}\ \textbf{is abstract}. & \text{(abstract declaration)} \\
& \mid \textit{RecordName}\ \textbf{has}\ [\textbf{a} \mid \textbf{an}]\ \textit{Type} & \text{(field declaration)} \\
& \qquad [\textbf{called}\ \textit{FieldName}]. & \\
\textit{Type} & ::= \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} & \text{(type constants)} \\
& \mid \textbf{String} \mid \textbf{Timestamp} \mid \textbf{Duration} & \\
& \mid \textit{RecordName} & \text{(record type)} \\
& \mid \textbf{list of}\ \textit{Type} & \text{(list type)} \\
& \mid \textit{RecordName}\ \textbf{entity} & \text{(entity type)}
\end{array}
$$

The language of types *Type* reflects the definition of types in T and there is an obvious bijection $[\![\cdot]\!] : \textit{Type} \to T$ with $[\![\textbf{list of}\ t]\!] = [[\![t]\!]]$, $[\![r\ \textbf{entity}]\!] = \langle r \rangle$, and otherwise $[\![t]\!] = t$.

The semantics of the ontology language is given by a straightforward mapping into the domain of record typing environments. Each sentence is translated into a record typing environment. The semantics of a sequence of sentences is simply the closure of the union of each sentence's record typing environment:

$$
\begin{aligned}
[\![s_1 \cdots s_n]\!] &= \mathrm{Cl}\left([\![s_1]\!] \cup [\![s_2]\!] \cup \cdots \cup [\![s_n]\!]\right) \\
[\![r_1\ \textbf{is}\ [\textbf{a} \mid \textbf{an}]\ r_2.]\!] &= \left(\{r_1, r_2\}, \emptyset, \emptyset, \{r_1 \mapsto \emptyset, r_2 \mapsto \emptyset\}, \{(r_1, r_2)\}\right) \\
[\![r\ \textbf{is abstract}.]\!] &= \left(\{r\}, \{r\}, \emptyset, \{r \mapsto \emptyset\}, \emptyset\right) \\
[\![r\ \textbf{has}\ [\textbf{a} \mid \textbf{an}]\ t\ \textbf{called}\ f.]\!] &= \left(\{r\}, \emptyset, \{f\}, \{r \mapsto \{(f, [\![t]\!])\}\}, \emptyset\right)
\end{aligned}
$$

We omit the case where the optional *FieldName* is not supplied in a field declaration. We treat this form as syntactic sugar for r **has** (**a** | **an**) t **called** f. where f is derived from the type t. In this case a default name is used based on the type, simply by changing the first letter to a lower-case. Hence, in the example above the field name of a customer's address is address. Note that the record typing environment need not be well-formed (Definition 2.3), and a subsequent check for well-formedness has to be performed.

Data definitions added to the system via *addDataDefs* are specified in the ontology language. We require, of course, that the result of adding data definitions must yield a well-defined record typing environment. Moreover, we impose further monotonicity constraints which ensure that existing data in the system remain well-typed. We return to these constraints when we discuss the event log in Section 2.2. Type definitions retrieved via *getRecordDef* provide the semantic structure of a record type, that is its immediate supertypes, its fields, and an indication whether the record type is abstract. *getSubTypes* returns a list of immediate subtypes of a given record type, hence *getRecordDef* and *getSubTypes* provide the means for clients of the system to traverse the type hierarchy—both upwards and downwards.

2.1.5 Predefined Ontology

Unlike the original POETS architecture [6], our generalised architecture is not fixed to an enterprise resource planning (ERP) domain. However, we require a set of predefined record types, which are included in Appendix A. That is, the

record typing environment $\mathcal{R}_0$ denoted by the ontology in Appendix A is the initial record typing environment in all POETS instances.

The predefined ontology defines five root concepts in the data model, that is record types maximal with respect to the subtype relation $\leq$. Each of these five root concepts Data, Event, Transaction, Report, and Contract are abstract and only Event and Contract define record fields. Custom data definitions added via *addDataDefs* are only permitted as subtypes of Data, Transaction, Report, and Contract. In contrast to that, Event has a predefined and fixed hierarchy.

Data types represent elements in the domain of the system such as customers, items, and resources.

Transaction types represent events that are associated with a contract, such as payments, deliveries, and issuing of invoices.

Report types are result types of report functions, that is the data of reports, such as inventory status, income statement, and list of customers. The Report structure does not define *how* reports are computed, only *what kind* of result is computed. We will return to this discussion in Section 2.4.

Contract types represent the different kinds of contracts, such as sales, purchases, and manufacturing procedures. Similar to Report, the structure does not define what the contract dictates, only what is required to instantiate the contract. The purpose of Contract is hence dual to the purpose of Report: the former determines an input type, and the latter determines an output type. We will return to contracts in Section 2.5.

Event types form a fixed hierarchy and represent events that are logged in the system. Events are conceptually separated into *internal* events and *external* events, which we describe further in the following section.

2.2 Event Log

The event log is the only persistent state of the system, and it describes the complete state of a running POETS instance. The event log is an append-only list of records of the type Event defined in Appendix A. Each event reflects an atomic interaction with the running system. This approach is also applied at the "meta level" of POETS: in order to allow agile evolution of a running POETS instance, changes to the data model, reports, and contracts are reflected in the event log as well.

The monotonic nature of the event log—data is never overwritten or deleted from the system—means that the state of the system can be reconstructed at any previous point in time. In particular, transactions are never deleted, which is a legal requirement for ERP systems. The only component of the architecture that reads directly from the event log is the report engine (compare Figure 2), hence the only way to access data in the log is via a report.

All events are equipped with an internal timestamp (internalTimeStamp), the time at which the event is registered in the system. Therefore, the event log is always monotonically decreasing with respect to internal timestamps, as the

Event	Description
AddDataDefs	A set of data definitions is added to the system. The field defs contains the ontology language specification.
CreateEntity	An entity is created. The field data contains the data associated with the entity, the field recordType contains the string representation of the declared type, and the field ent contains the newly created entity value.
UpdateEntity	The data associated with an entity is updated.
DeleteEntity	An entity is deleted.
CreateReport	A report is created. The field code contains the specification of the report, and the fields description and tags are meta data.
UpdateReport	A report is updated.
DeleteReport	A report is deleted.
CreateContractDef	A contract template is created. The field code contains the specification of the contract template, and the fields recordType and description are meta data.
UpdateContractDef	A contract template is updated.
DeleteContractDef	A contract template is deleted.
CreateContract	A contract is instantiated. The field contractId contains the newly created identifier of the contract and the field contract contains the name of the contract template to instantiate, as well as data needed to instantiate the contract template.
UpdateContract	A contract is updated.
ConcludeContract	A contract is concluded.

Figure 5: Internal events.

newest event is at the head of the list. Conceptually, events are divided into *external* and *internal* events.

External events are events that are associated with a contract, and only the contract engine writes external events to the event log. The event type TransactionEvent models external events, and it consists of three parts: (i) a contract identifier (contractId), (ii) a timestamp (timeStamp), and (iii) a transaction (transaction). The identifier associates the external event with a contract, and the timestamp represents the time at which the external event takes place. Note that the timestamp need not coincide with the internal timestamp. For instance, a payment in a sales contract may be registered in the system the day after it takes place. There is hence no a priori guarantee that external events have decreasing timestamps in the event log—only external events that pertain to the same contract are required to have decreasing timestamps. The last component, transaction, represents the actual action that takes place, such as a payment from one person or company to another. The transaction is a record of type Transaction, for which the system has no presumptions.

Internal events reflect changes in the state of the system at a meta level. This is the case for example when a contract is instantiated or when a new record definition is added. Internal events are represented by the remaining subtypes of the Event record type. Figure 5 provides an overview of all non-abstract record types that represent internal events.

Entity Store		
Function	**Input**	**Output**
createEntity	record name, record	entity
updateEntity	entity, record	
deleteEntity	entity	

Figure 6: Entity store interface.

A common pattern for internal events is to have three event types to represent creation, update, and deletion of respective components. For instance, when a report is added to the report engine, a CreateReport event is persisted to the log, and when it is updated or deleted, UpdateReport and DeleteReport events are persisted accordingly. This means that previous versions of the report specification can be retrieved, and more generally that the system can be restarted simply by replaying the events that are persisted in the log on an initially empty system. Another benefit to the approach is that the report engine, for instance, does not need to provide built-in functionality to retrieve, say, the list of all reports added within the last month—such a list can instead be computed as a report itself! We will see how to write such a "meta" report in Section 2.4. Similarly, lists of entities, contract templates, and running contracts can be defined as reports.

Since we allow the data model of the system to evolve over time, we must be careful to ensure that the event log, and thus all data in it, remains well-typed at any point in time. Let $(\mathcal{R}_t)_{t\in T}$, $(\mathcal{E}_t)_{t\in T}$, and $(l_t)_{t\in T}$ be sequences of record typing environments, entity typing environments, and event logs respectively. Since an entity might be deleted over time, and thus is removed from the entity typing environment, the event log may not be well-typed with respect to the current entity typing environment. To this end, we type the event log with respect to the *accumulated entity typing environment* $\widehat{\mathcal{E}}_t = \bigcup_{t'\leq t} \mathcal{E}_{t'}$. That is, $\widehat{\mathcal{E}}_t(e) = r$ iff there is some $t' \leq t$ with $\mathcal{E}_{t'}(e) = r$. The stable type invariant guarantees that $\widehat{\mathcal{E}}_t$ is indeed well-defined.

For changes to the record typing environment, we require the following invariants for any points in time t, t' and the event log l_t at time t:

$$\text{if } t' \geq t \text{ then } \mathcal{R}_{t'} = \mathcal{R}_t \cup \mathcal{R}_\Delta \text{ for some } \mathcal{R}_\Delta, \text{ and} \qquad \text{(monotonicity)}$$

$$\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash l_t : [\mathsf{Event}]\,. \qquad \text{(log typing)}$$

Note that the *log typing* invariant follows from the *monotonicity* invariant and the type checking $\mathcal{R}_t, \mathcal{E}_t \vdash e : \mathsf{Event}$ for each new incoming event, provided that for each record name r occurring in the event log, no additional record fields are added to r, and r is not made an abstract record type. We will refer to the two invariants above collectively as *record typing invariants*. They will become crucial in the following section.

2.3 Entity Store

The entity store provides very simple functionality, namely creation, deletion and updating of entities, respectively. To this end, the entity store maintains the

current entity typing environment $\mathcal{E}_t$ as well as the history of entity environments $\epsilon_0, \ldots, \epsilon_t$. The interface of the entity store is summarised in Figure 6.

The creation of a new entity via *createEntity* at time $t+1$ requires a declared type r and an initial record value v, and it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : r$. If the value type checks, a *fresh* entity value $e \notin \bigcup_{t' \leq t} \mathrm{dom}(\epsilon_{t'})$ is created, and the entity environment and the entity typing environment are updated accordingly:

$$\epsilon_{t+1}(x) = \begin{cases} v & \text{if } x = e, \\ \epsilon_t(x) & \text{otherwise,} \end{cases} \qquad \mathcal{E}_{t+1}(x) = \begin{cases} r & \text{if } x = e, \\ \mathcal{E}_t(x) & \text{otherwise.} \end{cases}$$

Moreover, a CreateEntity event is persisted to the event log containing e, r, and v for the relevant fields.

Similarly, if the data associated with an entity e is updated to the value v at time $t+1$, then it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : \mathcal{E}_t(e)$, and the entity store is updated like above. Note that the entity typing environment is unchanged, that is $\mathcal{E}_{t+1} = \mathcal{E}_t$. A corresponding UpdateEntity event is persisted to the event log containing e and v for the relevant fields.

Finally, if an entity e is deleted at time $t+1$, then it is removed from both the entity store and the entity typing environment:

$$\epsilon_{t+1}(x) = \epsilon_t(x) \text{ iff } x \in \mathrm{dom}(\epsilon_t) \setminus \{e\}$$
$$\mathcal{E}_{t+1}(x) = \mathcal{E}_t(x) \text{ iff } x \in \mathrm{dom}(\mathcal{E}_t) \setminus \{e\}\,.$$

A corresponding DeleteEntity event is persisted to the event log containing e for the relevant field.

Note that, by default, $\epsilon_{t+1} = \epsilon_t$ and $\mathcal{E}_{t+1} = \mathcal{E}_t$, unless one of the situations above apply. It is straightforward to show that the *entity integrity invariants* are maintained by the operations described above (the proof follows by induction on the timestamp t). Internally, that is, for the report engine compare Figure 2, the entity store provides a lookup function $\mathrm{lookup}_t : \mathit{Ent} \times [0,t] \rightharpoonup_{\mathrm{fin}} \mathit{Record}$, where $\mathrm{lookup}_t(e, t')$ provides the latest value associated with the entity e at time t', where t is the current time. Note that this includes the case in which e has been deleted at or before time t'. In that case, the value associated with e just before the deletion is returned. Formally, lookup_t is defined in terms of the entity environments as follows:

$$\mathrm{lookup}_t(e, t_1) = v \text{ iff } \exists t_2 \leq t_1 \colon \epsilon_{t_2}(e) = v \text{ and } \forall t_2 < t_3 \leq t_1 \colon e \notin \mathrm{dom}(\epsilon_{t_3}).$$

In particular, we have that if $e \in \mathrm{dom}(\epsilon_{t_1})$ then $\mathrm{lookup}_t(e, t_1) = \epsilon_{t_1}(e)$.

From this definition and the invariants of the system, we obtain the following property:

Corollary 2.8. *Let $(\mathcal{R}_t)_{t \in T}$, $(\mathcal{E}_t)_{t \in T}$, and $(\epsilon_t)_{t \in T}$ be sequences of record typing environments, entity typing environments, and entity environments respectively, satisfying the entity integrity invariants and the record typing invariants. Then the following holds for all timestamps $t \leq t_1 \leq t_2$ and entities $e \in \mathit{Ent}$:*

If $\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash e : \langle r \rangle$ then $\mathrm{lookup}_{t_2}(e, t_1) = v$ for some v and $\mathcal{R}_{t_2}, \widehat{\mathcal{E}}_{t_2} \vdash v : r$.

Report Engine		
Function	**Input**	**Output**
addReport	name, type, description, tags, report definition	
updateReport	name, type, description, tags, report definition	
deleteReport	name	
queryReport	name, list of values	value

Figure 7: Report engine interface.

Proof. Assume that $\mathcal{R}_t, \widehat{\mathcal{E}}_t \vdash e : \langle r \rangle$. Then it follows from the typing rule for entity values and the subtyping rules that $\widehat{\mathcal{E}}_t(e) = r'$ for some r' with $r' \leq_t r$. That is, there is some $t' \leq t$ with $\mathcal{E}_{t'}(e) = r'$. Hence, from the well-definedness invariant it follows that $\epsilon_{t'}(e)$ is defined. Since $t' \leq t \leq t_1$, we can thus conclude that $\text{lookup}_{t_2}(e, t_1) = (r'', m)$, for some record value (r'', m).

According to the definition of lookup_{t_2}, we then have some $t_3 \leq t_1$ with $\epsilon_{t_3}(e) = (r'', m)$. Applying the well-typing invariant, we obtain some $t_4 \leq t_3$ with $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : \mathcal{E}_{t_3}(e)$. Since, by the stable type invariant, $\mathcal{E}_{t_3}(e) = \mathcal{E}_{t'} = r'$, we then have $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : r'$. Moreover, according to the typing rules, this can only be the case if $r'' \leq_{t_4} r'$.

Due to the monotonicity invariant, we know that $\mathcal{R}_{t_2} = \mathcal{R}_{t_4} \cup \mathcal{R}_\Delta$ for some $\mathcal{R}_\Delta$. In particular, this means that $r'' \leq_{t_4} r'$ implies that $r'' \leq_{t_2} r'$. Similarly, $r' \leq_t r$ implies that $r' \leq_{t_2} r$. Hence, by transitivity of $\leq_{t_2}$, we have that $r'' \leq_{t_2} r$.

According to the implementation of the entity store, we know that $\epsilon_{t_3}(e) = (r'', m)$ implies that (r'', m) occurs in the event log (as part of an event of type CreateEntity or UpdateEntity) at least from t_3 onwards, in particular in the event log l_{t_2} at t_2. Since, by the log typing invariant, the event log l_{t_2} is well-typed as $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash l_{t_2} : [\mathsf{Event}]$, we know that $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r''$. From the subtype relation $r'' \leq_{t_2} r$ we can thus conclude $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r$. □

The corollary above describes the fundamental safety property with respect to entity values: if an entity value previously entered the system, and hence type checked, then all future dereferencing will not get stuck, and the obtained value will be well-typed with respect to the accumulated entity typing environment.

2.4 Report Engine

The purpose of the report engine is to provide a structured view of the database that is constituted by the system's event log. This structured view of the data in the event log comes in the form of a *report*, which provides a collection of condensed structured information compiled from the event log. Conceptually, the data provided by a report is compiled from the event log by a function of type $[\mathsf{Event}] \to \mathsf{Report}$, a *report function*. The *report language* provides a means to specify such a report function in a declarative manner. The interface of the report engine is summarised in Figure 7.

2.4.1 The Report Language

In this section, we provide an overview over the report language. For a detailed description of the language including the full static and dynamic semantics consult Appendix B.

The report language is—much like the query fragment of *SQL*—a functional language *without side effects*. It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log, the report language must provide operations to relate, filter, join, and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous—containing data of different kinds—the report language offers a comprehensive type system that allows us to safely operate in this setting.

Example 2.9. Consider the following simple report function that lists all reports available in the system:

```
reports : [PutReport]
reports = nubProj (λx → x.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events,
                                  ur.name ≡ cr.name]]
```

The report function above uses the two functions *nubProj* and *first*, which are defined in the standard library of the report language. The function *nubProj* of type $(\mathsf{Eq}\ b) \Rightarrow (a \to b) \to [a] \to [a]$ removes duplicates in the given list according to the equality on the result of the provided projection function. In the example above, reports with the same name are considered duplicates. The function $\mathit{first} : a \to [a] \to a$ returns the first element of the given list or the default value provided as first argument if the list is empty.

Every report function implicitly has as its first argument the event log of type [Event]—a list of events—bound to the name **events**. The syntax—and to large parts also the semantics—is based on Haskell [9]. The central data structure is that of lists. In order to formulate operations on lists concisely, we use list comprehensions [16] as seen in Example 2.9. A list comprehension of the form $[\ e \mid c\ ]$ denotes a list containing elements of the form e generated by c, where c is a sequence of *generators* and *filters*.

As we have mentioned, access to type information and its propagation to subsequent computations is essential due to the fact that the event log is a list of heterogeneously typed elements—events of different kinds. The generator $cr : \mathsf{CreateReport} \leftarrow \mathbf{events}$ iterates through elements of the list **events**, binding each element to the variable *cr*. The typing $cr : \mathsf{CreateReport}$ restricts this iteration to elements of type CreateReport, a subtype of Event. This type information is propagated through the subsequent generators and filters of the list comprehension. In the filter $\mathit{ur.name} \equiv \mathit{cr.name}$, we use the fact that elements of type ReportEvent have a field name of type **String**. When binding the first element of the result of the nested list comprehension to the variable *pr* it is also checked whether this element is in fact of type PutReport. Thus we ignore reports that are marked as deleted via a DeleteReport event.

The report language is based on the simply typed lambda calculus extended with polymorphic (non-recursive) let expressions as well as type case expressions. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\},$$

where x ranges over variables, and c over constants which include integers, Booleans, tuples and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we assume a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. List comprehensions are mere syntactic sugar and can be reduced to **fold** and let expressions as for example in Haskell [9].

The extended list comprehensions of the report language that allow filtering according to run-time type information depend on type case expressions of the form $\textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}$. In such a type case expression, an expression e of some record type r_e gets evaluated to record value v which is then bound to a variable x. The record type r that the record value v is matched against can be any subtype of r_e. Further evaluation of the type case expression depends on the type r_v of the record value v. This type can be any subtype of r_e. If $r_v \leq r$ then the evaluation proceeds with e_1, otherwise with e_2. Binding e to a variable x allows us to use the stricter type r in the expression e_1.

Another important component of the report language consists of the dereferencing operators ! and @, which give access to the lookup operator provided by the entity store. Given an expression e of an entity type $\langle r \rangle$, both dereferencing operators provide a value v of type r. That is, both ! and @ are unary operators of type $\langle r \rangle \to r$ for any record type r. In the case of the operator !, the resulting record value v is the latest value associated with the entity to which e evaluates. More concretely, given an entity value v, the expression $v!$ evaluates to the record value $\text{lookup}_t(v, t)$, where t is the current timestamp.

On the other hand, the *contextual* dereference operator @ provides as the result the value associated with the entity at the moment the entity was used in the event log (based on the internalTimeStamp field). This is the case when the entity is extracted from some event from the event log. Otherwise, the entity value stems from an actual argument to the report function. In the latter case @ behaves like the ordinary dereference operator !. In concrete terms, every entity value v that enters the event log is annotated with the timestamp of the event it occurs in. That is, each entity value embedded in an event e in the event log, occurs in an annotated form (v, s), where s is the value of e's internalTimeStamp field. Given such an annotated entity value (v, s), the expression $(v,s)@$ evaluates to $\text{lookup}_t(v, s)$ and given a bare entity value v the expression $v@$ evaluates to $\text{lookup}_t(v, t)$.

Note that in each case for either of the two dereference operators, Corollary 2.8 guarantees that the lookup operation yields a record value of the right type. That is, both $! : \langle r \rangle \to r$ and $@ : \langle r \rangle \to r$ are total functions that never get stuck.

Example 2.10. The entity store and the contextual dereferencing operator provide a solution to a recurring problem in ERP systems, namely how to maintain historical data for auditing. For example, when an invoice is issued in a sale, then a copy of the customer information *at the time* of the invoice is needed for

auditing. Traditional ERP systems solve the problem by explicit copying of data, since referenced data might otherwise get destructively updated.

Since data is never deleted in a POETS system, we can solve the problem without copying. Consider the following definition of transactions that represent issuing of invoices, and invoices respectively (we assume that the record types Customer and OrderLine are already defined):

IssueInvoice is a Transaction.	*Invoice is Data.*
IssueInvoice has a Customer entity.	*Invoice has a Customer.*
IssueInvoice has a list of OrderLine.	*Invoice has a list of OrderLine.*

Rather than containing a Customer record, an IssueInvoice transaction contains a Customer entity, which eliminates copying of data. From an IssueInvoice transaction we can instead *derive* the invoice data by the following report function:

$$
\begin{aligned}
&\mathit{invoices} : [\mathsf{Invoice}] \\
&\mathit{invoices} = [\mathsf{Invoice}\{\mathit{customer} = \mathit{ii.customer@},\ \mathit{orderLines} = \mathit{ii.orderLines}\} \mid \\
&\quad \mathit{tr} : \mathsf{TransactionEvent} \leftarrow \mathbf{events}, \\
&\quad \mathit{ii} : \mathsf{IssueInvoice} = \mathit{tr.transaction}]
\end{aligned}
$$

Note how the @ operator is used to dereference the customer data: since the *ii.customer* value originates from an event in the event log, the contextual dereferencing will produce data associated with the customer at the time when the invoice was issued, as required.

2.4.2 Incrementalisation

While the type system is important in order to avoid obvious specification errors, it is also important to ensure a fast execution of the thus obtained functional specifications. This is, of course, a general issue for querying systems. In our system it is, however, of even greater importance since shifting the structure of the data—from the data store to the domain of queries—means that queries operate on the complete data set of the database. In principle, the data of each report has to be recomputed after each transaction by applying the corresponding report function to the updated event log. In other words, if treated naïvely, the conceptual simplification provided by the flat event log has to be paid via more expensive computations.

This issue can be addressed by transforming a given report function f into an incremental function f' that updates the report data computed previously according to the changes that have occurred since the report data was computed before. That is, given an event log l and an update to it $l \oplus e$, we require that $f(l \oplus e) = f'(f(l), e)$. The new report data $f(l \oplus e)$ is obtained by updating the previous report data $f(l)$ according to the changes e. In the case of the event log, we have a list structure. Changes only occur *monotonically*, by adding new elements to it: given an event log l and a new event e, the new event log is $e \# l$, where $\#$ is the list constructor of type $\alpha \to [\alpha] \to [\alpha]$.

Here it is crucial that we have restricted the report language such that operations on lists are limited to the higher-order function **fold**. The fundamental idea of incrementalising report functions is based on the following equation satisfied

by **fold**:

$$\textbf{fold}\ f\ e\ (x\ \#\ xs) = f\ x\ (\textbf{fold}\ f\ e\ (xs))$$

Based on this idea, we are able to make the computation of most report functions independent of the size of the event log but only dependent of the changes to the event log and the previous result of the report function [12]. Unfortunately, if we consider for example list comprehensions containing more than one generator, we have functions with nested folds. In order to properly incrementalise such functions, we need to move from list structures to multisets. This is, however, only rarely a practical restriction since most aggregation functions are based on commutative binary operations and are thus oblivious to ordering.

2.4.3 Lifecycle of Reports

Like entities, the set of reports registered in a running POETS instance—and thus available for querying—can be changed via the external interface to the report engine. To this end, the report engine interface provides the operations *addReport*, *updateReport*, and *deleteReport*. The former two take a *report specification* that contains the name of the report, the definition of the report function that generates the report data and the type of the report function. Optionally, it may also contain further meta information in the form of a description text and a list of tags.

Example 2.11. Reconsider the function defined in Example 2.9 that lists all active reports with all their meta data. The following report specification uses the report function from Example 2.9 in order to define a report function that lists the names of all active report:

```
name: ReportNames
description: A list of names of all registered reports.
tags: internal, report

reports : [PutReport]
reports = nubProj (λx → x.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events,
                                   ur.name ≡ cr.name]]

report : [String]
report = [r.name | r ← reports]
```

In the header of the report specification, the name and optionally also a description text as well as a list of tags is provided as meta data to the actual report function specification. Every report specification must define a top-level function called **report**, which provides the report function that derives the report data from the event log. In the example above, this function takes no (additional) arguments and returns a list of strings—the names of active reports.

Calls to *addReport* and *updateReport* are both reflected by a corresponding event of type CreateReport and UpdateReport respectively. Both events are

Contract Engine		
Function	**Input**	**Output**
createTemplate	name, type, description, specification	
updateTemplate	name, type, description, specification	
deleteTemplate	name	
createContract	meta data	contract ID
updateContract	contract ID, meta data	
concludeContract	contract ID	
getContract	contract ID	contract state
registerTransaction	contract ID, timestamp, transaction	

Figure 8: Contract engine interface.

subtypes of PutReport and contain the meta information as well as the original specification text of the concerning report. When a report is no longer needed, it can be removed from the report engine by a corresponding *deleteReport* operation. Note that the change and removal of reports only affect the state of the POETS system from the given point in time. Transactions that occurred prior to a change or deletion of a report are not affected. This is important for the system's ability to fully recover after a crash by replaying the events from the event log.

The last operation provided by the report engine—*queryReport*—constitutes the core functionality of the reporting system. Given a name of a registered report and a list of arguments, this operation supplies the given arguments to the corresponding report function and returns the result. For example, the *ReportNames* report specified in Example 2.11 does not require any arguments—its type is [**String**]—and returns the names of registered reports.

2.5 Contract Engine

The role of the contract engine is to determine which transactions—that is external events, compare Section 2.2—are expected by the system. Transactions model events that take place according to an *agreement*, for instance a delivery of goods in a sale, a payment in a lease agreement, or a movement of items from one inventory to another in a production plan. Such agreements are referred to as *contracts*, although they need not be legally binding contracts. The purpose of a contract is to provide a detailed description of *what* is expected, by *whom*, and *when*. A sales contract, for example, may stipulate that first the company sends an invoice, then the customer pays within a certain deadline, and finally the company delivers goods within another deadline.

The interface of the contract engine is summarised in Figure 8.

2.5.1 Contract Templates

In order to specify contracts such as the aforementioned sales contract, we use an extended variant of the contract specification language (CSL) of Hvitved et al. [7], which we will refer to as the POETS contract specification language (PCSL) in

the following. For reusability, contracts are always specified as *contract templates* rather than as concrete contracts. A contract template consists of four parts: (i) a template name, (ii) a template type, which is a subtype of the Contract record type, (iii) a textual description, and (iv) a PCSL specification. We describe PCSL in Section 2.5.3.

The template name is a unique identifier, and the template type determines the parameters that are available in the contract template.

Example 2.12. We may define the following type for sales contracts in the ontology language (assuming that the record types Customer, Company, and Goods have been defined):

Sale is a Contract.
Sale has a Customer entity.
Sale has a Company entity.
Sale has a list of Goods.
Sale has an Int called amount.

With this definition, contract templates of type Sale are parametrised over the fields customer, company, goods, and amount of types ⟨Customer⟩, ⟨Company⟩, [Goods], and **Int**, respectively.

The contract engine provides an interface to add contract templates (*createTemplate*), update contract templates (*updateTemplate*), and remove contract templates (*deleteTemplate*) from the system at run-time. The structure of contract templates is reflected in the external event types CreateContractDef, UpdateContractDef, and DeleteContractDef, compare Section 2.2. A list of (non-deleted) contract templates can hence be computed by a report, similar to the list of (non-deleted) reports from Example 2.11.

2.5.2 Contract Instances

A contract template is instantiated via *createContract* by supplying a record value v of a subtype of Contract. Besides custom fields, which depend on the type at hand, such a record always contains the fields templateName and startDate inherited from the Contract record type, compare Appendix A. The field templateName contains the name of the template to instantiate, and the field startDate determines the start date of the contract. The fields of v are substituted into the contract template in order to obtain a *contract instance*, and the type of v must therefore match the template type. For instance, if v has type Sale then the field templateName must contain the name of a contract template that has type Sale. We refer to the record v as *contract meta data*.

When a contract c is instantiated by supplying contract meta data v, a *fresh* contract identifier i is created, and a CreateContract event is persisted in the event log with with contract $= v$ and contractId $= i$. Hereafter, transactions t can be registered with the contract via *registerTransaction*, which will update the contract to a *residual contract* c', written $c \xrightarrow{t} c'$, and a TransactionEvent with transaction $= t$ and contractId $= i$ is written to the event log. The state of the contract can be acquired from the contract engine at any given point in time via

getContract, which enables run-time analyses of contracts, for instance in order to generate a list of expected transactions.

Registration of a transaction $c \xrightarrow{t} c'$ is only permitted if the transaction is expected in the current state c. That is, there need not be a residual state for all transactions. After zero or more successful transactions, $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, the contract may be concluded via *concludeContract*, provided that the residual contract c_n does not contain any outstanding obligations. This results in a ConcludeContract event to be persisted in the event log.

The lifecycle described above does not take into account that contracts may have to be updated at run-time, for example if it is agreed to extend the payment deadline in a sales contract. To this end, running contracts are allowed to be updated, simply by supplying new contract meta data (*updateContract*). The difference in the new meta data compared to the old meta data may not only be a change of, say, items to be sold, but it may also be a change in the field templateName. The latter makes it is possible to replace the old contract by a qualitatively different contract, since the new contract template may describe a different workflow. There is, however, an important restriction: a contract can only be updated if any previous transactions registered with the contract also conform with the new contract. That is, if the contract has evolved like $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, and an update to a new contract c' is requested, then only if $c' \xrightarrow{t_1} c'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c'_n$, for some $c'_1, \ldots, c'_n$, is the update permitted. A successful update results in an UpdateContract event to be written to the event log with the new meta data.

Note that, for simplicity, we only allow the updates described above. Another possibility is to allow updates where the current state of the contract c is replaced directly by a new state c'. Although we can achieve this effect via a suitably defined contract template and the *updateContract* function above, a direct update is preferable.

Similarly to contract templates, a list of (non-concluded) contract instances can be computed by a report that inspects CreateContract, UpdateContract, and ConcludeContract events respectively.

2.5.3 The Contract Language

The fourth component of contract templates—the PCSL specification—is the actual normative content of contract templates. The core grammar for PCSL is presented in Figure 9. PCSL extends CSL mainly at the level of expressions E, by adding support for the value types in POETS, as well as lambda abstractions and function applications. At the level of clauses C, PCSL is similar to CSL, albeit with a slightly altered syntax.

The semantics of PCSL is a straightforward extension of that of CSL [7], although we use a partial small-step semantics rather than CSL's total small-step semantics. That is, there need not be a residue for all clauses and transactions, as described in Section 2.5.2. This is simply in order to prevent "unexpected" events from entering the system, for instance we only allow a payment to be entered into the system if a running contract expects that payment.

The type system for clauses is identical with CSL. Typing of expressions is,

$$
\begin{array}{lll}
\mathit{Tmp} & ::= & \textbf{name} : \mathit{ContractName} \qquad \text{(contract template)} \\
& & \quad \textbf{type} : \mathit{RecordName} \\
& & \quad \textbf{description} : \mathit{String} \\
& & \quad \mathit{Def} \ldots \mathit{Def}\ \textbf{contract} = C \\
\mathit{Def} & ::= & \textbf{val}\ \mathit{Var} = E \qquad \text{(value definition)} \\
& | & \textbf{clause}\ \mathit{ClauseName}(\mathit{Var} : T, \ldots, \mathit{Var} : T) \qquad \text{(clause template)} \\
& & \qquad\qquad \langle \mathit{Var} : T, \ldots, \mathit{Var} : T \rangle = C \\
C & ::= & \textbf{fulfilment} \qquad \text{(no obligations)} \\
& | & \langle E \rangle\ \mathit{RecordName}(F, \ldots, F) \qquad \text{(obligation)} \\
& & \quad \textbf{where}\ E\ \textbf{due}\ D\ \textbf{remaining}\ \mathit{Var}\ \textbf{then}\ C \\
& | & \textbf{when}\ \mathit{RecordName}(F, \ldots, F) \qquad \text{(external choice)} \\
& & \quad \textbf{where}\ E\ \textbf{due}\ D\ \textbf{remaining}\ \mathit{Var}\ \textbf{then}\ C\ \textbf{else}\ C \\
& | & \textbf{if}\ E\ \textbf{then}\ C\ \textbf{else}\ C \qquad \text{(internal choice)} \\
& | & C\ \textbf{and}\ C \qquad \text{(conjunction)} \\
& | & C\ \textbf{or}\ C \qquad \text{(disjunction)} \\
& | & \mathit{ClauseName}(E, \ldots, E)\langle E, \ldots, E \rangle \qquad \text{(instantiation)} \\
F & ::= & \mathit{FieldName}\ \mathit{Var} \qquad \text{(field binder)} \\
R & ::= & \mathit{RecordName}\ \mathit{Var} \qquad \text{(record binder)} \\
T & ::= & \mathit{TypeVar} \qquad \text{(type variable)} \\
& | & () \qquad \text{(unit type)} \\
& | & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \textbf{String} \qquad \text{(type constants)} \\
& | & \textbf{Timestamp} \mid \textbf{Duration} \\
& | & \mathit{RecordName} \qquad \text{(record type)} \\
& | & [T] \qquad \text{(list type)} \\
& | & \langle T \rangle \qquad \text{(entity type)} \\
& | & T \to T \qquad \text{(function type)} \\
E & ::= & \mathit{Var} \qquad \text{(variable)} \\
& | & \mathit{BaseValue} \qquad \text{(base value)} \\
& | & \mathit{RecordName}\{\mathit{FieldName} = E, \ldots, \mathit{FieldName} = E\} \qquad \text{(record expression)} \\
& | & [E, \ldots, E] \qquad \text{(list expression)} \\
& | & \lambda \mathit{Var} \to E \qquad \text{(function abstraction)} \\
& | & E\ E \qquad \text{(function application)} \\
& | & E \oplus E \qquad \text{(binary expression)} \\
& | & E.\mathit{FieldName} \qquad \text{(field projection)} \\
& | & E\{\mathit{FieldName} = E\} \qquad \text{(field update)} \\
& | & \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E \qquad \text{(conditional)} \\
& | & \textbf{case}\ E\ \textbf{of}\ R \to E | \cdots | R \to E \qquad \text{(record type casing)} \\
D & ::= & \textbf{after}\ E\ \textbf{within}\ E \qquad \text{(deadline expression)} \\
\oplus & ::= & \times \mid / \mid + \mid \langle\times\rangle \mid \langle+\rangle \mid \# \mid \equiv \mid \leq \mid \wedge \qquad \text{(binary operators)}
\end{array}
$$

Figure 9: Grammar for the core contract language PCSL. *ContractName* is the set of all contract template names, *ClauseName* is the set of all clause template names ranged over by k, *Var* is the set of all variable names ranged over by x, *TypeVar* is the set of all type variable names ranged over by α, and $\mathit{BaseValue} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Ent}$.

```
name: salesContract
type: Sale
description: "A simple sales contract between a company and a customer"

fun elem x = foldr (λy b → x ≡ y ∨ b) false
fun filter f = foldr (λx b → if f x then x # b else b) []
fun subset l1 l2 = all (λx → elem x l2) l1
fun diff l1 l2 = filter (λx → ¬ (elem x l2)) l1

clause sale(goods : [Goods], amount : Int)⟨comp : ⟨Company⟩, cust : ⟨Customer⟩⟩ =
 ⟨comp⟩ IssueInvoice(goods g, amount a)
   where g ≡ goods ∧ a ≡ amount due immediately
 then
 ⟨cust⟩ Payment(amount a)
   where a ≡ amount due within 14D
 and
 delivery(goods, 1 W)⟨comp⟩

clause delivery(goods : [Goods], deadline : Duration)⟨comp : ⟨company⟩⟩ =
 if goods ≡ [] then
   fulfilment
 else
   ⟨comp⟩ Delivery(goods g)
     where g ≢ [] ∧ subset g goods due within deadline remaining r
   then
   delivery(diff goods g, r)⟨comp⟩

contract = sale(goods, amount)⟨company, customer⟩
```

Figure 10: PCSL sales contract template of type Sale.

however, more challenging since we have introduced (record) polymorphism as well as subtyping. We will not present the extended semantics nor the extended typing rules, but only remark that the typing serves the same purpose as in CSL: evaluation of expressions does not get stuck and always terminates, and contracts have unique blame assignment.

Example 2.13. We demonstrate PCSL by means of an example, presented in Figure 10. The contract template is of the type Sale from Example 2.12, which means that the fields goods, amount, company, and customer are available in the body of the contract template, that is the right-hand side of the **contract** keyword. Hence, concrete values are substituted from the contract meta data when the template is instantiated, as described in Section 2.5.2.

The example uses standard syntactic sugar at the level of expressions, for instance $\neg e$ means **if** e **then** *false* **else** *true* and $e_1 \vee e_2$ means $\neg(\neg e_1 \wedge \neg e_2)$. Moreover, we omit the **after** part of a deadline if it is 0, we write **immediately** for **within** 0, we omit the **remaining** part if it is not used, and we write **fun** $f\, x_1 \cdots x_n = e$ for **val** $f = \lambda x_1 \to \cdots \lambda x_n \to e$.

The template implements a simple workflow: first the company issues an invoice, then the customer pays within 14 days, and simultaneously the company

delivers goods within a week. Delivery of goods is allowed to take place in multiple deliveries, which is coded as the recursive clause template *delivery*. Note how the variable r is bound to the remainder of the deadline: All deadlines in a **then** branch are relative to the time of the guarding event, hence the relative deadline for delivering the remaining goods is whatever remains of the original one week deadline. Note also that the initial reference time of a contract instance is determined by the field startDate in the contract meta data, compare Appendix A. Hence if the contract above is instantiated with start date $t \in \mathit{Timestamp}$, then the invoice is supposed to be issued at time t.

Finally, we remark that obligation clauses are binders. That is, for instance the variable g is bound to value of the field goods of the IssueInvoice transaction when it takes place, and the scope of g is the **where** clause and the continuation clause following the **then** keyword.

Built-in symbols PCSL has a small set of built-in symbols, from which other standard functions can be derived:

$\mathit{foldl} : (a \to b \to a) \to a \to [b] \to a$
$\mathit{foldr} : (a \to b \to b) \to b \to [a] \to b$
$\mathit{ceil} : \mathsf{Real} \to \mathbf{Int}$
$\mathit{reports} : \mathsf{Reports}$

The list includes fold operations in order to iterate over lists, since explicit recursion is not permitted, and a special constant *reports* of type Reports. The record type Reports is internally derived from the active reports in the report engine, and it is used only in the contract engine in order to enable querying of reports from within contracts. The record type contains one field per report. For instance, if the report engine contains a single report *Inventory* of type Inventory, then the typing of Report is (using the same notation as in Section 2.1.1):

$$\rho(\mathsf{Reports}) = \{(\mathsf{inventory}, () \to \mathsf{Inventory})\},$$

and the expression *reports.inventory* () invokes the report.

3 Use Case: μERP

In this section we describe a use case instantiation of POETS, which we refer to as μERP. With μERP we model a simple ERP system for a small bicycle shop. Naturally, we do not intend to model all features of a full-blown ERP system, but rather we demonstrate a limited set of core ERP features. In our use case, the shop purchases bicycles from a bicycle vendor, and sells those bicycles to customers. We want to make sure that the bicycle shop only sells bicycles in stock, and we want to model a repair guarantee, which entitles customers to have their bikes repaired free of charge up until three months after purchase.

Following Henglein et al. [6], we also provide core financial reports, namely the income statement, the balance sheet, the cash flow statement, the list of open (not yet paid) invoices, and the value-added tax (VAT) report. These reports are typical, minimal legal requirements for running a business. We provide some example code in this section, and the complete specification is included in

Appendix C. As we have seen in Section 2, instantiating POETS amounts to defining a data model, a set of reports, and a set of contract templates. We describe each of these components in the following subsections.

3.1 Data Model

The data model of μERP is tailored to the ERP domain in accordance with the REA ontology [10]. Therefore, the main components of the data model are resources, transactions (that is, events associated with contracts), and agents. The complete data model is provided in Appendix C.1.

Agents are modelled as an abstract type Agent. An agent is either a Customer, a Vendor, or a special Me agent. Customers and Vendors are equipped with a name and an address. The Me type is used to represent the bicycle company itself. In a more elaborate example, the Me type will have subtypes such as Inventory or SalesPerson to represent subdivisions of, or individuals in, the company. The agent model is summarised below:

Agent is Data.

Customer is an Agent.
Customer has a String called name.
Customer has an Address.

Vendor is an Agent.
Vendor has a String called name.
Vendor has an Address.

Me is an Agent.

Resources are—like agents—Data. In our modelling of resources, we make a distinction between *resource types* and *resources*. A resource type represents a kind of resource, and resource types are divided into currencies (Currency) and item types (ItemType). Since we are modelling a bicycle shop, the only item type (for now) is bicycles (Bicycle). A resource is an instance of a resource type, and—similar to resource types—resources are divided into money (Money) and items (Item). Our modelling of items assumes an implicit unit of measure, that is we do not explicitly model units of measure such as pieces, boxes, pallets, etc. Our resource model is summarised below:

ResourceType is Data.
ResourceType is abstract.

Currency is a ResourceType.
Currency is abstract.

DKK is a Currency.
EUR is a Currency.

ItemType is a ResourceType.
ItemType is abstract.

Bicycle is an ItemType.

Bicycle has a String called model.

Resource is Data.
Resource is abstract.

Money is a Resource.
Money has a Currency.
Money has a Real called amount.

Item is a Resource.
Item has an ItemType.
Item has a Real called quantity.

Transactions (events in the REA terminology) are, not surprisingly, subtypes of the built-in Transaction type. The only transactions we consider in our use case

are bilateral transactions (BiTransaction), that is transactions that have a sender and a receiver. Both the sender and the receiver are agent entities, that is a bilateral transaction contains references to two agents rather than copies of agent data. For our use case we model payments (Payment), deliveries (Delivery), issuing of invoices (IssueInvoice), requests for repair of a set of items (RequestRepair), and repair of a set of items (Repair). Issuing of invoices contain the relevant information for modelling of VAT, encapsulated in the OrderLine type. We include some of these definitions below:

BiTransaction is a Transaction.
BiTransaction is abstract.
BiTransaction has an Agent entity called sender.
BiTransaction has an Agent entity called receiver.

Transfer is a BiTransaction.
Transfer is abstract.

Payment is a Transfer.

Payment is abstract.
Payment has Money.

CashPayment is a Payment.
CreditCardPayment is a Payment.
BankTransfer is a Payment.

IssueInvoice is a BiTransaction.
IssueInvoice has a list of OrderLine called orderLines.

Besides agents, resources, and transactions, the data model defines the output types of reports (Appendix C.1.3) the input types of contracts (Appendix C.1.4), and generic data definitions such as Address and OrderLine. The report types define the five mandatory reports mentioned earlier, and additional Inventory and TopNCustomers report types. The contract types define the two types of contracts for the bicycle company, namely Purchase and Sale.

3.2 Reports

The specification of reports is divided into four parts: prelude functions (Appendix C.2.1), domain-specific prelude functions (Appendix C.2.2), internal reports (Appendix C.2.3), and external reports (Appendix C.2.4).

Prelude functions are utility functions that are independent of the custom data model. These functions are automatically added to all POETS instances, but they are included in the appendix for completeness. The prelude includes standard functions such as *filter*, but it also includes generators for accessing event log data such as *reports*. The event log generators provide access to data that has a lifecycle such as contracts or reports, compare Section 2.2.

Domain-specific prelude functions are utility functions that depend on the custom data model. The *itemsReceived* function, for example, computes a list of all items that have been delivered to the company, and it hence relies on the Delivery transaction type (*normaliseItems* and *isMe* are also defined in Appendix C.2.2):

```
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
```

Report	Result
Me	The special Me entity.
Entities	A list of all non-deleted entities.
EntitiesByType	A list of all non-deleted entities of a given type.
ReportNames	A list of names of all non-deleted reports.
ReportNamesByTags	A list of names of all non-deleted reports whose tags contain a given set and do not contain another given set.
ReportTags	A list of all tags used by non-deleted reports.
ContractTemplates	A list of names of all non-deleted contract templates.
ContractTemplatesByType	A list of names of all non-deleted contract templates of a given type.
Contracts	A list of all non-deleted contract instances.
ContractHistory	A list of previous transactions for a given contract instance.
ContractSummary	A list of meta data for a given contract instance.

Figure 11: Internal reports.

del : Delivery = $tr.transaction$,
$\neg(isMe\ del.sender) \wedge isMe\ del.receiver$,
$is \leftarrow del.items$]

Internal reports are reports that are needed either by clients of the system or by contracts. For instance, the *ContractTemplates* report is needed by clients of the system in order to instantiate contracts, and the *Me* report is needed by the two contracts, as we shall see in the following subsection. A list of internal reports, including a short description of what they compute, is summarised in Figure 11. Except for the *Me* report, all internal reports are independent from the custom data model.

External reports are reports that are expected to be rendered directly in clients of the system, but they may also be invoked by contracts. The external reports in our use case are the reports mentioned earlier, namely the income statement, the balance sheet, the cash flow statement, the list of unpaid invoices, and the VAT report. Moreover, we include reports for calulating the list of items in the inventory, and the list of top-n customers, respectively. We include the inventory report below as an example:

```
report : Inventory
report =
 let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
 in
 −− The available items is the list of received items minus the
 −− list of reserved or sold items
 Inventory{availableItems = normaliseItems (itemsReceived ++ itemsSold')}
```

The value *itemsSold* is defined in the domain-specific prelude, similar to the value *itemsReceived*. But unlike *itemsReceived*, the computation takes into account that items can be reserved but not yet delivered. Hence when we check that items are in stock using the inventory report, we also take into account that some items in the inventory may have been sold, and therefore cannot be sold again.

The five standard reports are defined according to the specifications given by Henglein et al. [6, Section 2.1], but for simplicity we do not model fixed costs, depreciation, and fixed assets. We do, however, model multiple currencies, exemplified via Danish Kroner (DKK) and Euro (EUR). This means that financial reports, such as IncomeStatement, provide lists of values of type Money—one for each currency used.

3.3 Contracts

The specification of contracts is divided into three parts: prelude functions (Appendix C.3.1), domain-specific prelude functions (Appendix C.3.2), and contract templates (Appendix C.3.3).

Prelude functions are utility functions similar to the report engine's prelude functions. They are independent from the custom data model, and are automatically added to all POETS instances. The prelude includes standard functions such as *filter*.

Domain-specific prelude functions are utility functions that depend on the custom data model. The *inStock* function, for example, checks whether the items described in a list of order lines are in stock, by querying the *Inventory* report (we assume that the item types are different for each line):

```
fun inStock lines =
  let inv = (reports.inventory ()).availableItems
  in
  all (λl → any (λi → (l.item).itemType ≡ i.itemType ∧
                      (l.item).quantity ≤ i.quantity) inv) lines
```

Contract templates describe the daily activities in the company, and in our μERP use case we only consider a purchase contract and a sales contract. The purchase contract is presented below:

```
name: purchase
type: Purchase
description: "Set up a purchase"

clause purchase(lines : [OrderLine])⟨me : ⟨Me⟩, vendor : ⟨Vendor⟩⟩ =
 ⟨vendor⟩ Delivery(sender s, receiver r, items i)
   where s ≡ vendor ∧ r ≡ me ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 when IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ vendor ∧ r ≡ me ∧ sl ≡ lines
   due within 1Y
 then
 payment(lines, vendor, 14D)⟨me⟩

clause payment(lines : [OrderLine], vendor : ⟨Vendor⟩, deadline : Duration)
                ⟨me : ⟨Me⟩⟩ =
 if null lines then
```

```
  fulfilment
 else
  ⟨me⟩ BankTransfer(sender s, receiver r, money m)
   where s ≡ me ∧ r ≡ vendor ∧ checkAmount m lines
   due within deadline
   remaining newDeadline
  then
  payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

The contract describes a simple workflow, in which the vendor delivers items, possibly followed by an invoice, which in turn is followed by a bank transfer of the company. Note how the *me* parameter in the contract template body refers to the value from the domain-specific prelude, which in turn invokes the *Me* report. Note also how the *payment* clause template is recursively defined in order to accommodate for potentially different currencies. That is, the total payment is split up in as many bank transfers as there are currencies in the purchase.

The sales contract is presented below:

```
name: sale
type: Sale
description: "Set up a sale"

clause sale(lines : [OrderLine])⟨me : ⟨Me⟩, customer : ⟨Customer⟩⟩ =
 ⟨me⟩ IssueInvoice(sender s, receiver r, orderLines sl)
  where s ≡ me ∧ r ≡ customer ∧ sl ≡ lines ∧ inStock lines
  due within 1H
 then
 payment(lines, me, 10m)⟨customer⟩
 and
 ⟨me⟩ Delivery(sender s, receiver r, items i)
  where s ≡ me ∧ r ≡ customer ∧ i ≡ map (λx → x.item) lines
  due within 1W
 then
 repair(map (λx → x.item) lines, customer, 3M)⟨me⟩

clause payment(lines : [OrderLine], me : ⟨Me⟩, deadline : Duration)
                ⟨customer : ⟨Customer⟩⟩ =
 if null lines then
  fulfilment
 else
  ⟨customer⟩ Payment(sender s, receiver r, money m)
   where s ≡ customer ∧ r ≡ me ∧ checkAmount m lines
   due within deadline
   remaining newDeadline
  then
  payment(remainingOrderLines m lines, me, newDeadline)⟨customer⟩

clause repair(items : [Item], customer : ⟨Customer⟩, deadline : Duration)
              ⟨me : ⟨Me⟩⟩ =
 when RequestRepair(sender s, receiver r, items i)
  where s ≡ customer ∧ r ≡ me ∧ subset i items
  due within deadline
```

remaining *newDeadline*
then
⟨*me*⟩ Repair(*sender s*, *receiver r*, *items i'*)
where $s \equiv me \wedge r \equiv customer \wedge i \equiv i'$
due within $5D$
and
repair(*items*, *customer*, *newDeadline*)⟨*me*⟩

contract = *sale*(*orderLines*)⟨*me*, *customer*⟩

The contract describes a workflow, in which the company issues an invoice to the customer—but only if the items on the invoice are in stock. The issuing of invoice is followed by an immediate (within an hour) payment by the customer to the company, and a delivery of goods by the company within a week. Moreover, we also model the repair guarantee mentioned in the introduction.

3.4 Bootstrapping the System

The previous subsections described the specification code for μERP. Since data definitions, report specifications, and contract specifications are added to the system at run-time, μERP is instantiated by invoking the following sequence of services on an initially empty POETS instance:

1. Add data definitions in Appendix C.1 via *addDataDefs*.
2. Create a designated Me entity via *createEntity*.
3. Add report specifications via *addReport*.
4. Add contract specifications via *createTemplate*.

Hence, the event log will, conceptually, have the form (we write the value of the field internalTimeStamp before each event):

t_1: AddDataDefs{defs = `"ResourceType is ..."`}

t_2: CreateEntity{ent = e_1, recordType = `"Me"`, data = Me}

t_3: CreateReport{name = `"Me"`, description = `"Returns the ..."`,
code = `"name: Me\n ..."`, tags = [`"internal"`,`"entity"`]}
⋮

t_i: CreateReport{name = `"TopNCustomers"`, description = `"A list ..."`,
code = `"name: TopNCustomers\n ..."`,
tags = [`"external"`,`"financial"`,`"crm"`]}

t_{i+1}: CreateContractDef{name = `"Purchase"`, recordType = `"Purchase"`,
code = `"name: purchase\n ..."`, description = `"Set up ..."`}

t_{i+2}: CreateContractDef{name = `"Sale"`, recordType = `"Sale"`,
code = `"name: sale\n ..."`, description = `"Set up a sale"`}

for some increasing timestamps $t_1 < t_2 < \ldots < t_{i+2}$. Note that the entity value e_1 of the CreateEntity event is automatically generated by the entity store, as described in Section 2.3.

After executing these operations, the system is operational. That is, (i) customers and vendors can be managed via *createEntity*, *updateEntity*, and *deleteEntity*, (ii) contracts can be instantiated, updated, concluded, and inspected via *createContract*, *updateContract*, *concludeContract*, and *getContract* respectively, (iii) transactions can be registered via *registerTransaction*, and (iv) reports can be queried via *queryReport*.

For example, if a sale is initiated with a new customer John Doe, starting at time t, then the following events will be added to the event log:

```
t_{i+3}: CreateEntity{ent = e_2, recordType = "Customer", data = Customer{
          name = "John Doe", address = Address{
           string = "Universitetsparken 1", country = Denmark}}}

t_{i+4}: CreateContract{contractId = 0, contract = Sale{
         startDate = t, templateName = "sale", customer = e_2,
         orderLines = [OrderLine{
           item = Item{itemType = Bicycle{model = "Avenue"}, quantity = 1.0},
           unitPrice = Money{currency = DKK, amount = 4000.0},
           vatPercentage = 25.0}]}}
```

That is, first the customer entity is created, and then we can instantiate a new sales contract. In this particular sale, one bicycle of the model "Avenue" is sold at a unit price of 4000 DKK, with an additional VAT of 25 percent. Note that the contract id 0 of the CreateContract is automatically generated and that the start time t is explicitly given in the CreateContract's startDate field independent from the internalTimeStamp field.

Following the events above, if the contract is executed successfully, events of type IssueInvoice, Delivery, and Payment will persisted in the event log with appropriate values—in particular, the payment will be 5000 DKK.

4 Implementation Aspects

In this section we briefly discuss some of the implementation techniques used in our implementation of POETS. POETS is implemented in Haskell [9], and the logical structure of the implementation reflects the diagram in Figure 2, that is each component is implemented as a separate Haskell module.

4.1 External Interface

The external interface to the POETS system is implemented in a separate Haskell module. We currently use Thrift [15] for implementing the communication layer between the server and its clients, but other communication layers can in principle be used. Changing the communication layer will only require a change in one module.

Besides offering an abstract, light-weight interface to communication, Thrift enables type-safe communication. The types and services of the server are specified in a language-independent description language, from which Haskell code is generated (or code in other languages for the clients). For example, the external interface to querying a report can be specified as follows:

```
Value queryReport(
  1 : string name      // name of the report to execute
  2 : list<Value> args // input arguments
) throws (
  1 : ReportNotFoundException notFound
  2 : RunTimeException runtime
  3 : TypeException type
)
```

From this specification, Thrift generates the Haskell code for the server interface, and implementing the interface amounts to supplying a function of the type $String \to [Value] \to IO\ Value$—namely the query function.

4.2 Domain-Specific Languages

The main ingredient of the POETS implementation is the implementation of the domain-specific languages. What is interesting in that respect—compared to implementations of domain-specific languages in isolation of each other—is the common core shared by the languages, in particular types and values.

In order to reuse and extend the structure of types and values in the report language and the contract language, we make use of the *compositional data types* [2] library. Compositional data types take the *data types as fixed points* [11] view on abstract syntax trees (ASTs), namely a separation of the recursive structure of ASTs from their signatures. As an example, we define the signatures of types from Section 2.1.1 as follows:

$$\begin{aligned}
&\textbf{type}\ \mathit{RecordName}\ &&= \mathit{String}\\
&\textbf{data}\ \mathit{TypeConstant}\ a\ &&= \mathit{TBool} \mid \mathit{TInt} \mid \cdots\\
&\textbf{data}\ \mathit{TypeRecord}\ a\ &&= \mathit{TRecord}\ \mathit{RecordName}\\
&\textbf{data}\ \mathit{TypeList}\ a\ &&= \mathit{TList}\ a\\
&\textbf{data}\ \mathit{TypeEnt}\ a\ &&= \mathit{TEnt}\ \mathit{RecordName}
\end{aligned}$$

The signature for the types of the data model is then obtained by combining the individual signatures above $TSig = TypeConstant :+: TypeRecord :+: TypeList :+: TypeEnt$, where $(:+:) :: (* \to *) \to (* \to *) \to * \to *$ is the sum of two functors. Finally, the data type for ASTs of types can be defined by tying the recursive knot $T = Term\ TSig$, where $Term :: (* \to *) \to *$ is the functor fixed point.

Recursive functions over ASTs are defined as type classes, with one instance per atomic signature. For instance, a pretty printer for types can be defined as follows:

$$\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Render}\ f\ \textbf{where}\\
&\quad \mathit{render} :: f\ \mathit{String} \to \mathit{String}
\end{aligned}$$

```
instance Render TypeConstant where
  render TInt  = "Int"
  render TBool = "Bool"
  ...
instance Render TypeRecord where
  render (TRecord r) = r
instance Render TypeList where
  render (TList τ) = "[" ++ τ ++ "]"
instance Render TypeEnt where
  render (TEnt r) = "<" ++ r ++ ">"
```

and pretty printing of terms is subsequently obtained by lifting the *render* algebra to a catamorphism, that is a function of type $Render\ f \Rightarrow Term\ f \to String$.

Extendability The first benefit of the approach above is that we can extend the signature for types to fit, for example, the contract language as in Figure 9:

```
type TypeVar         = String
data TypeUnit a      = TUnit
data TypeVar a       = TVar TypeVar
data TypeFunction a = TFunction a a
```

Extending the pretty printer amounts to only providing the new cases:

```
instance render TypeUnit where
  render TUnit = "()"
instance render TypeVar where
  render (TVar α) = α
instance render TypeFunction where
  render (TFunction τ1 τ2) = τ1 ++ " -> " ++ τ2
```

A similar modular encoding is used for the language of values:

```
data Value a = VInt Int | VBool Bool | VString String | ···
```

and the signature of expressions in the contract language of Figure 9 can be obtained by providing the extensions compared to the language of values:

```
type Var = String
data Exp a = EVar Var | ELambda Var a | EApply a a | ···
```

That is, *Term* (*Exp* :+: *Value*) represents the type of ASTs for expressions of the contract language. Reusing the signature for (core) values means that the values of Section 2.1.2, which are provided as input to the system for instance in the *registerTransaction* function, can be automatically coerced to the richer language of expressions. That is, values of type *Term Value* can be readily used as values of type *Term* (*Exp* :+: *Value*), without explicit copying or translation.

Notice the difference in the granularity of (core) value signatures and (core) type signatures: types are divided into three signatures, whereas values are in

one signature. The rule of thumb we apply is to divide signatures only when a function needs the granularity. For instance, the type inference algorithm used in the report language and the contract language implements a simplification procedure [5], which reduces type constraints to *atomic* type constraints. In order to guarantee this transformation invariant statically, we hence need a signature of atomic types, namely $\mathit{TypeConstant} :+: \mathit{TypeVar}$, which prompts the finer granularity on types.

Syntactic sugar Besides enabling a common core of ASTs and functions on them, compositional data type enable AST transformations where the invariant of the transformation is witnessed by the type. Most notably, desugaring can be implemented by providing a signature for syntactic sugar:

$$\textbf{data}\ \mathit{ExpSug}\ a = \mathit{ELet}\ \mathit{Var}\ a\ a \mid \cdots$$

as well as a transformation to the core signature:

$$\begin{aligned}&\textbf{instance}\ \mathit{Desugar}\ \mathit{ExpSug}\ \textbf{where}\\ &\quad \mathit{desugar}\ (\mathit{ELet}\ x\ e_1\ e_2) = \mathit{ELam}\ x\ e_2\ `\mathit{EApp}`\ e_1\\ &\quad \cdots\end{aligned}$$

This approach yields a desugaring function of the type $\mathit{Term}\ (\mathit{ExpSug} :+: \mathit{Exp} :+: \mathit{Value}) \to \mathit{Term}\ (\mathit{Exp} :+: \mathit{Value})$, which witnesses that the syntactic sugar has indeed been removed.

Moreover, since we define the desugaring translation in the style of a *term homomorphism* [2], we automatically get a lifted desugaring function that propagates AST annotations, such as source code positions, to the desugared term. This means, for instance, that type error messages can provide detailed source position information also for terms that originate from syntactic sugar.

5 Conclusion

We have presented an extended and generalised version of the POETS architecture [6], which we have fully implemented. We have presented domain-specific languages for specifying the data model, reports, and contracts of a POETS instance, and we have demonstrated an application of POETS in a small use case. The use case demonstrates the conciseness of our approach—Appendix C contains the complete source needed for a running system—as well as the domain-orientation of our specification languages. We believe that non-programmers should be able to read and understand the data model of Appendix C.1, to some extent the contract specifications of Appendix C.3.3, and to a lesser extent the reports of Appendix C.2 (after all, reports describe computations).

5.1 Future Work

With our implementation and revision of POETS we have only taken the first steps towards a software system that can be used in practice. In order to properly verify our hypothesis that POETS is practically feasible, we want to conduct a

larger use case in a live, industrial setting. Such use case will both serve as a means of testing the technical possibilities of POETS, that is whether we can model and implement more complex scenarios, as well as a means of testing our hypothesis that the use of domain-specific languages shortens the gap between requirements and implementation.

Expressivity As mentioned above, a larger and more realistic use case is needed in order to fully evaluate POETS. In particular, we are interested in investigating whether the data model, the report language, and the contract language have sufficient expressivity. For instance, a possible extension of the data model is to introduce finite maps. Such extension will, for example, simplify the reports from our μERP use case that deal with multiple currencies. Moreover, finite maps will enable a modelling of resources that is closer in structure to that of Henglein et al. [6].

Another possible extension is to allow types as values in the report language. For instance, the *EntitiesByType* report in Appendix C.2.3 takes a string representation of a record type, rather than the record type itself. Hence the function cannot take subtypes into account, that is if we query the report with input A, then we only get entities of declared type A and not entities of declared subtypes of A.

Rules A rule engine is a part of our extended architecture (Figure 2), however it remains to be implemented. The purpose of the rule engine is to provide rules—written in a separate domain-specific language—that can constrain the values that are accepted by the system. For instance, a rule might specify that the items list of a Delivery transaction always be non-empty.

More interestingly, the rule engine will enable values to be *inferred* from the rules in the engine. For instance, a set of rules for calculating VAT will enable the field vatPercentage of an OrderLine to be inferred automatically in the context of a Sale record. That is, based on the information of a sale and the items that are being sold, the VAT percentage can be calculated automatically for each item type.

The interface to the rule engine will be very simple: A record value, as defined in Section 2.1.2, with zero or more *holes* is sent to the engine, and the engine will return either (i) an indication that the record cannot possibly fulfil the rules in the engine, or (ii) a (partial) substitution that assigns inferred values to (some of) the holes of the value as dictated by the rules. Hence when we, for example, instantiate the sale of a bicycle in Section 3.4, then we first let the rule engine infer the VAT percentage before passing the contract meta data to the contract engine.

Forecasts A feature of the contract engine, or more specifically of the reduction semantics of contract instances, is the possibility to retrieve the state of a running contract at any given point in time. The state is essentially the AST of a contract clause, and it describes what is currently expected in the contract, as well as what is expected in the future.

Analysing the AST of a contract enables the possibility to do *forecasts*, for instance to calculate the expected outcome of a contract or the items needed for delivery within the next week. Forecasts are, in some sense, dual to reports. Reports derive data from transactions, that is facts about what has previously happened. Forecasts, on the other hand, look into the future, in terms of calculations over running contracts. We have currently implemented a single forecast, namely a forecast that lists the set of immediately expected transactions for a given contract. A more ambitious approach is to devise (yet another) language for writing forecasts, that is functions that operate on contract ASTs.

Practicality In order to make POETS useful in practice, many features are still missing. However, we see no inherent difficulties in adding them to POETS compared to traditional ERP architectures. To mention a few: (i) security, that is authorisation, users, roles, etc.; (ii) module systems for the report language and contract language, that is better support for code reuse; and (iii) check-pointing of a running system, that is a dump of the memory of a running system, so the event log does not have to be replayed from scratch when the system is restarted.

Acknowledgements We are grateful to Fritz Henglein for many fruitful discussions and for convincing us of the *POETS approach* in the first place. Morten Ib Nielsen and Mikkel Jønsson Thomsen both contributed to our implementation and design of POETS, for which we are thankful. Lastly, we thank the participants of the DIKU course "POETS Summer of Code" for valuable input.

Bibliography

[1] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, 2006. ISSN 1433-2779. doi: 10.1007/s10009-006-0010-1.

[2] P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, New York, NY, USA, 2011. ACM. doi: 10.1145/2036918.2036930.

[3] A. J. Bernstein and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001. ISBN 0321185579.

[4] N. E. Fuchs, K. Kaljurand, and T. Kuhn. Attempto Controlled English for Knowledge Representation. In C. Baroglio, P. A. Bonatti, J. Maluszynski, M. Marchiori, A. Polleres, and S. Schaffert, editors, *Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, volume 5224, pages 104–124. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85656-6. doi: 10.1007/978-3-540-85658-0˙3.

[5] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990. ISSN 0304-3975. doi: 10.1016/0304-3975(90)90144-7.

[6] F. Henglein, K. F. Larsen, J. G. Simonsen, and C. Stefansen. POETS: process-oriented event-driven transaction system. *The Journal of Logic and Algebraic Programming*, 78:381–401, 2009. ISSN 1567-8326. doi: 10.1016/j.jlap.2008.08.007.

[7] T. Hvitved, F. Klaedtke, and E. Zalinescu. A trace-based model for multi-party contracts. *Journal of Logic and Algebraic Programming*, 81(2):72–98, 2012. ISSN 1567-8326. doi: 10.1016/j.jlap.2011.04.010.

[8] M. Jønsson Thomsen. Using Controlled Natural Language for specifying ERP Requirements. Master's thesis, University of Copenhagen, Department of Computer Science, 2010.

[9] S. Marlow. Haskell 2010 Language Report, 2010.

[10] W. E. McCarthy. The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. *The Accounting Review*, LVII(3):554–578, 1982.

[11] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961˙7.

[12] M. Nissen and K. F. Larsen. FunSETL—Functional Reporting For ERP Systems. In O. Chitil, editor, *Procedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL)*, pages 268–289, 2007.

[13] A. Ohori. A Polymorphic Record Calculus and Its Compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. ISSN 0164-0925. doi: 10.1145/218570.218572.

[14] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0262162091.

[15] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, Palo Alto, CA, 2007.

[16] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. doi: 10.1017/S0960129500001560.

[17] J. J. Weygandt, D. E. Kieso, and P. D. Kimmel. *Financial Accounting, with Annual Report*. Wiley, 2004.

A Predefined Ontology

A.1 Data

Data is abstract.

A.2 Event

Event is abstract.
Event has a Timestamp
called internalTimeStamp.

 # Add data definitions to the system
AddDataDefs is an Event.
AddDataDefs has a String called defs.

 # Events associated with entities
EntityEvent is an Event.
EntityEvent is abstract.
EntityEvent has a Data entity called ent.

 # Put entity event
PutEntity is an EntityEvent.
PutEntity has Data.
PutEntity is abstract.

 # Create entity event
CreateEntity is a PutEntity.
CreateEntity has a String called recordType.

 # Update entity event
UpdateEntity is a PutEntity.

 # Delete entity event
DeleteEntity is an EntityEvent.

 # Events associated with a report definition
ReportEvent is an Event.
ReportEvent has a String called name.

 # Put report definition event
PutReport is a ReportEvent.
PutReport is abstract.
PutReport has a String called code.
PutReport has a String called description.
PutReport has a list of String called tags.

 # Create report definition event
CreateReport is a PutReport.

 # Update report definition event
UpdateReport is a PutReport.

 # Delete report definition event
DeleteReport is a ReportEvent.

 # Events associated with a contract template
ContractDefEvent is an Event.
ContractDefEvent has a String called name.

 # Put contract template event
PutContractDef is a ContractDefEvent.
PutContractDef is abstract.
PutContractDef has a String called recordType.
PutContractDef has a String called code.
PutContractDef has a String called description.

 # Create contract template event
CreateContractDef is a PutContractDef.

 # Update contract template event
UpdateContractDef is a PutContractDef.

 # Delete contract template event
DeleteContractDef is a ContractDefEvent.

 # Events associated with a contract
ContractEvent is an Event.
ContractEvent is abstract.
ContractEvent has an Int called contractId.

 # Put contract event
PutContract is a ContractEvent.
PutContract has a Contract.
PutContract is abstract.

 # Create contract event
CreateContract is a PutContract.

 # Update contract event
UpdateContract is a PutContract.

 # Conclude contract event
ConcludeContract is a ContractEvent.

 # Transaction super class
TransactionEvent is a ContractEvent.
TransactionEvent has a Timestamp.
TransactionEvent has a Transaction.

A.3 Transaction

Transaction is abstract.

A.4 Report

Report is abstract.

A.5 Contract

Contract is abstract.
Contract has a Timestamp called startDate.
Contract has a String called templateName.

B Static and Dynamic Semantics of the Report Language

B.1 Types, Type Constraints and Type Schemes

The following grammar describes the type expressions that are used in the report language:

$$\tau ::= r \mid \alpha \mid \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \textbf{Char} \mid \textbf{Timestamp} \mid \textbf{Duration} \\ \mid \textbf{DurationTimestamp} \mid [\tau] \mid \langle r \rangle \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid (\tau_1, \tau_2) \mid ()$$

where r ranges over record names and α over type variables.

The report language is polymorphically typed and permits to put constraints on types, for example, subtyping constraints. The language of type constraints is defined as follows:

$$C ::= \tau_1 <: \tau_2 \mid \tau_1.f : \tau_2 \mid \mathsf{Eq}(\tau) \mid \mathsf{Ord}(\tau)$$

Intuitively, these constraints can be interpreted as follows:

- A *subtype constraint* of the form $\tau_1 <: \tau_2$ requires τ_1 to be a subtype of τ_2,
- a *field constraint* of the form $\tau_1.f : \tau_2$ requires τ_1 to be a record type containing a field f of type τ_2,
- an *equality constraint* of the form $\mathsf{Eq}(\tau)$ requires the type τ to have an equality predicate $\equiv$ defined on it, and
- an *order constraint* of the form $\mathsf{Ord}(\tau)$ requires the type τ to have order predicates ($<$, $\leq$) defined on it.

In order to accommodate for the polymorphic typing, we have to move from types to *type schemes*. Type schemes are of the form $\forall\overline{\alpha}.\overline{C} \Rightarrow \tau$, that is, a type with a universal quantification over a sequence of type variables, restricted by a sequence of constraints. We abbreviate $\forall\,\langle\rangle\,.\overline{C} \Rightarrow \tau$ by writing $\overline{C} \Rightarrow \tau$, and $\langle\rangle \Rightarrow \tau$ by τ. The *universal closure* of a type scheme $\overline{C} \Rightarrow \tau$, that is, $\forall\overline{\alpha}.\overline{C} \Rightarrow \tau$ for $\overline{\alpha}$ the free variables $\mathsf{fv}(\overline{C}, \tau)$ in $\overline{C}$ and τ, is abbreviated by $\forall\overline{C} \Rightarrow \tau$.

B.2 Built-in Symbols

In the following we give an overview of the constants provided by the language. Along with each constant c we will associate a designated type scheme σ_c.

One part of the set of constants consists of literals: Numeric literals $\mathbb{R}$, Boolean literals $\{\textbf{True}, \textbf{False}\}$, character literals $\{\texttt{'a'}, \texttt{'b'}, \ldots\}$, and string literals. Each literal is associated with its obvious type: **Int** (respectively **Real**), **Bool**, **Char**, respectively **String**. Moreover, we also have entity values $\langle r, e\rangle$ of type $\langle r\rangle$ with e a unique identifier.

In the following we list the remaining built-in constants along with their respective type schemes. Many of the given constant symbols are used as mixfix operators. This is indicated by placeholders $\_$. For example a binary infix operator $\circ$ is then written as a constant $\_ \circ \_$. For a constant c we write $c : \overline{C} \Rightarrow \tau$ in order to indicate the type scheme $\sigma_c = \forall\overline{C} \Rightarrow \tau$ assigned to c.

$$\_ \circ \_ : \alpha <: \mathbf{Real} \Rightarrow \alpha \to \alpha \to \alpha \qquad \forall \circ \in \{+, -, *\}$$
$$\_/\_ : \mathbf{Real} \to \mathbf{Real} \to \mathbf{Real}$$
$$\_ \equiv \_ : \mathsf{Eq}(\alpha) \Rightarrow \alpha \to \alpha \to \mathbf{Bool}$$
$$\_ \circ \_ : \mathsf{Ord}(\alpha) \Rightarrow \alpha \to \alpha \to \mathbf{Bool} \qquad \forall \circ \in \{>, \geq, <, \leq\}$$
$$\_ \circ \_ : \alpha <: \mathbf{DurationTimestamp} \Rightarrow \alpha \to \mathbf{Duration} \to \alpha \qquad \forall \circ \in \{\langle + \rangle, \langle - \rangle\}$$

$$r\,\{f_1 = \_, \ldots, f_n = \_\} : \tau_1 \to \ldots \tau_n \to r \qquad \text{where } \rho(r) = \{(f_1, \tau_1), \ldots, (f_n, \tau_n)\}$$
$$\_.f : \alpha.f : \beta \Rightarrow \alpha \to \beta$$
$$\_\,\{f_1 = \_, \ldots, f_n = \_\} : \alpha.f_1 : \alpha_1, \ldots, \alpha.f_n : \alpha_n \Rightarrow \alpha \to \alpha_1 \to \ldots \to \alpha_n \to \alpha$$

$$\neg : \mathbf{Bool} \to \mathbf{Bool}$$
$$\_ \circ \_ : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \qquad \forall \circ \in \{\wedge, \vee\}$$
$$\mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_ : \mathbf{Bool} \to \alpha \to \alpha \to \alpha$$

$$[] : [\alpha]$$
$$\_ \# \_ : \alpha \to [\alpha] \to [\alpha]$$
$$\mathbf{fold} : \mathsf{Eq}(\beta) \Rightarrow (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$

$$() : ()$$
$$(\_, \_) : \alpha \to \beta \to (\alpha, \beta)$$
$$\mathbf{Inl} : \alpha \to \alpha + \beta$$
$$\mathbf{Inr} : \beta \to \alpha + \beta$$
$$\mathbf{case} : \alpha + \beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$$
$$\_.1 : (\alpha, \beta) \to \alpha$$
$$\_.2 : (\alpha, \beta) \to \beta$$

$$\_! : \langle r \rangle \to r$$
$$\_@ : \langle r \rangle \to r$$

$$\langle\langle \_ - \_ - \_ \quad \_ : \_ : \_ \rangle\rangle : \underbrace{\mathbf{Int} \to \cdots \to \mathbf{Int}}_{6\times} \to \mathbf{Timestamp}$$
$$\langle\langle \_\ s, \_\ min, \_\ h, \_\ d, \_\ w, \_\ mon, \_\ y \rangle\rangle : \underbrace{\mathbf{Int} \to \cdots \to \mathbf{Int}}_{7\times} \to \mathbf{Duration}$$

$$\mathbf{error} : \mathbf{String} \to \alpha$$

We assume that there is always defined a record type Event which is the type of an event stored in the central *event log* of the system. The list of all events in the event log can be accessed by the following constant:

$$\mathbf{events} : [\mathsf{Event}]$$

When considering built-in constants, we also distinguish between *defined functions* f and *constructors* F. Constructors are the constants $\langle\langle\_ - \_ - \_ \quad \_ : \_ : \_\rangle\rangle$, $\langle\langle\_\ s, \_\ min, \_\ h, \_\ d, \_\ w, \_\ mon, \_\ y\rangle\rangle$, $r\ \{f_1 = \_, \ldots, f_n = \_\}$, $\#$, $[]$, $()$, $(\_, \_)$, **Inl**, **Inr** and **error** as well as all literals. The remaining constants are defined functions.

Derived from its type scheme we can also assign an *arity* $\mathsf{ar}(c)$ to each constant c by defining $\mathsf{ar}(c)$ as the largest n such that $\sigma_c = \forall\overline{\alpha}.\overline{C} \Rightarrow \tau_1 \to \tau_2 \to \cdots \to \tau_{n+1}$

B.3 Type System

Before we can present the type system of the report language, we have to give the rules for the type constraints. To this end we extend the subtyping judgement $\mathcal{R} \vdash \tau_1 <: \tau_2$ for values from Figure 4. The constraint entailment judgement $\mathcal{R}, \mathcal{C} \Vdash C$ states that a constraint C follows from the set of constraints $\mathcal{C}$ and the record typing environment $\mathcal{R}$.

The type constraint entailment judgement $\mathcal{R}, \mathcal{C} \Vdash C$ is straightforwardly extended to sequences of constraints $\overline{C}$. We define that $\mathcal{R}, \mathcal{C} \Vdash C_1, \ldots, C_n$ iff $\mathcal{R}, \mathcal{C} \Vdash C_i$ for all $1 \leq i \leq n$.

The type system of the report language is a straightforward polymorphic lambda calculus extended with type constraints. The typing judgement for the report language is written $\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \sigma$, where $\mathcal{R}$ is a record typing environment, $\mathcal{C}$ a set of type constraints, Γ a type environment, e an expression and σ a type scheme. The inference rules for this judgement are given in Figure 13.

A typing $\mathcal{R}, \mathcal{C}', \Gamma' \vdash e : \tau'$ is an instance of $\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau$ iff there is a substitution S such that $\Gamma' \supseteq \Gamma S$, $\tau' = \tau S$, and $\mathcal{R}, \mathcal{C}' \Vdash \mathcal{C}S$. Deriving from that we say that the type scheme $\sigma' = \forall\overline{\alpha}'.\overline{C}' \Rightarrow \tau'$ is an instance of $\sigma = \forall\overline{\alpha}.\overline{C} \Rightarrow \tau$, written $\sigma' < \sigma$, iff there is a substitution S with $\mathrm{dom}(S) = \alpha$ such that $\tau' = \tau S$ and $\mathcal{R}, \mathcal{C}' \Vdash \mathcal{C}S$.

Top-level function definitions are of the form

$$f\ x_1\ \ldots\ x_n = e$$

and can be preceded by an explicit type signature declaration of the form $f : \sigma$.

Depending on whether an explicit type signature is present, the following inference rules define the typing of top-level function definitions:

$$\frac{\mathcal{R}, \mathcal{C} \cup \overline{C}, \Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash e : \tau \qquad \overline{\alpha} \notin \mathsf{fv}(\mathcal{C}) \cup \mathsf{fv}(\Gamma)}{\mathcal{R}, \mathcal{C}, \Gamma \vdash f\ x_1\ \ldots\ x_n = e : \forall\overline{\alpha}.\overline{C} \Rightarrow \tau_1 \to \cdots \to \tau_n \to \tau} \text{ (Fun)}$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash f\ x_1\ \ldots\ x_n = e : \sigma \qquad \sigma' < \sigma}{\mathcal{R}, \mathcal{C}, \Gamma \vdash f : \sigma'; f\ x_1\ \ldots\ x_n = e : \sigma'} \text{ (Fun')}$$

B.4 Operational Semantics

In order to simplify the presentation of the operational semantics we assign to each constant c of the language its set of *strict argument positions* $\mathsf{strict}(c) \subseteq \{1, \ldots, \mathsf{ar}(c)\}$:

$$\frac{C \in \mathcal{C}}{\mathcal{R}, \mathcal{C} \Vdash C}\ (\text{Hyp}) \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash r_1 <: r_2}\ (<: \text{Rec})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \tau <: \tau}\ (<: \text{Refl}) \qquad \frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_2 <: \tau_3}{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_3}\ (<: \text{Trans})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash \tau_2 \to \tau_3 <: \tau_1 \to \tau_4}\ (<: \text{Fun}) \qquad \frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2}{\mathcal{R}, \mathcal{C} \Vdash [\tau_1] <: [\tau_2]}\ (<: \text{List})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash \tau_1 + \tau_3 <: \tau_2 + \tau_4}\ (<: \text{Sum})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1 <: \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_3 <: \tau_4}{\mathcal{R}, \mathcal{C} \Vdash (\tau_1, \tau_3) <: (\tau_2, \tau_4)}\ (<: \text{Prod})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Int} <: \mathbf{Real}}\ (<: \text{Num})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Timestamp} <: \mathbf{DurationTimestamp}}\ (<: \text{Timestamp})$$

$$\frac{}{\mathcal{R}, \mathcal{C} \Vdash \mathbf{Duration} <: \mathbf{DurationTimestamp}}\ (<: \text{Duration})$$

$$\frac{(f, \tau) \in \rho(r)}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash r.f : \tau}\ (\text{Field})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \tau_1.f : \tau_2 \qquad \mathcal{R}, \mathcal{C} \Vdash \tau_1' <: \tau_1}{\mathcal{R}, \mathcal{C} \Vdash \tau_1'.f : \tau_2}\ (\text{Field Prop})$$

$$\frac{\tau \in \{\mathbf{Bool}, \mathbf{Int}, \mathbf{Real}, \mathbf{Char}, \mathbf{Duration}, \mathbf{Timestamp}, \mathbf{DurationTimestamp}\}}{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Ord}(\tau)}\ (\text{Ord Base})$$

$$\frac{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Ord}(\tau)}{\mathcal{R}, \mathcal{C} \Vdash \mathsf{Eq}(\tau)}\ (\text{Eq Ord}) \qquad \frac{r \in R}{(R, A, F, \rho, \leq), \mathcal{C} \Vdash \mathsf{Eq}(r)}\ (\text{Eq Rec})$$

$$\frac{F \in \{(\cdot, \cdot), +, [\cdot], \langle \cdot \rangle\} \qquad P \in \{\mathsf{Ord}(\cdot), \mathsf{Eq}(\cdot)\} \qquad \forall 1 \leq i \leq n\colon\ \mathcal{R}, \mathcal{C} \Vdash P(\tau_i)}{\mathcal{R}, \mathcal{C} \Vdash P(F(\tau_1, \ldots, \tau_n))}\ (P\ F)$$

Figure 12: Type constraint entailment $\mathcal{R}, \mathcal{C} \Vdash C$.

$$\frac{x : \sigma \in \Gamma}{\mathcal{R}, \mathcal{C}, \Gamma \vdash x : \sigma}\ (\text{Var}) \qquad \frac{}{\mathcal{R}, \mathcal{C}, \Gamma \vdash c : \sigma_c}\ (\text{Const})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau \qquad \mathcal{C} \Vdash \tau <: \tau'}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau'}\ (\text{Sub}) \qquad \frac{\mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \tau\} \vdash e : \tau'}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \lambda x \to e : \tau \to \tau'}\ (\text{Abs})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \mathcal{R}, \mathcal{C}, \Gamma \vdash e_2 : \tau_1}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1\ e_2 : \tau_2}\ (\text{App})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e_1 : \sigma \qquad \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \sigma\} \vdash e_2 : \tau}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}\ (\text{Let})$$

$$\frac{\begin{array}{ll}\mathcal{R}, \mathcal{C}, \Gamma \vdash e : r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : r\} \vdash e_1 : \tau \\ \mathcal{R}, \mathcal{C} \Vdash r <: r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : r'\} \vdash e_2 : \tau\end{array}}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \mathbf{type}\ x = e\ \mathbf{of}\ \{r \to e_1; \_ \to e_2\} : \tau}\ (\text{Type Of})$$

$$\frac{\begin{array}{ll}\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \langle r' \rangle & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \langle r \rangle\} \vdash e_1 : \tau \\ \mathcal{R}, \mathcal{C} \Vdash r <: r' & \mathcal{R}, \mathcal{C}, \Gamma \cup \{x : \langle r' \rangle\} \vdash e_2 : \tau\end{array}}{\mathcal{R}, \mathcal{C}, \Gamma \vdash \mathbf{type}\ x = e\ \mathbf{of}\ \{\langle r \rangle \to e_1; \_ \to e_2\} : \tau}\ (\text{Type Of Ref})$$

$$\frac{\mathcal{R}, \mathcal{C} \cup \overline{C}, \Gamma \vdash e : \tau \qquad \overline{\alpha} \notin \mathsf{fv}(\mathcal{C}) \cup \mathsf{fv}(\Gamma)}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \forall \overline{\alpha}.\overline{C} \Rightarrow \tau}\ (\forall\ \text{Intro})$$

$$\frac{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \forall \overline{\alpha}.\overline{C} \Rightarrow \tau' \qquad \mathcal{R}, \mathcal{C} \Vdash \overline{C}\,[\overline{\alpha}/\overline{\tau}]}{\mathcal{R}, \mathcal{C}, \Gamma \vdash e : \tau'\,[\overline{\alpha}/\overline{\tau}]}\ (\forall\ \text{Elim})$$

Figure 13: Type inference rules for the report language.

$$\begin{aligned}
\mathsf{strict}(_ \circ _) &= \{1, 2\} && \text{for all binary operators } \circ \neq \# \\
\mathsf{strict}(c) &= \{1\} && \forall c \in \{\neg, \mathbf{if}\ _\ \mathbf{then}\ _\ \mathbf{else}\ _, \mathbf{case}, \mathbf{error}, _@, _!\} \\
\mathsf{strict}(_.f) &= \{1\} \\
\mathsf{strict}(_\ \{\overline{f_i = e_i}\}) &= \{1\} \\
\mathsf{strict}(_.i) &= \{1\}
\end{aligned}$$

For all other constraints c for which the above equations do not apply $\mathsf{strict}(c)$ is defined as the empty set $\emptyset$.

Values form a subset of expressions which are fully evaluated at the top-level. Such expressions are also said to be in *weak head normal form* (*whnf*). An expression is in weak head normal form, if it is an application of a built-in function to too few arguments, an application of a constructor, or a lambda abstraction. Moreover, if a value is not of the form **error** v, it is called *defined*:

$$\begin{aligned}
v ::= {} & c\ e_1 \ldots\ e_n && n < \mathsf{ar}(f) \\
| {} & F\ e_1 \ldots\ e_n && n = \mathsf{ar}(F), \forall i \in \mathsf{strict}(F) \quad e_i \text{ is defined value} \\
| {} & \lambda x \to e
\end{aligned}$$

An even more restricted subset of the set of values is the set of *constructor values* which are expressions in *constructor head normal form*. It is similar to weak head normal form, but with the additional restriction, that arguments of a fully applied constructor are in constructor normal form as well:

$$
\begin{array}{lll}
V ::= c\ e_1 \ldots\ e_n & & n < \mathsf{ar}(f) \\
\quad |F\ V_1 \ldots\ V_n & & n = \mathsf{ar}(F), \forall i \in \mathsf{strict}(F) \quad V_i \text{ is defined} \\
\quad |\lambda x \to e & &
\end{array}
$$

To further simplify the presentation we introduce evaluation contexts. The following evaluation context $\mathbb{E}$ corresponds to weak head normal forms:

$$
\begin{array}{lll}
\mathbb{E} ::= [\cdot] \mid \mathbb{E}\ e \mid \textbf{type}\ x = \mathbb{E}\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} & & \\
\quad |c\ e_1\ \ldots e_{i-1}\ \mathbb{E}\ e_{i+1}\ \ldots\ e_n & & i \in \mathsf{strict}(c), n = \mathsf{ar}(c), \\
& & \forall j < i, j \in \mathsf{strict}(c) \colon e_j \text{ is defined value}
\end{array}
$$

The evaluation context $\mathbb{F}$ corresponds to constructor head normal forms:

$$
\begin{array}{lll}
\mathbb{F} ::= [\cdot] \mid \mathbb{E}\ e \mid \textbf{type}\ x = \mathbb{E}\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} & & \\
\quad |f\ e_1\ \ldots e_{i-1}\ \mathbb{E}\ e_{i+1}\ \ldots\ e_n & & i \in \mathsf{strict}(f), n = \mathsf{ar}(f), \\
& & \forall j < i, j \in \mathsf{strict}(f) \colon e_j \text{ is defined value} \\
\quad |F\ V_1\ \ldots V_{i-1}\ \mathbb{F}\ e_{i+1}\ \ldots\ e_n & & n = \mathsf{ar}(F), V_1\ \ldots V_{i-1} \text{ are defined}
\end{array}
$$

Computations take place in a context of an *event log*, i.e. a sequence of values of type Event. In the following definition of the semantics of the report language we use $(ev_i)_{i<n}$ to refer to this sequence, where each ev_i is of the form $r\{\overline{f_j = e_j}\}$ with $r \leq \mathsf{Event}$.

We assume that the Event record type has a field internalTimeStamp that records the time at which the event was added to the log. For each ev_i, we define its extension ev_i' as follows: Each occurrence of an entity value $\langle r, e\rangle$ is replaced by $\langle r, e, t\rangle$ where t is the value of the internalTimeStamp field of ev_i. This will allow us to define the semantics of the contextual dereference operator @. The semantics of both the @ and the ! operator are given by the lookup operator, which is provided by the entity store, compare Section 2.3. In order to retrieve the latest value associated to an entity, we assume the timestamp t_{now} that denotes the current time.

The rules describing the semantics of the report language in the form of a small step transition relation $\to$ are given in Figure 14.

$$\frac{e \to e'}{\mathbb{F}\,[e] \to \mathbb{F}\,[e']}\ \text{(Context)} \qquad \frac{}{\mathbb{F}\,[\textbf{error}\ v] \to \textbf{error}\ v}\ \text{(Error)}$$

$$\frac{}{(\lambda x \to e_1)e_2 \to e_1\,[x/e_2]}\ \text{(Abs)} \qquad \frac{}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \to e_2\,[x/e_1]}\ \text{(Let)}$$

$$\frac{r' \leq r \qquad v = r'\{\dots\}}{\textbf{type}\ x = v\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} \to e_1\,[x/v]}\ \text{(Type suc)}$$

$$\frac{r' \not\leq r \qquad v = r'\{\dots\}}{\textbf{type}\ x = v\ \textbf{of}\ \{r \to e_1; \_ \to e_2\} \to e_2\,[x/v]}\ \text{(Type def)}$$

$$\frac{\begin{array}{l}\text{injection } \phi\colon \{1,\dots,m\} \hookrightarrow \{1,\dots,n\} \qquad e''_i = \begin{cases} e'_{\phi^{-1}(i)} & \text{if } i \in \mathsf{Im}(\phi) \\ e_i & \text{otherwise} \end{cases} \\ \forall j \in \{1,\dots,m\}:\ f'_j = f_{\phi(j)}\end{array}}{r\{f_1 = e_1, \dots, f_n = e_n\}\ \{f'_1 = e'_1, \dots, f'_m = e'_m\} \to r\{f_1 = e''_1, \dots, f_n = e''_n\}}\ \text{(Mod)}$$

$$\frac{}{r\{f_1 = e_1, \dots, f_n = e_n\}.f_i \to e_i}\ \text{(Acc)} \qquad \frac{}{\textbf{if True then}\ e_1\ \textbf{else}\ e_2 \to e_1}\ \text{(If True)}$$

$$\frac{}{\textbf{if False then}\ e_1\ \textbf{else}\ e_2 \to e_2}\ \text{(If False)} \qquad \frac{}{\textbf{case}\ (\textbf{Inl}\ e)\ e_1\ e_2 \to e_1\ e}\ \text{(Case Left)}$$

$$\frac{}{\textbf{case}\ (\textbf{Inr}\ e)\ e_1\ e_2 \to e_2\ e}\ \text{(Case Right)} \qquad \frac{i \in \{1,2\}}{(e_1, e_2).i \to e_i}\ \text{(Proj)}$$

$$\frac{}{\textbf{events} \to [ev_1, ev_2, \dots, ev_n]}\ \text{(Events)} \qquad \frac{}{\textbf{fold}\ e_1\ e_2\ [] \to e_2}\ \text{(Fold Empty)}$$

$$\frac{}{\textbf{fold}\ e_1\ e_2\ (e_3 \,\#\, e_4) \to e_1\ e_3\ (\textbf{fold}\ e_1\ e_2\ e_4)}\ \text{(Fold Cons)}$$

$$\frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e, t\rangle ! \to v}\ \text{(! ignore)} \qquad \frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e\rangle ! \to v}\ \text{(!)}$$

$$\frac{\text{lookup}_{t_{\text{now}}}(e, t) = v}{\langle r, e, t\rangle\ @ \to v}\ \text{(@)} \qquad \frac{\text{lookup}_{t_{\text{now}}}(e, t_{\text{now}}) = v}{\langle r, e\rangle\ @ \to v}\ \text{(@ now)}$$

Figure 14: Small step operational semantics of the report language.

C μERP Specification

C.1 Ontology

C.1.1 Data

ResourceType is Data.
ResourceType is abstract.

Currency is a ResourceType.
Currency is abstract.

DKK is a Currency.
EUR is a Currency.

ItemType is a ResourceType.
ItemType is abstract.

Bicycle is an ItemType.
Bicycle has a String called model.

Resource is Data.
Resource is abstract.

Money is a Resource.
Money has a Currency.
Money has a Real called amount.

Item is a Resource.
Item has an ItemType.
Item has a Real called quantity.

Agent is Data.

Me is an Agent.

Customer is an Agent.
Customer has a String called name.
Customer has an Address.

Vendor is an Agent.
Vendor has a String called name.
Vendor has an Address.

Address is Data.
Address has a String.
Address has a Country.

Country is Data.
Country is abstract.

Denmark is a Country.

OrderLine is Data.
OrderLine has an Item.
OrderLine has Money called unitPrice.
OrderLine has a Real called vatPercentage.

CurrentAssets is Data.
CurrentAssets has a list of Money called currentAssets.
CurrentAssets has a list of Money called inventory.
CurrentAssets has a list of Money called accountsReceivable.
CurrentAssets has a list of Money called cashPlusEquiv.

Liabilities is Data.
Liabilities has a list of Money called liabilities.

Liabilities has a list of Money called accountsPayable.
Liabilities has a list of Money called vatPayable.

Invoice is Data.
Invoice has an Agent called sender.
Invoice has an Agent called receiver.
Invoice has a list of OrderLine called orderLines.

UnpaidInvoice is Data.
UnpaidInvoice has an Invoice.
UnpaidInvoice has a list of Money called remainder.

CustomerStatistics is Data.
CustomerStatistics has a Customer entity.
CustomerStatistics has Money called totalPaid.

C.1.2 Transaction

BiTransaction is a Transaction.
BiTransaction is abstract.
BiTransaction has an Agent entity called sender.
BiTransaction has an Agent entity called receiver.

Transfer is a BiTransaction.
Transfer is abstract.

Payment is a Transfer.
Payment is abstract.
Payment has Money.

CashPayment is a Payment.
CreditCardPayment is a Payment.
BankTransfer is a Payment.

Delivery is a Transfer.
Delivery has a list of Item called items.

IssueInvoice is a BiTransaction.
IssueInvoice has a list of OrderLine called orderLines.

RequestRepair is a BiTransaction.
RequestRepair has a list of Item called items.

Repair is a BiTransaction.
Repair has a list of Item called items.

C.1.3 Report

IncomeStatement is a Report.
IncomeStatement has a list of Money called revenue.
IncomeStatement has a list of Money called costOfGoodsSold.
IncomeStatement has a list of Money called contribMargin.
IncomeStatement has a list of Money called fixedCosts.
IncomeStatement has a list of Money called depreciation.
IncomeStatement has a list of Money called netOpIncome.

BalanceSheet is a Report.
BalanceSheet has a list of Money called fixedAssets.
BalanceSheet has CurrentAssets.
BalanceSheet has a list of Money called totalAssets.
BalanceSheet has Liabilities.
BalanceSheet has a list of Money called ownersEquity.
BalanceSheet has a list of Money called totalLiabilitiesPlusEquity.

CashFlowStatement is a Report.
CashFlowStatement has a list of Payment called expenses.
CashFlowStatement has a list of Payment called revenues.

CashFlowStatement has a list of Money called revenueTotal.
CashFlowStatement has a list of Money called expenseTotal.

UnpaidInvoices is a Report.
UnpaidInvoices has a list of UnpaidInvoice called invoices.

VATReport is a Report.
VATReport has a list of Money called outgoingVAT.
VATReport has a list of Money called incomingVAT.
VATReport has a list of Money called vatDue.

Inventory is a Report.
Inventory has a list of Item called availableItems.

TopNCustomers is a Report.
TopNCustomers has a list of CustomerStatistics.

C.1.4 Contract

Purchase is a Contract.
Purchase has a Vendor entity.
Purchase has a list of OrderLine called orderLines.

Sale is a Contract.
Sale has a Customer entity.
Sale has a list of OrderLine called orderLines.

C.2 Reports

C.2.1 Prelude Functions

$-- \ \textbf{\textit{Arithmetic}}$
$min : (\textbf{Ord}\ a) \Rightarrow a \to a \to a$
$min\ x\ y = \textbf{if}\ x < y\ \textbf{then}\ x\ \textbf{else}\ y$

$max : (\textbf{Ord}\ a) \Rightarrow a \to a \to a$
$max\ x\ y = \textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y$

$-- \ \textbf{\textit{List functions}}$
$null : [a] \to \textbf{Bool}$
$null = \textbf{fold}\ (\lambda e\ r \to \textbf{False})\ \textbf{True}$

$first : a \to [a] \to a$
$first = \textbf{fold}\ (\lambda x\ a \to x)$

$head : [a] \to a$
$head = first\ (\textbf{error}\ \texttt{"'head' applied to empty list"})$

$elemBy : (a \to a \to \textbf{Bool}) \to a \to [a] \to \textbf{Bool}$
$elemBy\ f\ e = \textbf{fold}\ (\lambda x\ a \to a \vee f\ x\ e)\ \textbf{False}$

$elem : (\textbf{Ord}\ a) \Rightarrow a \to [a] \to \textbf{Bool}$
$elem = elemBy\ (\equiv)$

$sum : (a < \mathsf{Real}, \textbf{Int} < a) \Rightarrow [a] \to a$
$sum = \textbf{fold}\ (+)\ 0$

$length : [a] \to \textbf{Int}$
$length = \textbf{fold}\ (\lambda\ x\ y \to y{+}1)\ 0$

$map : (a \to b) \to [a] \to [b]$
$map\ f = \textbf{fold}\ (\lambda x\ a \to (f\ x)\ \#\ a)\ []$

$filter : (a \to \textbf{Bool}) \to [a] \to [a]$
$filter\ f = \textbf{fold}\ (\lambda x\ a \to \textbf{if}\ f\ x\ \textbf{then}\ x\ \#\ a\ \textbf{else}\ a)\ []$

$nupBy : (a \to a \to \textbf{Bool}) \to [a] \to [a]$

```
nupBy f = fold (λx a → x # filter (λ y → ¬ (f x y)) a) []

nup : (Ord a) ⇒ [a] → [a]
nup = nupBy (≡)

all : (a → Bool) → [a] → Bool
all f = fold (λx a → f x ∧ a) True

any : (a → Bool) → [a] → Bool
any f = fold (λx a → f x ∨ a) False

concat : [[a]] → [a]
concat = fold (λx a → x ++ a) []

concatMap : (a → [b]) → [a] → [b]
concatMap f l = concat (map f l)

take : Int → [a] → [a]
take n l = (fold (λx a → if a.2 > 0 then (x # a.1,a.2 − 1) else a) ([],n) l).1

-- Grouping functions
addGroupBy : (a → a → Bool) → a → [[a]] → [[a]]
addGroupBy f a ll =
  let felem l = fold (λ el r → f el a) False l
      run el r =
        if r.1 then (True,el # r.2)
        else if felem el then (True, (a # el) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else [a] # res.2

groupBy : (a → a → Bool) → [a] → [[a]]
groupBy f = fold (addGroupBy f) []

addGroupProj : (Ord b) ⇒ (a → b) → a → [(b,[a])] → [(b,[a])]
addGroupProj f a ll =
  let run el r =
        if r.1 then(True,el # r.2)
        else if el.1 ≡ f a then (True, (el.1,a # el.2) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else  (f a,[a]) # res.2

groupProj : (Ord b) ⇒ (a → b) → [a] → [(b, [a])]
groupProj f = fold (addGroupProj f) []


-- Sorting functions
insertBy : (a → a → Bool) → a → [a] → [a]
insertBy le a l =
  let ins e r =
        if r.1 then (True, e # r.2)
        else if le e a then (True,e # a # r.2)
        else (False, e # r.2)
      res = fold ins (False,[]) l
  in if res.1 then res.2 else a # res.2

insertProj : (Ord b) ⇒ (a → b) → a → [a] → [a]
insertProj proj = insertBy (λx y → proj x ≤ proj y)

insert : (Ord a) ⇒ a → [a] → [a]
insert = insertBy (≤)

sortBy : (a → a → Bool) → [a] → [a]
sortBy le = fold (λe r → insertBy le e r) []

sortProj : (Ord b) ⇒ (a → b) → [a] → [a]
```

```
sortProj proj = sortBy (λx y → proj x ≤ proj y)

sort : (Ord a) ⇒ [a] → [a]
sort = sortBy (≤)


-- Generators for 'lifecycled' data
reports : [PutReport]
reports = nupBy (λpr1 pr2 → pr1.name ≡ pr2.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events, ur.name ≡ cr.name]]

entities : [(⟨Data⟩,String)]
entities = [(ce.ent,ce.recordType) |
  ce : CreateEntity ← events,
  null [de | de : DeleteEntity ← events, de.ent ≡ ce.ent]]

contracts : [PutContract]
contracts = [pc |
  cc : CreateContract ← events,
  pc = first cc [uc | uc : UpdateContract ← events, uc.contractId ≡ cc.contractId],
  null [cc | cc : ConcludeContract ← events, cc.contractId ≡ pc.contractId]]

contractDefs : [PutContractDef]
contractDefs = nupBy (λpcd1 pcd2 → pcd1.name ≡ pcd2.name) [pcd |
  ccd : CreateContractDef ← events,
  pcd : PutContractDef = first ccd [ucd | ucd : ContractDefEvent ← events, ucd.name ≡ ccd.name]]

transactionEvents : [TransactionEvent]
transactionEvents = [tr | tr : TransactionEvent ← events]

transactions : [Transaction]
transactions = [tr.transaction | tr ← transactionEvents]
```

C.2.2 Domain-Specific Prelude Functions

```
-- Check if an agent is the company itself
isMe : ⟨Agent⟩ → Bool
isMe a = a :? ⟨Me⟩

-- Normalise a list of money by grouping currencies together
normaliseMoney : [Money] → [Money]
normaliseMoney ms = [Money{currency = m.1, amount = sum (map (λm → m.amount) m.2)} |
  m ← groupProj (λm → m.currency) ms]

-- Add one list of money from another
addMoney : [Money] → [Money] → [Money]
addMoney m1 m2 = normaliseMoney (m1 ++ m2)

-- Subtract one list of money from another
subtractMoney : [Money] → [Money] → [Money]
subtractMoney m1 m2 = addMoney m1 (map (λm → m{amount = 0 − m.amount}) m2)

-- Produce normalised list of all items given in list
normaliseItems : [Item] → [Item]
normaliseItems its = [Item{itemType = i.1, quantity = sum (map (λis → is.quantity) i.2)} |
  i ← groupProj (λis → is.itemType) its]

-- List of all invoices and their associated contract ID
invoices : [(Int,IssueInvoice)]
invoices = [(tr.contractId,inv) |
  tr ← transactionEvents,
  inv : IssueInvoice = tr.transaction]

-- List of all received invoices and their associated contract ID
invoicesReceived : [(Int,IssueInvoice)]
invoicesReceived =
```

```
  filter (λinv → ¬ (isMe (inv.2).sender) ∧ isMe (inv.2).receiver) invoices

-- List of all sent invoices and their associated contract ID
invoicesSent : [(Int,IssueInvoice)]
invoicesSent = filter (λinv → isMe inv.2.sender ∧ ¬ (isMe inv.2.receiver)) invoices

-- Calculate the total price including VAT on an invoice
invoiceTotal : (a.orderLines : [OrderLine]) ⇒ a → [Money]
invoiceTotal inv = normaliseMoney [line.unitPrice{amount = price} |
  line ← inv.orderLines,
  quantity = line.item.quantity,
  price = ((100 + line.vatPercentage) × line.unitPrice.amount × quantity) / 100]

-- List of all items delivered to the company
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
  del : Delivery = tr.transaction,
  ¬(isMe del.sender) ∧ isMe del.receiver,
  is ← del.items]

-- List of all items that have been sold
itemsSold : [Item]
itemsSold = normaliseItems [line.item | inv ← invoicesSent, line ← inv.2.orderLines]

-- Inventory acquisitions, that is a list of all received items and the unit
-- price of each item, exluding VAT.
invAcq : [(Item,Money)]
invAcq = [(item,line.unitPrice) |
  inv ← invoicesReceived,
  tr ← transactionEvents,
  tr.contractId ≡ inv.1,
  deliv : Delivery = tr.transaction,
  item ← deliv.items,
  line ← inv.2.orderLines,
  line.item.itemType ≡ item.itemType]

-- FIFO costing: Calculate the cost of all sold goods based on FIFO costing.
fifoCost : [Money]
fifoCost = let
   -- Check whether a set of items equals the current set of items in the
   -- inventory. If so, 'take' as many of the inventory items as possible
   -- and add the price of these items to the totals.
   checkInventory y x = let
      invItem = y.1  -- The current item in the inventory
      invPrice = y.2 -- The price of the current item in the inventory
      oldInv = x.1 -- The part of the inventory that has been processed
      item = x.2 -- The item to find in the inventory
      total = x.3 -- The total costs so far
    in
    if item.itemType ≡ invItem.itemType then let
       deltaInv =
         if invItem.quantity ≤ item.quantity then
           []
         else
           [(invItem{quantity = invItem.quantity − item.quantity},invPrice)]
       remainingItem = item{quantity = max 0 (item.quantity − invItem.quantity)}
       price = invPrice{amount = invPrice.amount × (min item.quantity invItem.quantity)}
      in
      (oldInv ++ deltaInv, remainingItem, price # total)
    else
      (oldInv ++ [(invItem,invPrice)], item, total)

   -- Process a sold item
   processSoldItem soldItem x = let
      total = x.1 -- the total costs so far
      inv = x.2   -- the remaning inventory so far
```

```
    y = fold checkInventory ([],soldItem,total) inv
  in
  (y.3,y.1)
in
normaliseMoney ((fold processSoldItem ([],invAcq) itemsSold).1)

-- Outoing VAT
vatOutgoing : [Money]
vatOutgoing = normaliseMoney [price |
  inv ← invoicesReceived,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]

-- Incoming VAT
vatIncoming : [Money]
vatIncoming = normaliseMoney [price |
  inv ← invoicesSent,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]
```

C.2.3 Internal Reports

Me

name: *Me*
description:
Returns the pseudo entity 'Me' that represents the company.
tags: *internal*, *entity*

```
report : ⟨Me⟩
report = head [me | me : ⟨Me⟩ ← map (λe → e.1) entities]
```

Entities

name: *Entities*
description:
A list of all entities.
tags: *internal*, *entity*

```
report : [⟨Data⟩]
report = map (λe → e.1) entities
```

EntitiesByType

name: *EntitiesByType*
description:
A list of all entities with the given type.
tags: *internal*, *entity*

```
report : String → [⟨Data⟩]
report t = map (λe → e.1) (filter (λe → e.2 ≡ t) entities)
```

ReportNames

name: *ReportNames*
description:
A list of names of all registered reports.
tags: *internal*, *report*

```
report : [String]
report = [r.name | r ← reports]
```

ReportNamesByTags

name: *ReportNamesByTags*
description:
A list of reports that have the all **tags** *provided as first argument to the function and none of the* **tags** *provided as second argument.*
tags: *internal*, *report*

```
filt allOf noneOf rep =
  all (λx → elem x rep.tags) allOf ∧
  ¬ (any (λx → elem x rep.tags) noneOf)

report : [String] → [String] → [String]
report allOf noneOf = [r.name | r ← filter (filt allOf noneOf) reports]
```

ReportTags

name: *ReportTags*
description:
A list of **tags** *that are used in registered reports.*
tags: *internal*, *report*

```
report : [String]
report = nup (concatMap (λx → x.tags) reports)
```

ContractTemplates

name: *ContractTemplates*
description:
A list of 'PutContractDef' events for each non−deleted contract template.
tags: *internal*, *contract*

```
report : [PutContractDef]
report = contractDefs
```

ContractTemplatesByType

name: *ContractTemplatesByType*
description:
A list of 'PutContractDef' events for each non−deleted contract template of the given type.
tags: *internal*, *contract*

```
report : String → [PutContractDef]
report r = filter (λx → x.recordType ≡ r) contractDefs
```

Contracts

name: *Contracts*
description:
A list of all running (i.e. non−concluded) contracts.
tags: *internal*, *contract*

```
report : [PutContract]
report = contracts
```

ContractHistory

name: *ContractHistory*
description:
A list of previous transactions for the given contract.
tags: *internal*, *contract*

```
report : Int → [TransactionEvent]
report cid = [transaction |
  transaction : TransactionEvent ← events,
  transaction.contractId ≡ cid]
```

ContractSummary

name: *ContractSummary*
description:
A list of meta data for the given contract.
tags: *internal*, *contract*

```
report : Int → [PutContract]
report cid = [createCon |
  createCon : PutContract ← contracts,
  createCon.contractId ≡ cid]
```

C.2.4 External Reports

IncomeStatement

name: *IncomeStatement*
description:
The Income Statement.
tags: *external*, *financial*

```
-- Revenue
revenue = normaliseMoney [line.unitPrice{amount = amount} |
 inv ← invoicesSent,
 line ← inv.2.orderLines,
 amount = line.unitPrice.amount × line.items.numberOfItems]

costOfGoodsSold = fifoCost
contribMargin = subtractMoney revenue fifoCost
fixedCosts = [] -- For simplicity
depreciation = [] -- For simplicity
netOpIncome = subtractMoney (subtractMoney contribMargin fixedCosts) depreciation

report : IncomeStatement
report = IncomeStatement{
  revenue = revenue,
  costOfGoodsSold = costOfGoodsSold,
  contribMargin = contribMargin,
  fixedCosts = fixedCosts,
  depreciation = depreciation,
  netOpIncome = netOpIncome}
```

BalanceSheet

name: *BalanceSheet*
description:
The Balance Sheet.
tags: *external*, *financial*

```
-- List of all payments and their associated contract ID
payments : [(Int,Payment)]
payments = [ (tr.contractId,payment) |
  tr ← transactionEvents,
  payment : Payment = tr.transaction]

-- List of all received payments and their associated contract ID
paymentsReceived : [(Int,Payment)]
paymentsReceived = filter (λp → ¬ (isMe p.2.sender) ∧ isMe p.2.receiver) payments

-- List of all payments made and their associated contract ID
paymentsMade : [(Int,Payment)]
paymentsMade = filter (λp → isMe p.2.sender ∧ ¬ (isMe p.2.receiver)) payments

cashReceived : [Money]
cashReceived = normaliseMoney (map (λp → p.2.money) paymentsReceived)
```

```
cashPaid : [Money]
cashPaid = normaliseMoney (map (λp → p.2.money) paymentsMade)

netCashFlow : [Money]
netCashFlow = subtractMoney cashReceived cashPaid

depreciation : [Money]
depreciation = [] -- For simplicity

fAssetAcq : [Money]
fAssetAcq = [] -- For simplicity

fixedAssets : [Money]
fixedAssets = subtractMoney fAssetAcq depreciation

inventory : [Money]
inventory =
  let inventoryValue = [price |
        item ← invAcq,
        price = item.2{amount = item.2.amount × item.1.quantity}]
  in
  subtractMoney inventoryValue fifoCost

accReceivable : [Money]
accReceivable =
  let paymentsDue = normaliseMoney [line.unitPrice{amount = amount} |
        inv ← invoicesSent,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashReceived

currentAssets : [Money]
currentAssets = addMoney inventory (addMoney accReceivable netCashFlow)

totalAssets : [Money]
totalAssets = addMoney fixedAssets currentAssets

accPayable : [Money]
accPayable =
  let paymentsDue = [line.unitPrice{amount = amount} |
        inv ← invoicesReceived,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashPaid

vatPayable : [Money]
vatPayable = subtractMoney vatIncoming vatOutgoing

liabilities : [Money]
liabilities = addMoney accPayable vatPayable

ownersEq : [Money]
ownersEq = subtractMoney totalAssets liabilities

totalLiabPlusEq : [Money]
totalLiabPlusEq = addMoney liabilities ownersEq

report : BalanceSheet
report = BalanceSheet{
  fixedAssets = fixedAssets,
  currentAssets = CurrentAssets{
    currentAssets = currentAssets,
    inventory = inventory,
    accountsReceivable = accReceivable,
    cashPlusEquiv = netCashFlow},
```

```
  totalAssets = totalAssets,
  liabilities = Liabilities{
    liabilities = liabilities,
    accountsPayable = accPayable,
    vatPayable = vatPayable},
  ownersEquity = ownersEq,
  totalLiabilitiesPlusEquity = totalLiabPlusEq}
```

CashFlowStatement

name: *CashFlowStatement*
description:
The Cash Flow Statement.
tags: *external*, *financial*

```
sumPayments : [Payment] → [Money]
sumPayments ps = normaliseMoney (map (λp → p.money) ps)

report : CashFlowStatement
report = let
    payments = [payment | payment : Payment ← transactions]
    mRevenues = [payment | payment ← payments, isMe (payment.receiver)]
    mExpenses = [payment | payment ← payments, isMe (payment.sender)]
  in
  CashFlowStatement{
    revenues = mRevenues,
    expenses = mExpenses,
    revenueTotal = sumPayments mRevenues,
    expenseTotal = sumPayments mExpenses}
```

UnpaidInvoices

name: *UnpaidInvoices*
description:
A list of unpaid invoices.
tags: *external*, *financial*

```
-- Generate a list of unpaid invoices
unpaidInvoices : [UnpaidInvoice]
unpaidInvoices = [UnpaidInvoice{invoice = inv, remainder = remainder} |
  invS ← invoicesSent,
  inv = Invoice{
    sender = invS.2.sender @,
    receiver = invS.2.receiver @,
    orderLines = invS.2.orderLines},
  payments = [payment.money |
    tr ← transactionEvents,
    tr.contractId ≡ invS.1,
    payment : Payment = tr.transaction],
  remainder = subtractMoney (invoiceTotal inv) payments,
  any (λm → m.amount > 0) remainder]

report : UnpaidInvoices
report = UnpaidInvoices{invoices = unpaidInvoices}
```

VATReport

name: *VATReport*
description:
The VAT report.
tags: *external*, *financial*

```
report : VATReport
report = VATReport{
  outgoingVAT = vatOutgoing,
```

```
  incomingVAT = vatIncoming,
  vatDue = subtractMoney vatIncoming vatOutgoing}
```

Inventory

name: *Inventory*
description:
A list of items in the inventory available for sale (regardless of whether we have paid for them).
tags: *external*, *inventory*

```
report : Inventory
report =
  let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
  in
  -- The available items is the list of received items minus the list of reserved
  -- or sold items
  Inventory{availableItems = normaliseItems (itemsReceived ++ itemsSold')}
```

TopNCustomers

name: *TopNCustomers*
description:
A list of customers who have spent must money in the given currency.
tags: *external*, *financial*, *crm*

```
customers : [⟨Customer⟩]
customers = [c | c : ⟨Customer⟩ ← map (λe → e.1) entities]

totalPayments : Currency → ⟨Customer⟩ → Real
totalPayments c cu = sum [d |
  p : Payment ← transactions,
  p.sender ≡ cu ∨ p.receiver ≡ cu,
  p.money.currency ≡ c,
  d = if p.sender ≡ cu then p.money.amount else 0 − p.money.amount]

customerStatistics : Currency → [CustomerStatistics]
customerStatistics c = [CustomerStatistics{customer = cu, totalPaid = p} |
  cu ← customers,
  p = Money{currency = c, amount = totalPayments c cu}]

topN : Int → [CustomerStatistics] → [CustomerStatistics]
topN n cs = take n (sortBy (λcs1 cs2 → cs1.totalPaid > cs2.totalPaid) cs)

report : Int → Currency → TopNCustomers
report n c = TopNCustomers{customerStatistics = topN n (customerStatistics c)}
```

C.3 Contracts

C.3.1 Prelude

```
// Arithmetic
fun floor x = let n = ceil x in if n > x then n − 1 else n
fun round x = let n1 = ceil x in let n2 = floor x in if n1 + n2 > 2 × x then n2 else n1
fun max a b = if a > b then a else b
fun min a b = if a > b then b else a

// List functions
fun filter f = foldr (λx b → if f x then x # b else b) []
fun map f = foldr (λx b → (f x) # b) []
val length = foldr (λx b → b + 1) 0
fun null l = l ≡ []
fun elem x = foldr (λy b → x ≡ y ∨ b) false
fun all f = foldr (λx b → b ∧ f x) true
```

```
fun any f = foldr (λx b → b ∨ f x) false
val reverse = foldl (λa e → e # a) []
fun append l1 l2 = foldr (λe a → e # a) l2 l1

// Lists as sets
fun subset l1 l2 = all (λx → elem x l2) l1
fun diff l1 l2 = filter (λx → ¬ (elem x l2)) l1
```

C.3.2 Domain-Specific Prelude

```
// Check if 'lines' are in stock by invoking the 'Inventory' report
fun inStock lines =
  let inv = (reports.inventory ()).availableItems
  in
  all (λl → any (λi → (l.item).itemType ≡ i.itemType ∧ (l.item).quantity ≤ i.quantity) inv) lines

// Check that amount 'm' equals the total amount in m's currency of a list of sales lines
fun checkAmount m orderLines =
  let a = foldr (λx acc →
            if (x.unitPrice).currency ≡ m.currency then
              (x.item).quantity × (100 + x.vatPercentage) × (x.unitPrice).amount + acc
            else
              acc) 0 orderLines
  in
  m.amount × 100 ≡ a

// Remove sales lines that have the currency of 'm'
fun remainingOrderLines m = filter (λx → (x.unitPrice).currency ≢ m.currency)

// A reference to the designated entity that represents the company
val me = reports.me ()
```

C.3.3 Contract Templates

Purchase

```
name: purchase
type: Purchase
description: "Set up a purchase"

clause purchase(lines : [OrderLine])⟨me : ⟨Me⟩, vendor : ⟨Vendor⟩⟩ =
 ⟨vendor⟩ Delivery(sender s, receiver r, items i)
   where s ≡ vendor ∧ r ≡ me ∧ i ≡ map (λx → x.item) lines
   due within 1 W
 then
 when IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ vendor ∧ r ≡ me ∧ sl ≡ lines
   due within 1 Y
 then
 payment(lines, vendor, 14D)⟨me⟩

clause payment(lines : [OrderLine], vendor : ⟨Vendor⟩, deadline : Duration)
                  ⟨me : ⟨Me⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨me⟩ BankTransfer(sender s, receiver r, money m)
     where s ≡ me ∧ r ≡ vendor ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

Sale

name: *sale*
type: Sale
description: `"Set up a sale"`

clause *sale*(*lines* : [OrderLine])⟨*me* : ⟨Me⟩, *customer* : ⟨Customer⟩⟩ =
⟨*me*⟩ IssueInvoice(*sender s*, *receiver r*, *orderLines sl*)
where $s \equiv me \wedge r \equiv customer \wedge sl \equiv lines \wedge inStock\ lines$
due within $1H$
then
payment(*lines*, *me*, $10m$)⟨*customer*⟩
and
⟨*me*⟩ Delivery(*sender s*, *receiver r*, *items i*)
where $s \equiv me \wedge r \equiv customer \wedge i \equiv map\ (\lambda x \rightarrow x.item)\ lines$
due within $1W$
then
repair(*map* $(\lambda x \rightarrow x.item)$ *lines*, *customer*, $3M$)⟨*me*⟩

clause *payment*(*lines* : [OrderLine], *me* : ⟨Me⟩, *deadline* : **Duration**)
⟨*customer* : ⟨Customer⟩⟩ =
if *null lines* **then**
fulfilment
else
⟨*customer*⟩ Payment(*sender s*, *receiver r*, *money m*)
where $s \equiv customer \wedge r \equiv me \wedge checkAmount\ m\ lines$
due within *deadline*
remaining *newDeadline*
then
payment(*remainingOrderLines m lines*, *me*, *newDeadline*)⟨*customer*⟩

clause *repair*(*items* : [Item], *customer* : ⟨Customer⟩, *deadline* : **Duration**)
⟨*me* : ⟨Me⟩⟩ =
when RequestRepair(*sender s*, *receiver r*, *items i*)
where $s \equiv customer \wedge r \equiv me \wedge subset\ i\ items$
due within *deadline*
remaining *newDeadline*
then
⟨*me*⟩ Repair(*sender s*, *receiver r*, *items i'*)
where $s \equiv me \wedge r \equiv customer \wedge i \equiv i'$
due within $5D$
and
repair(*items*, *customer*, *newDeadline*)⟨*me*⟩

contract = *sale*(*orderLines*)⟨*me*, *customer*⟩

Appendix B

Papers on the Partial-Order Approach to Infinitary Rewriting

Paper B1 P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49

Paper B2[1] P. Bahr. Partial Order Infinitary Term Rewriting and Bhm Trees. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.67

Paper B3[2] P. Bahr. Modes of Convergence for Term Graph Rewriting. *Logical Methods in Computer Science*, 8(2):6, 2012. doi: 10.2168/LMCS-8(2:6)2012

Paper B4 P. Bahr. Convergence in Infinitary Term Graph Rewriting Systems is Simple. Submitted to Math. Struct. in Comp. Science, 2012

Paper B5 P. Bahr. Infinitary Term Graph Rewriting is Simple, Sound and Complete. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69–84, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2012.69

[1] The paper that is included here is an extended and revised version that is currently under peer review for publication in Logical Methods in Computer Science.

[2] This paper is an extended and revised version of [11].

Abstract Models of Transfinite Reductions

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

We investigate transfinite reductions in abstract reduction systems. To this end, we study two abstract models for transfinite reductions: a metric model generalising the usual metric approach to infinitary term rewriting and a novel partial order model. For both models we distinguish between a weak and a strong variant of convergence as known from infinitary term rewriting. Furthermore, we introduce an axiomatic model of reductions that is general enough to cover all of these models of transfinite reductions as well as the ordinary model of finite reductions. It is shown that, in this unifying axiomatic model, many basic relations between termination and confluence properties known from finite reductions still hold. The introduced models are applied to term rewriting but also to term graph rewriting. We can show that for both term rewriting as well as for term graph rewriting the partial order model forms a conservative extension to the metric model.

Contents

1 Introduction

The study of infinitary term rewriting, introduced by Dershowitz et al. [7], is concerned with reductions of possibly infinite length. To formalise the concept of transfinite reductions, a variety of different models were investigated in the last 20 years. The most thoroughly studied model is the metric model, both its weak [7] and its strong [13] variant. Other models, using for example general topological spaces [18] or partial orders [5, 6], were mostly considered to pursue specific purposes. Within these models many fundamental properties do not depend on the particular structure of terms, e.g. the property that strongly converging reductions in the metric model have countable length. Moreover, when studying these different approaches to transfinite reductions, one realises that they often share many basic properties, e.g. in how reductions can be composed and decomposed.

The purpose of this paper is to study transfinite reductions on an abstract level using several different models. This includes a metric model (Section 5) as well as a novel partial order model (Section 6), each of which induces a weak and a strong variant of convergence. Moreover, we introduce an *axiomatic model* of *transfinite abstract reduction systems* (Section 4) which captures the fundamental properties of transfinite reductions. This model subsumes both variants of the metric and the partial order model, respectively, as well as ordinary finite reductions. In fact, we formulate these more concrete models in terms of the axiomatic model, which simplifies their presentation and their analysis substantially. To illustrate this, we reformulate well-known termination and confluence properties in the unifying axiomatic model and show that this then yields the corresponding standard termination and confluence properties for standard finite term rewriting resp. infinitary term rewriting. Additionally, we also prove that basic relations between these properties known from the finite setting also hold in this more general setting.

Lastly, we briefly mention that our models can be applied to term graph rewriting [4] (Section 7) which yields the first formalisation of infinitary term graph rewriting. Moreover, we show that the partial order model is in fact superior to the metric model, at least for interesting cases like terms and term graphs: It can model convergence as in the metric model but additionally allows to distinguish between different levels of divergence.

1.1 Related Work

The idea of investigating transfinite reductions on an abstract level was first pursued by Kennaway [11] by studying strongly convergent reductions in an abstract metric framework similar to ours. In this paper we will show that almost all of Kennaway's positive results (except countability of strong convergence) already hold in our more general axiomatic framework, and that countability already holds for strongly continuous reductions.

Kahrs [9] investigated a more concrete model in which he considered weakly convergent reductions in term rewriting systems parametrised by the metric on

terms. Although being parametric in the metric space, the results of Kahrs are tied to term rewriting and are for example not applicable to term graph rewriting [2].

The use of partial orders and their notion of *limit inferior* for transfinite reductions is inspired by Blom [5] who studied strongly convergent reductions in lambda calculus using a partial order and compared this to the ordinary metric model of strongly convergent reductions.

2 Preliminaries

We assume familiarity with the basic theory of ordinal numbers, orders and topological spaces [10], as well as term rewriting [20]. In the following, we briefly recall the most important notions.

2.1 Transfinite Sequences

We use $\alpha, \beta, \gamma, \lambda, \iota$ to denote ordinal numbers. A *transfinite sequence* (or simply called *sequence*) S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*. For a finite sequence $(a_\iota)_{\iota<n}$, we also write $\langle a_0, a_1, \ldots, a_{n-1}\rangle$.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a (proper) *prefix* of a sequence T, denoted $S \leq T$ (resp. $S < T$), if there is a (non-empty) sequence S' with $S \cdot S' = T$. The prefix of T of length β is denoted $T|_\beta$. The relation $\leq$ forms a complete semilattice.

2.2 Metric Spaces

A pair $(M, \mathbf{d})$ is called a *metric space* if $\mathbf{d}\colon M \times M \to \mathbb{R}_0^+$ is a function satisfying $\mathbf{d}(x, y) = 0$ iff $x = y$ (identity), $\mathbf{d}(x, y) = \mathbf{d}(y, x)$ (symmetry), and $\mathbf{d}(x, z) \leq \mathbf{d}(x, y) + \mathbf{d}(y, z)$ (triangle inequality), for all $x, y, z \in M$. If $\mathbf{d}$ instead of the triangle inequality, satisfies the stronger property $\mathbf{d}(x, z) \leq \max\{\mathbf{d}(x, y), \mathbf{d}(y, z)\}$ (strong triangle), $(M, \mathbf{d})$ is called an *ultrametric space*. If a sequence $(a_\iota)_{\iota<\alpha}$ in a metric space converges to an element a, we write $\lim_{\iota\to\alpha} a_\iota$ to denote a. A sequence $(a_\iota)_{\iota<\alpha}$ in a metric space is called *Cauchy* if, for any $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that, for all $\beta < \iota < \iota' < \alpha$, we have that $\mathbf{d}(m_\iota, m_{\iota'}) < \varepsilon$. A metric space is called *complete* if each of its non-empty Cauchy sequences converges.

2.3 Partial Orders

A *partial order* $\leq$ on a class A is a binary relation on A that is *transitive*, *reflexive*, and *antisymmetric*. A partial order $\leq$ on A is called a *complete semilattice* if it has a *least element*, every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$ in A, and every subset of A having an upper bound in A also has a least upper bound in A. Hence, complete semilattices also admit a *greatest lower bound* (*glb*) $\bigsqcap B$ for every *non-empty* subset B of A. In particular, this means that for any

non-empty sequence $(a_\iota)_{\iota<\alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha}\left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists. A partial order is called a *linear order* if $a \leq b$ or $b \leq a$ holds for each pair of elements a, b. A linearly ordered subclass of a partially ordered class is also called a *chain*.

2.4 Term Rewriting Systems

Instead of finite terms, we consider the set $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ of *infinitary terms* over some *signature* Σ and a countably infinite set $\mathcal{V}$ of variables. We consider $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ as a superset of the set $\mathcal{T}(\Sigma,\mathcal{V})$ of *finite terms*. For a term $t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. For terms $s, t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s.

On $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ a distance function $\mathbf{d}$ can be defined by $\mathbf{d}(s,t) = 0$ if $s = t$ and $\mathbf{d}(s,t) = 2^{-k}$ if $s \neq t$, where k is the minimal depth at witch s and t differ. The pair $(\mathcal{T}^\infty(\Sigma,\mathcal{V}),\mathbf{d})$ is known to form a complete ultrametric space [1]. *Partial terms*, i.e. terms over signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$, can be endowed with a relation $\leq_\bot$ by defining $s \leq_\bot t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\bot$. The pair $(\mathcal{T}^\infty(\Sigma_\bot,\mathcal{V}),\leq_\bot)$ is known to form a complete semilattice [8].

A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}(\Sigma,\mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ such that all variables in r are contained in l. Note that this notion of a TRS is standard in infinitary rewriting [12], but deviates from standard TRSs as it allows infinitary terms on the right-hand side of rules.

As in the finitary case, every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \sigma\colon\ s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

We write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π.

3 Abstract Reduction Systems

In order to analyse transfinite reductions on an abstract level, we consider *abstract reduction systems (ARS)*. In ARSs, the principal items of interest are the reduction steps of the system. Therefore, the structure of the individual objects on which the reductions are performed is neglected. This abstraction is usually modelled by a pair (A, R) consisting of a set A of objects and a binary relation R on A describing the possible reductions on the objects. The ARS induced by a TRS $\mathcal{R}$ is then simply the pair $(\mathcal{T}^\infty(\Sigma,\mathcal{V}), R)$ with $(s,t) \in R$ iff $s \to_\mathcal{R} t$.

In the setting of infinitary rewriting, however, this model is not appropriate. Instead, we need a model which *reifies* the reduction steps of the system since the semantics of transfinite reductions does not only depend on the objects involved in the reduction but also on *how* each reduction step is performed – at least when we consider *strong convergence*. However, it is not always possible to reconstruct how a reduction was performed given only the starting and end object of it due to so-called *syntactic accidents* [17]: Consider the term rewrite rule $\rho\colon\ f(x) \to x$

and the term $f(f(x))$. The rule ρ can be applied both at root position $\langle\rangle$ and at position $\langle 0\rangle$ of $f(f(x))$. In both cases the resulting term is $f(x)$.

Therefore, we rather choose a model in which reduction steps are "first-class citizens" [20] similarly to morphisms in a category:

Definition 3.1 (abstract reduction system)**.** An *abstract reduction system (ARS)* $\mathcal{A}$ is a quadruple $(A, \Phi, \mathsf{src}, \mathsf{tgt})$ consisting of a set of *objects* A, a set of *reduction steps* Φ, and *source* and *target functions* $\mathsf{src}\colon \Phi \to A$ and $\mathsf{tgt}\colon \Phi \to A$, respectively. We write $\varphi\colon a \to_{\mathcal{A}} b$ whenever there are $\varphi \in \Phi$, $a, b \in A$ such that $\mathsf{src}(\varphi) = a$ and $\mathsf{tgt}(\varphi) = b$.

In order to define the semantics of a TRS in terms of an ARS we only need to define an appropriate notion of a reduction step:

Definition 3.2 (operational semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The ARS *induced* by $\mathcal{R}$, denoted $\mathcal{A}_{\mathcal{R}}$, is given by $(\mathcal{T}^{\infty}(\Sigma, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$, where $\Phi = \{(s, \pi, \rho, t) \mid s \to_{\pi,\rho} t\}$, $\mathsf{src}(\varphi) = s$ and $\mathsf{tgt}(\varphi) = t$, for each $\varphi = (s, \pi, \rho, t) \in \Phi$.

A *reduction* in this setting is simply a sequence of reduction steps in an ARS such that consecutive steps are "compatible":

Definition 3.3 (reduction)**.** A sequence $S = (\varphi_\iota)_{\iota<\alpha}$ of reduction steps in an ARS $\mathcal{A}$ is called a *reduction* if there is a sequence of objects $(a_\iota)_{\iota<\widehat{\alpha}}$ in the underlying set A, where $\widehat{\alpha} = \alpha$ if S is open, and $\widehat{\alpha} = \alpha+1$ if S is closed, such that $\varphi_\iota\colon a_\iota \to a_{\iota+1}$ for all $\iota < \alpha$. For such a sequence, we also write $(\varphi\colon a_\iota \to a_{\iota+1})_{\iota<\alpha}$ or simply $(a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The reduction S is said to start in a_0, and if S is closed, it is said to end in a_α. If S is finite, we write $S\colon a_0 \to^*_{\mathcal{A}} a_\alpha$. We use the notation $\mathsf{Red}(\mathcal{A})$ to refer to the class of all *non-empty reductions* in $\mathcal{A}$.

Observe that the empty sequence $\langle\rangle$ is always a reduction, and that $\langle\rangle$ starts and ends in a for every object a of the ARS. Also note that this notion of reductions alone does only make sense for sequences of length at most ω. For longer reductions, the ω-th step is not related to the preceding steps of the reduction:

Example 3.4. In the TRS consisting of the rules $a \to f(a)$ and $b \to g(b)$ the following constitutes a valid reduction of length $\omega \cdot 2$:

$$S\colon\ a \to f(a) \to f(f(a)) \to f(f(f(a))) \to\ \ldots\ b \to g(b) \to g(g(b)) \to\ \ldots$$

The second half of the reduction is completely unrelated to the first half. The reason for this issue is that the ω-th reduction step $b \to g(b)$ has no immediate predecessor.

The above problem can occur for all reduction steps indexed by a limit ordinal. For successor ordinals, this is not a problem as by Definition 3.3 the $(\iota+1)$-st step is required to start in the object that the ι-th step ends in. Meaningful definitions for reductions of length beyond ω have to include an appropriate notion of *continuity* which bridges the gaps caused by limit ordinals. Exploring different variants of such a notion of continuity is the topic of the subsequent sections.

4 Transfinite Abstract Reduction Systems

In the last section we have seen that we need a notion of continuity in order to obtain a meaningful model of transfinite reductions. In this section we introduce an axiomatic framework for convergence in which we can derive a corresponding notion of continuity.

The resulting notion of continuity is quite natural and resembles the definition of continuity of real-valued functions: A reduction is *continuous* if every proper prefix *converges* to the object the subsequent suffix is starting in. In order to use this idea, we need to endow an ARS with a notion of convergence:

Definition 4.1 (transfinite abstract reduction system)**.** A *transfinite abstract reduction system (TARS)* $\mathcal{T}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$, such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{T}$, and

(ii) $\mathsf{conv}\colon \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ is a partial function, called *notion of convergence*, which satisfies the following two axioms:

$$\mathsf{conv}(\langle\varphi\rangle) = \mathsf{tgt}(\varphi) \quad \text{for all } \varphi \in \Phi \tag{STEP}$$

$$\mathsf{conv}(S) = a \text{ and } \mathsf{conv}(T) = b \iff \mathsf{conv}(S \cdot T) = b \quad \text{for all } a, b \in A,\ S, T \in \mathsf{Red}(\mathcal{A}) \text{ with } T \text{ starting in } a. \tag{CONCATENATION}$$

That is, we require convergence to include single reduction steps and to be preserved by both composition and decomposition.

Axiom (CONCATENATION) is, in fact, quite comprehensive. But we can split it up into two axioms whose conjunction is equivalent to it:

$$\mathsf{conv}(S) = a \implies \mathsf{conv}(S \cdot T) = \mathsf{conv}(T) \tag{COMPOSITION}$$

$$\mathsf{conv}(S \cdot T) \text{ defined} \implies \mathsf{conv}(S) = a \tag{CONTINUITY}$$

where S and T range over reductions in $\mathsf{Red}(\mathcal{A})$ with T starting in $a \in A$.

Axiom (COMPOSITION) states that the composition of reductions preserves the convergence behaviour whereas (CONTINUITY) ensures that every notion of convergence already includes continuity. To see the latter we need to define convergence and continuity in TARSs:

Definition 4.2 (convergence, continuity)**.** Let $\mathcal{T} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$ be a TARS and $S \in \mathsf{Red}(\mathcal{T})$ a non-empty reduction starting in $a \in A$. S is said to *converge* to $b \in A$, written $S\colon\ a \twoheadrightarrow_{\mathcal{T}} b$, if $\mathsf{conv}(S) = b$. S is said to be *continuous*, written $S\colon\ a \twoheadrightarrow_{\mathcal{T}} \ldots$, if for every two $S_1, S_2 \in \mathsf{Red}(\mathcal{T})$ with $S = S_1 \cdot S_2$, we have that S_1 converges to the object S_2 is starting in. If S is continuous but not converging, then S is called *divergent*. For the empty reduction $\langle\rangle$, we define to have $\langle\rangle\colon\ a \twoheadrightarrow_{\mathcal{T}} a$ and $\langle\rangle\colon\ a \twoheadrightarrow_{\mathcal{T}} \ldots$ for all $a \in A$, i.e. $\langle\rangle$ is always convergent and continuous. To indicate the length α of a reduction we use the notation $\twoheadrightarrow_{\mathcal{T}}^{\alpha}$. For some object $a \in A$, we write $\mathsf{Cont}(\mathcal{T}, a)$ and $\mathsf{Conv}(\mathcal{T}, a)$ to denote the class of all continuous resp. convergent reductions in $\mathcal{T}$ starting in a.

Axiom (CONTINUITY) is equivalent to the statement that every converging reduction is also continuous. That is, only meaningful – i.e. continuous – reductions can be convergent. This is a natural model which is in particular also adopted in the theory of infinitary term rewriting [12].

Returning to Example 3.4, we can see that for S to be continuous the prefix $S|_\omega$ has to converge to b. However, as one might expect, all notions of convergence for TRSs we will introduce in this paper agree on that $S|_\omega$ converges to f^ω.

Since for closed reductions not only does convergence imply continuity, but also the converse holds true, we have the following proposition:

Proposition 4.3 (convergence of closed reductions)**.** *Let $\mathcal{T}$ be a TARS and S a closed reduction in $\mathcal{T}$. Then S is continuous iff S is converging.*

Proof. The "if" direction follows from (CONTINUITY). The "only if" direction is trivial if S is empty and follows from (STEP) if S has length one. Otherwise, S is of the form $T \cdot \varphi$. Since φ is converging by (STEP) and T is converging by (CONTINUITY), S is converging due to (COMPOSITION). □

It is obvious from the definition that a well-defined notion of convergence has to include at least all finite (non-empty) reductions. In fact, the trivial notion of convergence which consists of precisely the *finite reductions* is the least notion of convergence w.r.t. set inclusion of its domain:

Definition 4.4 (finite convergence)**.** Let $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ be an ARS. Then the *finite convergence* of $\mathcal{A}$ is the TARS $\mathcal{A}^f = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$, where conv is defined by $\mathsf{conv}(S) = b$ iff $S\colon a \to^*_{\mathcal{A}} b$. That is, $\mathsf{conv}(S)$ is undefined iff S is infinite.

The TARS given above can be easily checked to be well-defined, i.e. conv satisfies the axioms given in Definition 4.1. We then obtain for every reduction S that $S\colon a \to^*_{\mathcal{A}} b$ iff $S\colon a \twoheadrightarrow_{\mathcal{A}^f} b$. This shows that TARSs merely provide a generalisation of what is considered to be a well-formed reduction.

Defining conv for the finite convergence was simple. In general, however, it is quite cumbersome to define, as a notion of convergence has to already comprise the corresponding notion of continuity, i.e. satisfy (CONTINUITY). We can avoid this by defining for each partial function $\mathsf{conv}\colon \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ its *continuous core* $\overline{\mathsf{conv}}\colon \mathsf{Red}(\mathcal{A}) \rightharpoonup A$. For each non-empty reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$ in $\mathcal{A}$ we define

$$\overline{\mathsf{conv}}(S) = \begin{cases} \mathsf{conv}(S) & \text{if } \forall 0 < \beta < \alpha \quad \mathsf{conv}(S|_\beta) = a_\beta \\ \text{undefined} & \text{otherwise} \end{cases}$$

We then have the following lemma:

Lemma 4.5 (continuous core)**.** *Let $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ be an ARS and $\mathsf{conv}\colon \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ a partial function satisfying* (STEP) *and* (COMPOSITION)*. Then $\overline{\mathsf{conv}}$ satisfies* (STEP) *and* (CONCATENATION)*, i.e. $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}})$ is a TARS.*

Proof. Straightforward. □

Next we have a look at transfinite versions of well-known termination and confluence properties. The basic idea for lifting these properties to the setting of transfinite reductions is to replace finite reductions, i.e. $\to^*$, with transfinite reductions, i.e. $\twoheadrightarrow$.

Applied to the properties *confluence* (CR), *normalisation* (WN), and the *unique normal form property w.r.t. reduction* ($\mathsf{UN}_\to$) we obtain the following transfinite properties:

- CR^∞: If $b \twoheadleftarrow a \twoheadrightarrow c$, then $b \twoheadrightarrow d \twoheadleftarrow c$.
- WN^∞: For each a, there is a normal form b with $a \twoheadrightarrow b$.
- $\mathsf{UN}^\infty_\to$: If $b \twoheadleftarrow a \twoheadrightarrow c$ and b, c are normal forms, then $b = c$.

For properties involving convertibility, i.e. $\leftrightarrow^*$, one has to be more careful. The seemingly straightforward formalisation using transfinite reductions in the *symmetric closure* of the underlying ARS does not work since we do not have a notion of convergence for the symmetric closure. Even if we had one, as in the more concrete models that use a metric space or a partial order, the resulting transfinite convertibility relation would not be symmetric [2].

We therefore follow the approach of Kennaway [11]:

Definition 4.6 (transfinite convertibility)**.** Let $\mathcal{T}$ be a TARS, and a, b objects in $\mathcal{T}$. The objects a and b are called *transfinitely convertible*, written $a \twoheadleftarrow\!\!\twoheadrightarrow_\mathcal{T} b$, whenever there is a finite sequence of objects $a_0, \ldots, a_n$, $n \geq 0$, in $\mathcal{T}$ such that $a_0 = a$, $a_n = b$, and, for each $0 \leq i < n$, we have $a_i \twoheadrightarrow_\mathcal{T} a_{i+1}$ or $a_i \twoheadleftarrow_\mathcal{T} a_{i+1}$. The minimal n of such a sequence is called the length of $a \twoheadleftarrow\!\!\twoheadrightarrow_\mathcal{T} b$.

This definition of transfinite convertibility is in some sense not "fully transfinite": For two objects to be transfinitely convertible, there has to be a transfinite "reduction" which may only finitely often changes its direction. However, with this definition, transfinite convertibility is an equivalence relation as desired, and we can establish an alternative characterisation of CR^∞ analogously to the original finite version:

Proposition 4.7 (alternative characterisation of CR^∞)**.** *Let $\mathcal{T}$ be a TARS.*

$$\mathcal{T} \textit{ is } \mathsf{CR}^\infty \qquad \Longleftrightarrow \qquad \textit{Whenever } a \twoheadleftarrow\!\!\twoheadrightarrow b, \textit{ then } a \twoheadrightarrow c \twoheadleftarrow b.$$

Proof. The argument is the same as for finite reductions: The "if" direction is trivial, and the "only if" direction can be proved by an induction on the length of $a \twoheadleftarrow\!\!\twoheadrightarrow b$. □

With the definition of transfinite convertibility in place, we can define the transfinite versions of the *normal form property* (NF) and the *unique normal form property* (UN):

- NF^∞: For each object a and normal form b with $a \twoheadleftarrow\!\!\twoheadrightarrow b$, we have $a \twoheadrightarrow b$.
- UN^∞: All normal forms a, b with $a \twoheadleftarrow\!\!\twoheadrightarrow b$ are identical.

The above definition of NF^∞ differs from that of Kennaway et al. [13] who, instead of $a \nleftrightarrow b$, use $a \nleftarrow c \nrightarrow b$ as the precondition. One can, however, easily show that both definitions are equivalent.

Having these transfinite properties, we can establish some relations between them analogously to the setting of finite reductions:

Proposition 4.8 (confluence properties)**.** *For every TARS, the following implications hold:*

(i) $\mathsf{CR}^\infty \implies \mathsf{NF}^\infty \implies \mathsf{UN}^\infty \implies \mathsf{UN}^\infty_\to$

(ii) $\mathsf{WN}^\infty \;\&\; \mathsf{UN}^\infty_\to \implies \mathsf{CR}^\infty$

Proof. The arguments are the same as for their finite variants. □

Also when formulating a transfinite version of the termination property, we have to be careful. In fact, several different formalisations of transfinite termination can be found in the literature [11, 16, 18].

We suggest a notion of transfinite termination which we belief is a direct generalisation of finite termination. Recall that an object a in an ARS is terminating iff there is no infinite reduction starting in a. From this we can see that for finite reductions, we can make use of infinite reductions as a meta-concept for defining finite termination. A corresponding meta-concept for transfinite reductions is provided by the class $\mathsf{Conv}(\mathcal{T}, a)$ of converging reductions starting in a ordered by the prefix order $\leq$. The analogue of an infinite reduction, which witnesses finite non-termination, is an unbounded chain in $\mathsf{Conv}(\mathcal{T}, a)$, which witnesses transfinite non-termination:

Definition 4.9 (transfinite termination)**.** Let $\mathcal{T}$ be a TARS. An object a in $\mathcal{T}$ is said to be *transfinitely terminating* (SN^∞) if each chain in $\mathsf{Conv}(\mathcal{T}, a)$ has an upper bound in $\mathsf{Conv}(\mathcal{T}, a)$. The TARS $\mathcal{T}$ itself is called *transfinitely terminating* (SN^∞) if every object in $\mathcal{T}$ is.

The following alternative characterisation of SN^∞ will be useful for comparing our definition to other formalisations of SN^∞ in the literature:

Proposition 4.10 (transfinite termination)**.** *An object a in a TARS $\mathcal{T}$ is SN^∞ iff*

(a) $\mathsf{Cont}(\mathcal{T}, a) \subseteq \mathsf{Conv}(\mathcal{T}, a)$*, and*

(b) every chain in $\mathsf{Conv}(\mathcal{T}, a)$ *is a set.*

Proof. Note that (b) is equivalent to the statement that, for every chain C in $\mathsf{Conv}(\mathcal{T}, a)$, there is an upper bound on the length of the reductions in C.

We show the "only if" direction by proving its contraposition: If (a) is violated, then there is a divergent reduction $S\colon\ a \twoheadrightarrow \ldots$. Hence, the set of all proper prefixes of S forms a chain in $\mathsf{Conv}(\mathcal{T}, a)$ which has no upper bound. Consequently, a is not SN^∞. If (b) is violated, transfinite non-termination of a follows immediately.

For the "if" direction, consider an arbitrary chain C in $\mathsf{Conv}(\mathcal{T}, a)$. Because of (b), C has a lub S. For each proper prefix $S' < S$, there has to be an extension

$S'' \geq S$ in C. Since S'' is converging, so is S'. Consequently, S is continuous and, therefore, also convergent, due to (a). Hence, S is an upper bound for C in $\mathsf{Conv}(\mathcal{T}, a)$. □

The above characterisation shows that there are two different reasons for transfinite non-termination: Diverging reductions and reductions that can be extended indefinitely. This characterisation of termination closely resembles that of Rodenburg [18] which, however, additionally to (a) and instead of (b) requires an upper bound on the length of reductions. This is too restrictive, since an object, in which for each ordinal α a reduction of length α to a normal form starts, is not transfinitely terminating according to Rodenburg's definition.[1] An example witnessing this difference to our definition can be devised straightforwardly.

In order to verify that our formalisation of SN^∞ is appropriate, we have to make sure that it implies WN^∞:

Proposition 4.11 (SN^∞ is stronger than WN^∞)**.** *For every TARS $\mathcal{T}$, it holds that SN^∞ implies WN^∞ for every object in $\mathcal{T}$.*

Proof. We prove the contraposition of the implication using Proposition 4.10. For this purpose, let $\mathcal{T}$ be an TARS and a some object in $\mathcal{T}$ that is not WN^∞. We show that then (a) or (b) of Proposition 4.10 is violated. For this purpose, we assume (a) and show that then (b) does not hold. To this end we define a function f on the class On of ordinal numbers such that, for each $\alpha \in \mathsf{On}$, (1) $f(\alpha)$ is a converging reduction of length α starting in a and (2) $f(\alpha)$ is a proper extension of $f(\iota)$ for all $\iota < \alpha$, i.e. $f(\alpha) > f(\iota)$. Hence, the class $\{f(\alpha) \mid \alpha \in \mathsf{On}\}$ is a chain in $\mathsf{Conv}(\mathcal{T}, a)$ which is not a set since f is a bijection from the proper class On to $\{f(\alpha) \mid \alpha \in \mathsf{On}\}$. The construction of f is justified by the principle of transfinite recursion, and the properties (1) and (2) are established by transfinite induction.

For $\alpha = 0$, both (1) and (2) are trivial. Let α be a successor ordinal $\beta + 1$. By induction hypothesis, we have $f(\beta)\colon\ a \twoheadrightarrow^\beta b$ for some b. Since a is not WN^∞, b cannot be a normal form. Hence, there is a step $\varphi\colon\ b \to b'$ in $\mathcal{M}$. Define $f(\alpha) = f(\beta) \cdot \langle\varphi\rangle$. That is, $f(\alpha)\colon\ a \twoheadrightarrow^\alpha b'$ which shows (1). (2) follows from the induction hypothesis since $f(\beta) < f(\alpha)$.

Let α be a limit ordinal. Since, by the induction hypothesis, (2) holds for all $f(\beta)$, we have that $F = \{f(\beta) \mid \beta < \alpha\}$ is a directed set. Hence, $f(\alpha) = \bigsqcup F$ is well-defined. Consequently, all elements in F are proper prefixes of $f(\alpha)$. This shows (2) and, additionally, it shows that $f(\alpha)$ is a reduction of length α starting in a. Since, by the induction hypothesis $f(\beta)$ is converging for each $\beta < \alpha$, we have that $f(\alpha)$ is continuous. Due to (a), $f(\alpha)$ is also convergent, which shows (1). □

Note that the transfinite properties we have introduced are equivalent to their finite counterpart if we consider the finite convergence of an ARS. This shows that the transfinite properties that we have given here are in fact generalisations of their original finite versions to the setting of TARS. Moreover, all counterexamples known from the finite setting carry over to the setting of transfinite reductions.

[1] In fact, in an earlier draft of this paper we adopted Rodenburg's definition. We thank the anonymous referee who pointed out the mentioned issue.

This means, for example, that the implications shown in Proposition 4.11 and Proposition 4.8 are in fact strict as they are in the setting of finite reductions.

There are also many interrelations between finite properties which do not hold in the transfinite setting. Notable examples are Newman's Lemma and the implication from subcommutativity to confluence. Counterexamples for these and other interrelations are given by Kennaway [11].

5 Metric Model of Transfinite Reductions

The most common model of infinitary term rewriting is based on the complete ultrametric space of $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. One usually distinguishes between two different variants in this context: A weak variant [7], which only takes into account the metric space, and a strong variant [13], which stipulates additional restrictions on the applications of rewrite rules in order to obtain a more well-behaved notion of convergence.

At first we introduce the abstract theory of metric reduction systems. Afterwards, we describe how this can be applied to term rewriting.

Definition 5.1 (metric reduction system)**.** A *metric reduction system (MRS)* $\mathcal{M}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$, such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{M}$,

(ii) $\mathbf{d}\colon\ A \times A \to \mathbb{R}_0^+$ is a function such that $(A, \mathbf{d})$ is a metric space,

(iii) $\mathsf{hgt}\colon\ \Phi \to \mathbb{R}^+$ is a function, called the *height function*, and

(iv) if $\varphi\colon\ a \to_{\mathcal{A}} b$, then $\mathbf{d}(a, b) \leq \mathsf{hgt}(\varphi)$.

If the metric of an MRS $\mathcal{M}$ is an ultrametric, then $\mathcal{M}$ is called an *ultrametric reduction system (URS)*. Furthermore, an MRS is referred to as *complete* if the underlying metric space is complete. We use the notation $\varphi\colon\ a \to_h b$ to indicate that $\mathsf{hgt}(\varphi) = h$.

The definition of metric reduction systems follows the idea of *metric abstract reduction systems* investigated by Kennaway [11]. The essential difference between our approach and that of Kennaway is the use of abstract reduction systems with reified reduction steps instead of a family of binary relations. Moreover, unlike Kennaway, we do not restrict ourselves to complete ultrametric spaces. This will allow us to distinguish in which circumstances completeness or an ultrametric is necessary and in which not.

Before continuing the discussion of the abstract model, let us have a look at how TRSs fit into it:

Definition 5.2 (MRS semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The MRS *induced* by $\mathcal{R}$, denoted $\mathcal{M}_\mathcal{R}$, is given by $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$, where $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$ is the ARS $\mathcal{A}_\mathcal{R}$ induced by $\mathcal{R}$, $\mathbf{d}$ is the metric on $\mathcal{T}^\infty(\Sigma, \mathcal{V})$, and hgt is defined as

$$\mathsf{hgt}(\varphi) = 2^{-|\pi|}, \text{ where } \varphi\colon\ t \to_{\pi,\rho} t'.$$

One can easily check that $\mathcal{M}_\mathcal{R}$ indeed forms an MRS for each TRS $\mathcal{R}$. In fact, since the metric on $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ is a *complete ultrametric* [1], $\mathcal{M}_\mathcal{R}$ is a *complete URS*.

Next we define for each MRS two notions of convergence:

Definition 5.3 (convergence in MRSs)**.** Let $\mathcal{M} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ be an MRS. The *weak convergence* of $\mathcal{M}$, denoted $\mathcal{M}^w$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^w)$, where $\mathsf{conv}^w(S) = \lim_{\iota\to\widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The *strong convergence* of $\mathcal{M}$, denoted $\mathcal{M}^s$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^s)$, where $\mathsf{conv}^s(S) = \lim_{\iota\to\widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ if S is closed or $\lim_{\iota\to\alpha} h_\iota = 0$; otherwise it is undefined.

The notions of convergence defined above yield precisely the weakly converging [7] resp. the strongly converging [13] reductions typically considered in the literature on infinitary term rewriting [12].

From the definition we can immediately derive that strong convergence implies weak convergence. Hence, also strong continuity implies weak continuity.

Note that the height function hgt provides an overapproximation $\mathsf{hgt}(\varphi)$ of the real distance $\mathbf{d}(a,b)$ between the objects a, b involved in a reduction step $\varphi\colon\ a \to b$. Intuitively, speaking, the difference between weak and strong convergence is that, in the latter variant, the underlying sequence of objects $(a_\iota)_{\iota<\widehat{\alpha}}$ has to converge for the overapproximation provided by hgt as well. In fact, if it is a precise approximation, then weak and strong convergence coincide:

Fact 5.4 (equivalence of weak and strong convergence)**.** *Let $\mathcal{M}$ be an MRS $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ with $\mathsf{hgt}(\varphi) = \mathbf{d}(a,b)$ for every reduction step $\varphi\colon\ a \to b \in \Phi$. Then for each reduction S in $\mathcal{M}$ we have*

$$(i)\ S\colon\ a \twoheadrightarrow_{\mathcal{M}^w} \ldots \text{ iff } S\colon\ a \twoheadrightarrow_{\mathcal{M}^s} \ldots, \quad \text{and} \quad (ii)\ S\colon\ a \twoheadrightarrow_{\mathcal{M}^w} b \text{ iff } S\colon\ a \twoheadrightarrow_{\mathcal{M}^s} b.$$

Proof. We only need to show that conv^s and conv^w coincide for $\mathcal{M}$. For closed reductions this is trivial. Let $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ be an open reduction. If $\mathsf{conv}^w(S)$ is undefined, then so is $\mathsf{conv}^s(S)$. If $\mathsf{conv}^w(S)$ is defined, then the sequence $(a_\iota)_{\iota<\alpha}$ converges and is therefore Cauchy. Consequently, the sequence $(\mathbf{d}(a_\iota, a_{\iota+1}))_{\iota<\alpha}$ tends to 0 which implies that also $(h_\iota)_{\iota<\alpha}$ tends to 0 as $h_\iota = \mathbf{d}(a_\iota, a_{\iota+1})$ for each $\iota < \alpha$. Thus, $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$. □

It is instructive to see how hgt provides an overapproximation of the distance function for the example of terms: It assumes that the metric distance between redex and contractum is maximal. That is, the height function only provides a precise approximation if every redex has a root symbol different from the one of its contractum as it is the case for the rule $\rho_1\colon\ c \to g(c)$: The reduction $f(c) \to_{\rho_1} f(g(c)) \to_{\rho_1} f(g(g(c))) \to_{\rho_1} \ldots$ converges both weakly and strongly to $f(g^\omega)$. For the rule $\rho_2\colon\ f(x) \to f(g(x))$ this is not the case; both redex and contractum have the same root symbol f. The reduction $f(c) \to_{\rho_2} f(g(c)) \to_{\rho_2} f(g(g(c))) \to_{\rho_2} \ldots$ now converges weakly to $f(g^\omega)$ but is not strongly converging.

Note that this also shows the need for reifying reduction steps since in a system containing both ρ_1 and ρ_2 a reduction of the shape $f(c) \to f(g(c)) \to f(g(g(c))) \to \ldots$ can be strongly convergent or not, depending on which rules are applied. Similarly, with only a single rule $\rho_3\colon\ g(x) \to g(g(x))$ a reduction of

the shape $g(c) \to g(g(c)) \to g(g(g(c))) \to \dots$ can be strongly converging or not, depending on *where* ρ_3 is applied.

The reason for considering strong convergence is that it is considerably more well-behaved [13] than weak convergence [19]. However, weak convergence in the systems characterised in Fact 5.4 inherit the nice properties of strong convergence. For TRSs these systems are precisely those for which the root-symbol of each right-hand side is a function symbol different from the root symbol of the corresponding left-hand side.

When dealing with complete URSs, strong convergence can be characterised by the height only:

Proposition 5.5 (strong convergence in complete URSs)**.** *Let $\mathcal{M}$ be a complete URS. Every open strongly continuous reduction $(a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ in $\mathcal{M}$ is strongly convergent iff $(h_\iota)_{\iota<\alpha}$ tends to 0.*

Proof. The "only if" direction is immediate from the definition of strong convergence. For the "if" direction, assume a strongly continuous reduction $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ with $\lim_{\iota\to\alpha} h_\iota = 0$. Then $\lim_{\iota\to\alpha} \mathbf{d}(a_\iota, a_{\iota+1}) = 0$ which in turn implies that $(a_\iota)_{\iota<\alpha}$ is Cauchy as $\mathbf{d}$ is an ultrametric. Since we have a complete metric space, this means that $(a_\iota)_{\iota<\alpha}$ converges. From this and $\lim_{\iota\to\alpha} h_\iota = 0$ we can conclude that S is strongly converging. □

Having a complete URS is crucial for the "if" direction of Proposition 5.5. If $\mathcal{M}$ it is not a URS, the underlying sequence $(a_\iota)_{\iota<\alpha}$ might not be Cauchy:

Example 5.6. Consider the MRS $\mathcal{M}$ in the complete metric (but not ultrametric) space $(\mathbb{R}, \mathbf{d})$ with reduction steps of the form $a \to_b (a+b)$, for each $a \in \mathbb{R}, b \in \mathbb{R}^+$. More formally, $\mathcal{M}$ is defined by $\mathcal{M} = (\mathbb{R}, \mathbb{R}\times\mathbb{R}^+, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ with $\mathsf{src}((a,b)) = a$, $\mathsf{tgt}((a,b)) = a+b$, and $\mathsf{hgt}((a,b)) = b$ for all $(a,b) \in \mathbb{R}\times\mathbb{R}^+$. We then have the following reduction in $\mathcal{M}$:

$$0 \to_1 1 \to_{\frac{1}{2}} \left(1+\frac{1}{2}\right) \to_{\frac{1}{3}} \left(1+\frac{1}{2}+\frac{1}{3}\right) \to_{\frac{1}{4}} \dots$$

This reduction is trivially strongly continuous but not strongly convergent even though the sequence $(\frac{1}{1+i})_{i<\omega}$ of heights tends to 0. It is not even weakly converging since the series $\sum_{k=1}^{\infty} \frac{1}{k}$ is known to be diverging.

On the other hand, if $\mathcal{M}$ is not complete $(a_\iota)_{\iota<\alpha}$ might not converge:

Example 5.7. Consider the TRS $\mathcal{R}$ with the single rule $a \to f(a)$ and the MRS $\mathcal{M}$ which can be obtained from the induced MRS $\mathcal{M}_\mathcal{R}$ by taking $\mathcal{T}(\Sigma,\mathcal{V})$ as the set of objects instead of $\mathcal{T}^\infty(\Sigma,\mathcal{V})$. Then we have the following reduction in $\mathcal{M}$:

$$a \to_1 f(a) \to_{\frac{1}{2}} f(f(a)) \to_{\frac{1}{4}} f(f(f(a))) \to_{\frac{1}{8}} \dots$$

This reduction is trivially strongly continuous but not strongly convergent, even though the sequence $(2^{-i})_{i<\omega}$ of heights tends to 0. The reduction is not even weakly convergent as the sequence $(f^i(a))_{i<\omega}$ does converge to f^ω in the complete ultrametric space $(\mathcal{T}^\infty(\Sigma,\mathcal{V}), \mathbf{d})$ but does not converge in the incomplete ultrametric space $(\mathcal{T}(\Sigma,\mathcal{V}), \mathbf{d})$.

From the above characterisation of strong convergence, we can derive the following more general characterisation:

Proposition 5.8 (strong convergence)**.** *Let S be a reduction in an MRS $\mathcal{M}$.*

(i) If S is strongly convergent, then, for any $h \in \mathbb{R}^+$, there are at most finitely many steps in S whose height is greater than h.

(ii) If S is weakly continuous and, for any $h \in \mathbb{R}^+$, there are at most finitely many steps in S whose height is greater than h, then S is strongly continuous. If, additionally, $\mathcal{M}$ is a complete URS, then S is even strongly convergent.

Proof. (i) The proof of Kennaway [11] also works for MRSs.

(ii) Let $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{M}$. Suppose that S is weakly continuous, and that the set $\{\iota \,|\, h_\iota > h\}$ is finite for each $h \in \mathbb{R}^+$. We have to show that $\lim_{\iota\to\lambda} h_\iota = 0$ for each limit ordinal $\lambda < \alpha$. To this end, let $\varepsilon > 0$. Then choose some h such that $0 < h < \varepsilon$. Since, by hypothesis, the set $\{\iota \,|\, h_\iota > h\}$ is finite, there is some ordinal $\beta < \lambda$ such that $h_\iota \leq h < \varepsilon$ for all $\beta < \iota < \lambda$. Hence, $\lim_{\iota\to\lambda} h_\iota = 0$.

The second part of (ii) is follows from Proposition 4.3 if S is closed. Otherwise it follows from Proposition 5.5. □

The restriction to complete URSs in the second part of (ii) is essential as Example 5.6 and Example 5.7 illustrate.

From this proposition, the following corollary follows as shown by Kennaway [11]:

Corollary 5.9 (countable length of strongly convergent reductions)**.** *In an MRS every strongly convergent reduction has countable length.*

As a result of the above corollary, part (b) of Proposition 4.10 is always satisfied for strong convergence. This makes our definition of SN^∞ equivalent to that of Klop and de Vrijer [16], who considered strong convergence only.

By employing an argument similar to the one used by Klop and de Vrijer [16] for the particular case of infinitary term rewriting, we can generalise Corollary 5.9 to strongly *continuous* reductions, provided we have a complete URS.

Proposition 5.10 (countable length of strongly continuous reductions)**.** *Every strongly continuous reduction in a complete URS has countable length.*

This generalises corresponding results of Kennaway [11] and Klop and de Vrijer [16]. The above proposition is not true for weakly continuous (or convergent) reductions as pointed out by Kennaway [11].

6 Partial Order Model of Transfinite Reductions

The metric model of transfinite reductions has rather restrictive notions of convergence. For example, suppose that we have a TRS consisting of the rules

$$f(x, a) \to f(s(x), b), \quad f(x, b) \to f(s(x), a).$$

Then we can construct the reduction

$$f(0,a) \to f(s(0),b) \to f(s(s(0)),a) \to f(s(s(s(0))),b) \to \ \ldots$$

which is neither strongly nor weakly convergent in terms of its MRS semantics. The culprit is the second argument of the f symbol which constantly changes between a and b. However, excluding this "flickering", the reduction seems to converge somehow. The investigation of *partial reduction systems* is aimed at formalising this relaxation of the notion of convergence. With this tool we will be able to identify $f(s^\omega, \bot)$ as the limit of the reduction above.

To this end, a partially ordered set is employed rather than a metric space, and the limit construction is replaced by the limit inferior.

Definition 6.1 (partial reduction system)**.** A *partial reduction system (PRS)* $\mathcal{P}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{P}$,

(ii) $(A, \leq)$ is a partially ordered set,

(iii) $\mathsf{cxt}\colon\ \Phi \to A$ is a function, called the *context function*, and

(iv) if $\varphi\colon\ a \to_{\mathcal{A}} b$, then $\mathsf{cxt}(\varphi) \leq a, b$.

If the partial order $\leq$ is a complete semilattice, then $\mathcal{P}$ is called *complete*. We use the notation $\varphi\colon\ a \to_c b$ to indicate that $\mathsf{cxt}(\varphi) = c$.

Also this model can be applied to TRSs. Note, however, that we have to add a fresh constant symbol $\bot$ to the signature in order to use the partial order $\leq_\bot$:

Definition 6.2 (PRS semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The PRS *induced* by $\mathcal{R}$, denoted $\mathcal{P}_\mathcal{R}$, is given by $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt}, \leq_\bot, \mathsf{cxt})$, with $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$ the ARS $\mathcal{A}_{\mathcal{R}'}$ induced by the TRS $\mathcal{R}' = (\Sigma_\bot, R)$, $\leq_\bot$ the usual partial order on $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$, and cxt defined by

$$\mathsf{cxt}(\varphi) = t[\bot]_\pi, \text{ where } \varphi\colon\ t \to_{\pi,\rho} t'.$$

One can easily verify that the context function defined for TRSs satisfies the condition $\mathsf{cxt}(\varphi\colon\ a \to b) \leq a, b$. Since the partial order on terms forms a *complete semilattice*, this means that the PRS $\mathcal{P}_\mathcal{R}$ induced by a TRS $\mathcal{R}$ is always a *complete PRS*.

Definition 6.3 (convergence of PRSs)**.** Let $\mathcal{P} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ be a PRS. The *weak convergence* of $\mathcal{P}$, denoted $\mathcal{P}^w$, is the TARS $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^w)$, where $\mathsf{conv}^w(S) = \liminf_{\iota \to \widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The *strong convergence* of $\mathcal{P}$, denoted $\mathcal{P}^s$, is the TARS $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^s)$, where, for a reduction $S = (a_\iota \to_{c_\iota} a_{\iota+1})_{\iota<\alpha}$, $\mathsf{conv}^s(S) = a_\alpha$ if α is a successor ordinal, and $\mathsf{conv}^s(S) = \liminf_{\iota \to \alpha} c_\iota$ if α is a limit ordinal.

Since the limit inferior is always defined for complete semilattices, we immediately obtain that for complete PRSs, continuity and convergence coincide. That is, a reduction is weakly (resp. strongly) continuous iff it is weakly (resp.

strongly) convergent. This fact is the main motivation for considering the partial order model as an alternative to the metric model. As a consequence, part (a) of Proposition 4.10 is always satisfied for complete PRSs.

Returning to the initial example of this section we can now observe that the given reduction sequence weakly converges to $f(s^\omega, \bot)$ and strongly converges to $\bot$.

This example also illustrates a major difference compared to the metric model: In MRSs strong convergence is defined by restricting weak convergence. Hence, if a reduction is both weakly and strongly converging, the final result is the same and strong convergence implies weak convergence. For PRSs, however, strong convergence and weak convergence are defined differently. As a result, unlike for MRSs, strong convergence does not imply weak convergence. In order to obtain this behaviour we have to consider *total reductions*:

Definition 6.4 (total reduction)**.** Let $\mathcal{P}$ be a PRS and $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$ a reduction in $\mathcal{P}$. We say that S is *total* if each element a_ι is maximal w.r.t. the partial order of $\mathcal{P}$. If we write S as $S\colon\ a_0 \twoheadrightarrow_{\mathcal{P}^w} a_\alpha$ or $S\colon\ a_0 \twoheadrightarrow_{\mathcal{P}^s} a_\alpha$, i.e. the convergence of the reduction is explicitly stated, we additionally require a_α to be maximal for S to be total.

Proposition 6.5 (strong convergence implies weak convergence)**.** *For every total reduction S in a PRS $\mathcal{P}$, it holds that*

(i) $S\colon\ a \twoheadrightarrow_{\mathcal{P}^s} \ldots$ *implies* $S\colon\ a \twoheadrightarrow_{\mathcal{P}^w} \ldots$, *and that*

(ii) $S\colon\ a \twoheadrightarrow_{\mathcal{P}^s} b$ *implies* $S\colon\ a \twoheadrightarrow_{\mathcal{P}^w} b$.

Proof. Let $S = (a_\iota \to_{c_\iota} a_{\iota+1})_{\iota<\alpha}$. We only need to show that $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$ whenever $\mathsf{conv}^s(S)$ is a maximal object in $\mathcal{P}$. If S is closed, this is trivial. If S is open we have $\mathsf{conv}^s(S) = \liminf_{\iota\to\alpha} c_\iota \leq \liminf_{\iota\to\alpha} a_\iota = \mathsf{conv}^w(S)$ since, by definition, $c_\iota \leq a_\iota$ for each $\iota < \alpha$. Because $\mathsf{conv}^s(S)$ is maximal, we can conclude that $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$. □

Despite this difference to MRSs, the intuition of the distinction between weak and strong convergence remains the same: Like the height in an MRS, the context $\mathsf{cxt}(\varphi)$ in a PRS overapproximates the difference between the objects a, b involved in a reduction step $\varphi\colon\ a \to b$. More precisely, it underapproximates the shared structure $a \sqcap b$ of a and b, where $a \sqcap b$ denotes the glb of $\{a, b\}$ w.r.t. the partial order of the PRS. This follows from the condition $\mathsf{cxt}(\varphi) \leq a, b$ which implies $\mathsf{cxt}(\varphi) \leq a \sqcap b$. Likewise, weak and strong convergence coincide if the approximation provided by cxt is precise:

Fact 6.6 (equivalence of weak and strong convergence)**.** *Let $\mathcal{P} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ be a complete PRS with $\mathsf{cxt}(\varphi) = a \sqcap b$ for every reduction step $\varphi\colon\ a \to b \in \Phi$. Then for each reduction S in $\mathcal{P}$ we have*

(i) $S\colon\ a \twoheadrightarrow_{\mathcal{P}^w} \ldots$ *iff* $S\colon\ a \twoheadrightarrow_{\mathcal{P}^s} \ldots$, *and* (ii) $S\colon\ a \twoheadrightarrow_{\mathcal{P}^w} b$ *iff* $S\colon\ a \twoheadrightarrow_{\mathcal{P}^s} b$.

Proof. Analogously to the proof of Fact 5.4 using the equality $\liminf_{\iota\to\lambda} a_\iota = \liminf_{\iota\to\lambda}(a_\iota \sqcap a_{\iota+1})$ for all open sequences $(a_\iota)_{\iota<\lambda}$ in a complete semilattice. □

Again this fact allows us to transfer results for strong convergence [2] to the setting of weak convergence. And as for Fact 5.4 we can derive from Fact 6.6 that weak and strong convergence coincide for TRSs for which the root symbol of each right-hand side is a function symbol different from the root symbol of the corresponding left-hand side.

7 Metric vs. Partial Order Model

The main motivation for the partial order model is to have a more fine-grained notion of convergence. That is, instead of only being able to distinguish converging and diverging reductions, we have intermediate levels between full convergence and full divergence. Since, in complete PRSs, continuous reductions are always convergent, the final object of a reduction S indicates the "level of convergence" according to the partial order on objects. If it is $\bot$, the least element of the partial order, then S can be considered fully diverging. If it is a maximal element, e.g. in $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ a term not containing $\bot$, then S is fully converging.

Using this intuition, the partial order model also gives rise to a notion of *meaninglessness*: We can consider an object a of a complete PRS meaningless if there is an open reduction from a converging to $\bot$. In fact, for strong convergence in orthogonal TRSs, this concept of meaninglessness coincides with so-called *root-active terms* [3].

Under certain quite natural conditions [2], metric convergence can be considered as the fragment of partial order convergence that only considers full convergence. Vice versa, partial order convergence is a conservative extension to metric convergence which also allows partial convergence. This is, in fact, the case for TRSs:

Theorem 7.1 (PRS semantics of TRSs extends MRS semantics)**.** *For each TRS $\mathcal{R}$, the following holds for each $c \in \{w, s\}$:*

(i) $S\colon a \twoheadrightarrow_{\mathcal{P}^c_\mathcal{R}} \ldots$ *is total iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^c_\mathcal{R}} \ldots$

(ii) $S\colon a \twoheadrightarrow_{\mathcal{P}^c_\mathcal{R}} b$ *is total iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^c_\mathcal{R}} b$.

It has been shown [2] that also on so-called *term graphs*, a generalisation of terms, an appropriate complete ultrametric and complete semilattice can be defined. These concepts generalise the metric and the partial order on terms and allow to define infinitary term graph rewriting in our models of transfinite reductions. Following the framework of term graph rewriting systems (TGRSs) of Barendregt et al. [4] one can show that, at least for weak convergence, the same relation between the partial order and the metric model can be observed:

Theorem 7.2 (PRS semantics of TGRSs extends MRS semantics)**.** *For each TGRS $\mathcal{R}$, the following holds:*

(i) $S\colon a \twoheadrightarrow_{\mathcal{P}^w_\mathcal{R}} \ldots$ *is total iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^w_\mathcal{R}} \ldots$

(ii) $S\colon a \twoheadrightarrow_{\mathcal{P}^w_\mathcal{R}} b$ *is total iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^w_\mathcal{R}} b$.

8 Conclusions

The axiomatic model of transfinite reductions provides a simple framework to formulate and analyse the more concrete models presented here and is yet powerful enough to establish many of their fundamental properties. Moreover, the equivalence of transfinite properties for finite convergence and their respective finite counterparts provides additional evidence for the appropriateness of the definition of these transfinite properties.

Fact 5.4 and Fact 6.6 suggest that the metric and the partial order model have a considerable similarity in their discrimination between weak and strong convergence. This raises the question whether there is an appropriate abstraction of these two models that, in contrast to the axiomatic model, is also able to distinguish between weak and strong convergence.

Theorems 7.1 and 7.2 indicate that the partial order model is superior to the metric model as it is able to express convergence as the metric model but additionally allows to explore different levels of divergence in the metric model. Moreover, these results allow to make use of well-known properties of metric infinitary term rewriting in order to study partial order infinitary term rewriting. This was used in [3] to establish several properties of partial order infinitary orthogonal term rewriting such as compression and convergence.

The models that we presented here can be, of course, easily applied to higher-order rewriting systems [15]. However, in the metric approach to infinitary lambda-calculus [14] one usually considers various different metrics and it is not clear what the corresponding partial orders are which then admit a higher-order version of Theorem 7.1.

Acknowledgements

I would like to thank Jakob Grue Simonsen and the alert anonymous referees for carefully reading earlier drafts of this paper and providing valuable feedback. Especially, I want to thank Bernhard Gramlich for his support and his challenging questions during my work on my master's thesis which made this work possible.

Bibliography

[1] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[2] P. Bahr. Infinitary Rewriting - Theory and Applications. Master's thesis, Vienna University of Technology, Vienna, 2009.

[3] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[4] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In P. C. T. de Bakker A. J. Nijman, editor, *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Berlin / Heidelberg, 1987. doi: 10.1007/3-540-17945-3_8.

[5] S. Blom. An Approximation Based Approach to Infinitary Lambda Calculi. In V. van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin / Heidelberg, 2004. doi: 10.1007/b98160.

[6] A. Corradini. Term rewriting in CTΣ. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 468–484. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56610-4_83.

[7] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90040-9.

[8] G. Kahn and G. D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2):187–277, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90090-G.

[9] S. Kahrs. Infinitary rewriting: meta-theory and convergence. *Acta Informatica*, 44(2):91–121, 2007. ISSN 0001-5903 (Print) 1432-0525 (Online). doi: 10.1007/s00236-007-0043-2.

[10] J. L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955. ISBN 0387901256.

[11] R. Kennaway. On transfinite abstract reduction systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

[12] R. Kennaway and F.-J. de Vries. Infinitary Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 12, pages 668–711. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[13] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[14] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)00171-5.

[15] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems. In J. Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin / Heidelberg, 2005. doi: 10.1007/b135673.

[16] J. W. Klop and R. C. de Vrijer. Infinitary Normalization. In S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005. ISBN 1-904987-26-5.

[17] J.-J. Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. PhD thesis, Université Paris VII, 1978.

[18] P. H. Rodenburg. Termination and Confluence in Infinitary Term Rewriting. *The Journal of Symbolic Logic*, 63(4):1286–1296, 1998. ISSN 00224812.

[19] J. G. Simonsen. On confluence and residuals in Cauchy convergent transfinite rewriting. *Information Processing Letters*, 91(3):141–146, 2004. ISSN 0020-0190. doi: 10.1016/j.ipl.2004.03.018.

[20] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

Partial Order Infinitary Term Rewriting and Böhm Trees

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

We study an alternative model of infinitary term rewriting. Instead of a metric on terms, a partial order on partial terms is employed to formalise convergence of reductions. We consider both a weak and a strong notion of convergence and show that the metric model of convergence coincides with the partial model restricted to total terms. Hence, partial order convergence constitutes a conservative extension of metric convergence that additionally offers a fine-grained distinction between different levels of divergence.

In the second part, we focus our investigation on strong convergence of orthogonal systems. The main result is that the gap between the metric model and the partial order model can be bridged by simply extending the term rewriting system by additional rules. These extensions are the well-known Böhm extensions. Based on this result, we are able to establish that – contrary to the metric setting – orthogonal systems are both infinitarily confluent and infinitarily normalising in the partial order setting. The unique infinitary normal forms that the partial order model admits are Böhm trees.

Contents

Introduction

Infinitary term rewriting [13] extends the theory of term rewriting by giving a meaning to transfinite rewriting sequences. Its formalisation [8] is chiefly based on the metric space of terms as studied by Arnold and Nivat [2]. Other models for transfinite reductions, using for example general topological spaces [21] or partial orders [5, 6], were mainly considered to pursue quite specific purposes and have not seen nearly as much attention as the metric model. In this paper we introduce a novel formalisation of infinitary term rewriting based on the partially ordered set of partial terms [11]. We show that this model of infinitary term rewriting is superior to the metric model. This assessment includes two parts: First, the partial order model of infinitary term rewriting conservatively extends the metric model. That is, anything that can be done in the metric model can be achieved in the partial order model as well by simply restricting it to the set of total terms. Secondly, unlike the metric model, the partial order model provides a fine-grained distinction between different levels of divergence and exhibits nice properties like infinitary confluence and normalisation of orthogonal systems.

The defining core of a theory of infinitary term rewriting is its notion of convergence for transfinite reductions: Which transfinite reductions are "admissible" and what is their final outcome. In this paper we study both variants of convergence that are usually considered in the established theory of metric infinitary term rewriting: Weak convergence [8] and strong convergence [15]. For both variants we introduce a corresponding notion of convergence based on the partially ordered set of partial terms.

The first part of this paper is concerned with comparing the metric model and the partial order model both in their respective weak and strong variants. In both cases, the partial order approach constitutes a conservative extension of the metric approach: A reduction in the metric model is converging iff it is converging in the partial order model and only contains total terms.

In the second part we focus on strong convergence in orthogonal systems. To this end we reconsider the theory of meaningless terms of Kennaway et al. [16]. In particular, we consider Böhm extensions. The Böhm extension of a term rewriting system adds rewrite rules which allow to contract meaningless terms to $\bot$. The central result of the second part of this paper is that the additional rules in Böhm extensions close the gap between partial order convergence and metric convergence. More precisely, we show that reachability w.r.t. partial order convergence in a term rewriting system coincides with reachability w.r.t. metric convergence in the corresponding Böhm extension.

From this result we can easily derive a number of properties for strong partial order convergence in orthogonal systems:

- Infinitary confluence,
- infinitary normalisation, and
- compression, i.e. each reduction can be compressed to length at most ω

The first two properties exhibit another improvement over the metric model which does not have neither of these. Moreover, it means that each term has a unique infinitary normal form – its Böhm tree.

The most important tool for establishing these results is provided by a notion of complete developments that we have transferred from the metric approach to infinitary rewriting [15]. We show, that the final outcome of a complete development is unique and that, in contrast to the metric model, the partial order model admits complete developments for any set of redex occurrences. To this end, we use a technique similar to paths and finite jumps known from metric infinitary term rewriting [13, 20].

Outline

After providing the basic preliminaries for this paper in Section 1, we will briefly recapitulate the metric model of infinitary term rewriting including meaningless terms and Böhm extensions in Section 2. In Section 3, we introduce our novel approach to infinitary term rewriting based on the partial order on terms. In Section 4, we compare both models and establish that the partial order model provides a conservative extension of the metric model. In the remaining part of this paper, we focus on the strong notion of convergence. In Section 5, we establish a theory of complete developments in the setting of partial order convergence. This is then used in Section 6 to prove the equality of reachability w.r.t. partial order convergence and reachability w.r.t. metric convergence in the Böhm extension. Finally, we evaluate our results and point to interesting open questions in Section 7.

1 Preliminaries

We assume the reader to be familiar with the basic theory of ordinal numbers, orders and topological spaces [12], as well as term rewriting [23]. In the following, we briefly recall the most important notions.

1.1 Transfinite Sequences

We use $\alpha, \beta, \gamma, \lambda, \iota$ to denote ordinal numbers. A *transfinite sequence* (or simply called *sequence*) S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*. For a finite sequence $(a_i)_{i<n}$ we also use the notation $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. In particular, $\langle\rangle$ denotes an empty sequence.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a (proper) *prefix* of a sequence T, denoted $S \leq T$ (resp. $S < T$), if there is a (non-empty) sequence S' with $S \cdot S' = T$. The prefix of T of length β is denoted $T|_\beta$. The binary relation $\leq$ forms a complete semilattice. Similarly, a sequence S is a (proper) *suffix* of a sequence T if there is a (non-empty) sequence S' with $S' \cdot S = T$.

Let $S = (a_\iota)_{\iota<\alpha}$ be a sequence. A sequence $T = (b_\iota)_{\iota<\beta}$ is called a *subsequence* of S if there is a monotone function $f\colon \beta \to \alpha$ such that $b_\iota = a_{f(\iota)}$ for all $\iota < \beta$. To indicate this, we write S/f for the subsequence T. If $f(\iota) = f(0) + \iota$ for all $\iota < \beta$, then S/f is called a *segment* of S. That is, T is a segment of S iff there are two sequences T_1, T_2 such that $S = T_1 \cdot T \cdot T_2$. We write $S|_{[\beta,\gamma)}$ for the segment S/f, where $f\colon \alpha' \to \alpha$ is the mapping defined by $f(\iota) = \beta + \iota$ for all $\iota < \alpha'$, with α' the unique ordinal with $\gamma = \beta + \alpha'$. Note that in particular $S|_{[0,\alpha)} = S|_\alpha$ for each sequence S and ordinal $\alpha \leq |S|$.

1.2 Metric Spaces

A pair $(M, \mathbf{d})$ is called a *metric space* if $\mathbf{d}\colon M \times M \to \mathbb{R}_0^+$ is a function satisfying $\mathbf{d}(x, y) = 0$ iff $x = y$ (identity), $\mathbf{d}(x, y) = \mathbf{d}(y, x)$ (symmetry), and $\mathbf{d}(x, z) \leq \mathbf{d}(x, y) + \mathbf{d}(y, z)$ (triangle inequality), for all $x, y, z \in M$. If $\mathbf{d}$ instead of the triangle inequality, satisfies the stronger property $\mathbf{d}(x, z) \leq \max\{\mathbf{d}(x, y), \mathbf{d}(y, z)\}$ (strong triangle), then $(M, \mathbf{d})$ is called an *ultrametric space*. Let $(a_\iota)_{\iota<\alpha}$ be a sequence in a metric space $(M, \mathbf{d})$. The sequence $(a_\iota)_{\iota<\alpha}$ *converges* to an element $a \in M$, written $\lim_{\iota\to\alpha} a_\iota$, if, for each $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that $\mathbf{d}(a, a_\iota) < \varepsilon$ for every $\beta < \iota < \alpha$; $(a_\iota)_{\iota<\alpha}$ is *continuous* if $\lim_{\iota\to\lambda} a_\iota = a_\lambda$ for each limit ordinal $\lambda < \alpha$. The sequence $(a_\iota)_{\iota<\alpha}$ is called *Cauchy* if, for any $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that, for all $\beta < \iota < \iota' < \alpha$, we have that $\mathbf{d}(m_\iota, m_{\iota'}) < \varepsilon$. A metric space is called *complete* if each of its non-empty Cauchy sequences converges.

1.3 Partial Orders

A *partial order* $\leq$ on a set A is a binary relation on A that is *transitive*, *reflexive*, and *antisymmetric*. The pair $(A, \leq)$ is then called a *partially ordered set*. We use

$<$ to denote the strict part of $\leq$, i.e. $a < b$ iff $a \leq b$ and $b \not\leq a$. A sequence $(a_\iota)_{\iota<\alpha}$ in $(A, \leq)$ is called a (*strict*) *chain* if $a_\iota \leq a_\gamma$ (resp. $a_\iota < a_\gamma$) for all $\iota < \gamma < \alpha$. A subset D of the underlying set A is called *directed* if it is non-empty and each pair of elements in D has an upper bound in D. A partially ordered set $(A, \leq)$ is called a *complete semilattice* if it has a *least element*, every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$, and every subset of A having an upper bound also has a least upper bound. Hence, complete semilattices also admit a *greatest lower bound* (*glb*) $\bigsqcap B$ for every *non-empty* subset B of A. In particular, this means that for any non-empty sequence $(a_\iota)_{\iota<\alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha} \left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists.

It is easy to see that the limit inferior of closed sequences is simply the last element of the sequence. This is, however, only a special case of the following more general proposition:

Proposition 1.1 (invariance of the limit inferior)**.** *If $(a_\iota)_{\iota<\alpha}$ is a sequence in a partially ordered set and $(b_\iota)_{\iota<\beta}$ a non-empty suffix of $(a_\iota)_{\iota<\alpha}$, then $\liminf_{\iota\to\alpha} a_\iota = \liminf_{\iota\to\beta} b_\iota$.*

Proof. We have to show that $\bigsqcup_{\gamma<\alpha} \bigsqcap_{\gamma\leq\iota<\alpha} a_\iota = \bigsqcup_{\beta\leq\gamma<\alpha} \bigsqcap_{\gamma\leq\iota<\alpha} a_\iota = \overline{a}'$ holds for each $\beta < \alpha$. Let $b_\gamma = \bigsqcap_{\gamma\leq\iota<\alpha} a_\iota$ for each $\gamma < \alpha$, $A = \{b_\gamma \mid \gamma < \alpha\}$ and $A' = \{b_\gamma \mid \beta \leq \gamma < \alpha\}$. Note that $\overline{a} = \bigsqcup A$ and $\overline{a}' = \bigsqcup A'$. Because $A' \subseteq A$, we have that $\overline{a}' \leq \overline{a}$. On the other hand, since $b_\gamma \leq b_{\gamma'}$ for $\gamma \leq \gamma'$, we find, for each $b_\gamma \in A$, some $b_{\gamma'} \in A'$ with $b_\gamma \leq b_{\gamma'}$. Hence, $\overline{a} \leq \overline{a}'$. Therefore, due to the antisymmetry of $\leq$, we can conclude that $\overline{a} = \overline{a}'$. □

Note that the limit in a metric space has the same behaviour as the one for the limit inferior described by the proposition above. However, one has to keep in mind that – unlike the limit – the limit inferior is not invariant under taking cofinal subsequences!

With the prefix order $\leq$ on sequences we can generalise concatenation to arbitrary sequences of sequences: Let $(S_\iota)_{\iota<\alpha}$ be a sequence of sequences in a common set. The concatenation of $(S_\iota)_{\iota<\alpha}$, written $\prod_{\iota<\alpha} S_\iota$, is recursively defined as the empty sequence $\langle\rangle$ if $\alpha = 0$, $\left(\prod_{\iota<\alpha'} S_\iota\right) \cdot S_{\alpha'}$ if $\alpha = \alpha' + 1$, and $\bigsqcup_{\gamma<\alpha} \prod_{\iota<\gamma} S_\iota$ if α is a limit ordinal.

1.4 Terms

Unlike in the traditional – i.e. finitary – framework of term rewriting, we consider the set $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ of *infinitary terms* (or simply *terms*) over some *signature* Σ and a countably infinite set $\mathcal{V}$ of variables. A *signature* Σ is a countable set of symbols. Each symbol f is associated with its arity $\mathsf{ar}(f) \in \mathbb{N}$, and we write $\Sigma^{(n)}$ for the set of symbols in Σ which have arity n. The set $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ is defined as the *greatest* set T such that, for each element $t \in T$, we either have $t \in \mathcal{V}$ or $t = f(t_1, \ldots, t_k)$, where $f \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T$. We consider $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ as a superset of the set $\mathcal{T}(\Sigma, \mathcal{V})$ of *finite terms*. For a term $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. For terms $s, t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, $t(\pi)$ for the symbol in t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s. Two terms s

and t are said to *coincide* in a set of positions $P \subseteq \mathcal{P}(s) \cap \mathcal{P}(t)$ if $s(\pi) = t(\pi)$ for all $\pi \in P$. A position is also called an *occurrence* if the focus lies on the subterm at that position rather than the position itself. Two positions π_1, π_2 are called *disjoint* if neither $\pi_1 \leq \pi_2$ nor $\pi_2 \leq \pi_1$.

A *context* is a "term with holes" which are represented by a distinguished variable $\Box$. We write $C[,\ldots,]$ for a context with at least one occurrence of $\Box$, and $C\langle,\ldots,\rangle$ for a context with zero more occurrences of $\Box$. $C[t_1,\ldots,t_n]$ denotes the result of replacing the occurrences of $\Box$ in C (from left to right) by $t_1,\ldots,t_n$. $C\langle t_1,\ldots,t_n\rangle$ is defined accordingly.

A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathcal{T}^\infty(\Sigma,\mathcal{V})$. Its *domain*, denoted $\mathsf{dom}(\sigma)$, is the set $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ of variables not mapped to itself by σ. Substitutions are uniquely extended to morphisms from $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ to $\mathcal{T}^\infty(\Sigma,\mathcal{V})$, by the finality of the coalgebra $\mathcal{T}^\infty(\Sigma,\mathcal{V})$, via $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for $f \in \Sigma^{(n)}$ and $t_1,\ldots,t_n \in \mathcal{T}(\Sigma,\mathcal{V})$. Instead of $\sigma(s)$, we shall also write $s\sigma$.

On $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ a similarity measure $\mathsf{sim}(\cdot,\cdot) \in \mathbb{N} \cup \{\infty\}$ can be defined by setting

$$\mathsf{sim}(s,t) = \min\{|\pi| \mid \pi \in \mathcal{P}(s) \cap \mathcal{P}(t), s(\pi) \neq t(\pi)\} \cup \{\infty\} \qquad \text{for } s,t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$$

That is, $\mathsf{sim}(s,t)$ is the minimal depth at which s and t differ, resp. ∞ if $s = t$. Based on this, a distance function $\mathbf{d}$ can be defined by $\mathbf{d}(s,t) = 2^{-\mathsf{sim}(s,t)}$, where we interpret $2^{-\infty}$ as 0. The pair $(\mathcal{T}^\infty(\Sigma,\mathcal{V}),\mathbf{d})$ is known to form a complete ultrametric space [2]. *Partial terms*, i.e. terms over signature $\Sigma_\perp = \Sigma \uplus \{\perp\}$ with $\perp$ a fresh constant symbol, can be endowed with a binary relation $\leq_\perp$ by defining $s \leq_\perp t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\perp$. Interpreting the term $\perp$ as denoting "undefined", $\leq_\perp$ can be read as "is less defined than". The pair $(\mathcal{T}^\infty(\Sigma_\perp,\mathcal{V}),\leq_\perp)$ is known to form a complete semilattice [11]. For a partial term $t \in \mathcal{T}^\infty(\Sigma_\perp,\mathcal{V})$ we use the notation $\mathcal{P}_{\not\perp}(t)$ and $\mathcal{P}_\Sigma(t)$ for the set $\{\pi \in \mathcal{P}(t) \mid t(\pi) \neq \perp\}$ of non-$\perp$ positions resp. the set $\{\pi \in \mathcal{P}(t) \mid t(\pi) \in \Sigma\}$ of positions of function symbols. With this, $\leq_\perp$ can be characterised alternatively by $s \leq_\perp t$ iff $s(\pi) = t(\pi)$ for all $\pi \in \mathcal{P}_{\not\perp}(s)$. To explicitly distinguish them from partial terms, we call terms in $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ *total*.

1.5 Term Rewriting Systems

A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ,R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}^\infty(\Sigma,\mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ such that all variables in r are contained in l. Note that this notion of a TRS deviates slightly from the standard notion of TRSs in the literature on infinitary rewriting [13] in that it allows infinite terms on the left-hand side of rewrite rules! This generalisation will be necessary to accommodate Böhm extensions which are introduced later in Section 2.2. TRSs having only finite left-hand sides are called *left-finite*.

As in the finitary setting, every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \sigma\colon\ s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

Instead of $s \to_\mathcal{R} t$, we sometimes write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π, or simply $s \to t$. The subterm $s|_\pi$ is called a *ρ-redex* or simply *redex*, $r\sigma$ its *contractum*, and $s|_\pi$ is said to be *contracted* to $r\sigma$.

Let $\rho\colon\ l \to r$ be an term rewrite rule. The *pattern* of ρ is the context $l\sigma_\Box$, where $\sigma_\Box$ is the substitution $\{x \mapsto \Box \,|\, x \in \mathcal{V}\}$ that maps all variables to $\Box$. If t is a ρ-redex, then the pattern P of ρ is also called the *redex pattern* of t w.r.t. ρ. When referring to the occurrences in a pattern, occurrences of the symbol $\Box$ are neglected.

Let $\rho_1\colon\ l_1 \to r_1$, $\rho_2\colon\ l_2 \to r_2$ be rules in a TRS $\mathcal{R}$. The rules ρ_1, ρ_2 are said to *overlap* if there is a non-variable position π in l_1 such that $l_1|_\pi$ and l_2 are unifiable and π is not the root position $\langle\rangle$ in case ρ_1, ρ_2 are renamed copies of the same rule. A TRS is called *non-overlapping* if none of its rules overlap. A term t is called *linear* if each variable occurs at most once in t. The TRS $\mathcal{R}$ is called *left-linear* if the left-hand side of every rule in $\mathcal{R}$ is linear. It is called *orthogonal* if it is left-linear and non-overlapping.

2 Metric Infinitary Term Rewriting

In this section we briefly recall the metric model of infinitary term rewriting [15] and some of its properties. We will use the metric model in two ways: Firstly, it will serve as a yardstick to compare the partial order model to. But most importantly, we will use known results for metric infinitary rewriting and transfer them to the partial order model. In order to accomplish the latter, we will make use of Theorem 4.12 which we shall present at the end of Section 4.2.

At first we have to make clear what a *reduction* in our setting of infinitary rewriting is:

Definition 2.1 (reduction (step))**.** Let $\mathcal{R}$ be a TRS. A *reduction step* φ in $\mathcal{R}$ is a tuple (s, π, ρ, t) such that $s \to_{\pi,\rho} t$; we also write $\varphi\colon\ s \to_{\pi,\rho} t$. A *reduction* S in $\mathcal{R}$ is a sequence $(\varphi_\iota)_{\iota<\alpha}$ of reduction steps in $\mathcal{R}$ such that there is a sequence $(t_\iota)_{\iota<\widehat{\alpha}}$ of terms, with $\widehat{\alpha} = \alpha$ if S is open and $\widehat{\alpha} = \alpha + 1$ if S is closed, such that $\varphi_\iota\colon\ t_\iota \to t_{\iota+1}$. If S is finite, we write $S\colon\ t_0 \to^* t_\alpha$.

This definition of reductions is a straightforward generalisation of finite reductions. As an example consider the TRS with the single rule $a \to f(a)$. In this system we get a reduction $S\colon\ a \to^* f(f(f(a)))$ of length 3:

$$S = \langle \varphi_0\colon\ a \to f(a), \varphi_1\colon\ f(a) \to f(f(a)), \varphi_2\colon\ f(f(a)) \to f(f(f(a))) \rangle$$

In a more concise notation we write

$$S\colon\ a \to f(a) \to f^2(a) \to f^3(a)$$

Clearly, we can extend this reduction arbitrarily often which results in the following infinite reduction of length ω:

$$T\colon\ a \to f(a) \to f^2(a) \to f^3(a) \to f^4(a) \to \dots$$

However, this is as far we can go with this simple definition of reductions. As soon as we go beyond ω, we get reductions which do not make sense. For example, consider the following reduction:

$$T \cdot S\colon\ a \to f(a) \to f^2(a) \to f^3(a) \to f^4(a) \to \dots\ a \to f(a) \to f^2(a) \to f^3(a)$$

The reduction T of length ω can be extended by an arbitrary reduction, e.g. by the reduction S. The notion of reductions according to Definition 2.1 is only meaningful if restricted to reductions of length at most ω. The problem is that the ω-th step in the reduction, viz. the second step of the form $a \to f(a)$ in the example above, is completely independent of all previous steps since it does not have an immediate predecessor. This issue occurs at each limit ordinal number. An appropriate definition of a reduction of length beyond ω requires a notion of continuity to bridge the gaps that arise at limit ordinals. In the next section we will present the well-know metric approach to this. Later in Section 3, we will introduce a novel approach using partial orders.

2.1 Metric Convergence

In this section we consider two notions of *convergence* modelled by the metric on terms – a weak [8] and a strong [15] variant. Related to this notion of convergence is a corresponding notion of *continuity*. In order to distinguish both from the partial order model that we will introduce in Section 3 we will use the names *weak* resp. *strong m-convergence* and *weak* resp. *strong m-continuity*.

It is important to understand that a reduction is a *sequence of reduction steps* rather than just a sequence of terms. This is crucial for a proper definition of strong convergence resp. continuity, which does not only depend on the sequence of terms that are derived within the reduction but does also depend on the positions where the contractions take place:

Definition 2.2 (m-continuity/-convergence)**.** Let $\mathcal{R}$ be a TRS and S a non-empty reduction $(\varphi_\iota\colon t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ in $\mathcal{R}$. Then reduction S is called

(i) *weakly m-continuous* in $\mathcal{R}$, written $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} \ldots$, if $\lim_{\iota\to\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$.

(ii) *strongly m-continuous* in $\mathcal{R}$, written $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} \ldots$, if it is weakly m-continuous and for each limit ordinal $\lambda < \alpha$, the sequence $(|\pi_\iota|)_{\iota<\lambda}$ of contraction depths tends to infinity.

(iii) *weakly m-converging* to t in $\mathcal{R}$, written $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} t$, if it is weakly m-continuous and $t = \lim_{\iota\to\widehat{\alpha}} t_\iota$.

(iv) *strongly m-converging* to t in $\mathcal{R}$, written $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} t$, if it is strongly m-continuous, weakly m-converges to t and, in case that S is open, $(|\pi_\iota|)_{\iota<\alpha}$ tends to infinity.

Whenever $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} t$ or $S\colon t_0 \xrightarrow{m}_{\mathcal{R}} t$, we say that t is weakly resp. strongly *m-reachable* from t_0 in $\mathcal{R}$. By abuse of notation we use $\xrightarrow{m}_{\mathcal{R}}$ and $\xrightarrow{m}_{\mathcal{R}}$ as a binary relation to indicate weakly resp. strongly m-reachability. In order to indicate the length of S and the TRS $\mathcal{R}$, we write $S\colon t_0 \xrightarrow{m}{}^{\alpha}_{\mathcal{R}} t$ resp. $S\colon t_0 \xrightarrow{m}{}^{\alpha}_{\mathcal{R}} t$. The empty reduction $\langle\rangle$ is considered weakly/strongly m-continuous and m-convergent for any start and end term, i.e. $\langle\rangle\colon t \xrightarrow{m}_{\mathcal{R}} t$ for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$.

From the above definition it is clear that strong m-convergence implies both weak m-convergence and strong m-continuity and that both weak m-convergence

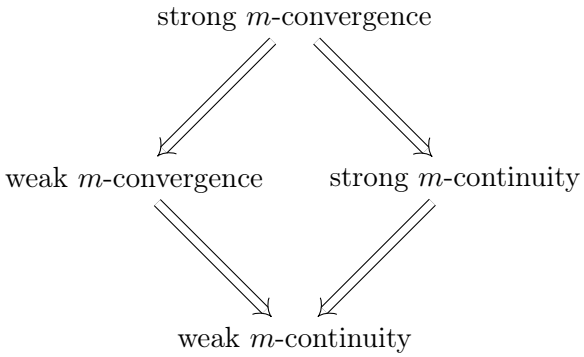


Figure 1: Relation between continuity and convergence properties.

and strong m-continuity imply weak m-continuity, respectively. This is indicated in Figure 1. It is important to recognise that m-convergence implies m-continuity. Hence, only meaningful, i.e. m-continuous, reductions can be m-convergent.

For a reduction to be weakly m-continuous, each open *proper* prefix of the underlying sequence $(t_\iota)_{\iota<\widehat{\alpha}}$ of terms must converge to the term following next in the sequence – or, equivalently, $(t_\iota)_{\iota<\widehat{\alpha}}$ must be continuous. For strong m-continuity, additionally, the depth at which contractions take place has to tend to infinity for each of the reduction's open proper prefixes. The convergence properties do only differ from the continuity properties in that they require the above conditions to hold for *all* open prefixes, i.e. including the whole reduction itself unless it is closed. For example, considering the rule $a \to f(a)$, the reduction $g(a) \to g(f(a)) \to g(f(f(a))) \to \dots$ strongly m-converges to the infinite term $g(f^\omega)$. The first step takes place at depth 1, the next step at depth 2 and so on. Having the rule $g(x) \to g(f(x))$ instead, the reduction $g(a) \to g(f(a)) \to g(f(f(a))) \to \dots$ is trivially strongly m-continuous but is now not strongly m-convergent since every step in this reduction takes place at depth 0, i.e. the sequence of reduction depths does not tend to infinity. However, the reduction still weakly m-converges to $g(f^\omega)$.

In contrast to the strong notions of continuity and convergence, the corresponding weak variants are independent from the rules that are applied during the reduction. What makes strong m-convergence (and -continuity) strong is the fact that it employs a conservative overapproximation of the differences between consecutive terms in the reduction. For weak m-convergence the distance $\mathbf{d}(t_\iota, t_{\iota+1})$ between consecutive terms in a reduction $(t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ has to tend to 0. For strong m-convergence the depth $|\pi_\iota|$ of the reduction steps has to tend to infinity. In other words, $2^{-|\pi_\iota|}$ has to tend to 0. Note that $2^{-|\pi_\iota|}$ is a conservative overapproximation of $\mathbf{d}(t_\iota, t_{\iota+1})$, i.e. $2^{-|\pi_\iota|} \geq \mathbf{d}(t_\iota, t_{\iota+1})$. So strong m-convergence is simply weak m-convergence w.r.t. this overapproximation of $\mathbf{d}$ [4]. If this approximation is actually precise, i.e. coincides with the actual value, both notions of m-convergence coincide.

Remark 2.3. The notion of m-continuity can be defined solely in terms of m-convergence [4]. More precisely, we have for each reduction $S = (t_\iota \to t_{\iota+1})_{\iota<\alpha}$ that S is weakly m-continuous iff every (open) proper prefix of $S|_\beta$ weakly m-converges to t_β. Analogously, strong m-continuity can be characterised in terms

of strong m-convergence. An easy consequence of this is that m-converging reductions are closed under concatenation, i.e. $S\colon\ s \xrightarrow{m} t, T\colon\ t \xrightarrow{m} u$ implies $S{\cdot}T\colon\ s \xrightarrow{m} u$ and likewise for strong m-convergence.

For the most part our focus in this paper is set on strong m-convergence and its partial order correspondent that we will introduce in Section 3. Weak m-convergence is well-known to be rather unruly [22]. Strong convergence is far more well-behaved [15]. Most prominently, we have the following Compression Lemma [15] which in general does not hold for weak m-convergence:

Theorem 2.4 (Compression Lemma)**.** *For each left-linear, left-finite TRS, $s \xrightarrow{m}\!\!\!\!\!\rightarrow t$ implies $s \xrightarrow{m}\!\!\!\!\!\rightarrow^{\leq\omega} t$.*

As an easy corollary we obtain that the final term of a strongly m-converging reduction can be approximated arbitrarily accurately by a finite reduction:

Corollary 2.5 (finite approximation)**.** *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $s \xrightarrow{m}\!\!\!\!\!\rightarrow t$. Then, for each depth $d \in \mathbb{N}$, there is a finite reduction $s \to^* t'$ such that t and t' coincide up to depth d, i.e. $\mathbf{d}(t, t') < 2^{-d}$.*

Proof. Assume $s \xrightarrow{m}\!\!\!\!\!\rightarrow t$. By Theorem 2.4, there is a reduction $S\colon\ s \xrightarrow{m}\!\!\!\!\!\rightarrow^{\leq\omega} t$. If S is of finite length, then we are done. If $S\colon\ s \xrightarrow{m}\!\!\!\!\!\rightarrow^{\omega} t$, then, by strong m-convergence, there is some $n < \omega$ such that all reductions steps in S after n take place at a depth greater than d. Consider $S|_n\colon\ s \to^* t'$. It is clear that t and t' coincide up to depth d. □

An important difference between m-converging reductions and finite reductions is the confluence of orthogonal systems. In contrast to finite reachability, m-reachability of orthogonal TRSs – even in its strong variant – does not necessarily have the diamond property, i.e. orthogonal systems are confluent but not infinitarily confluent [15]:

Example 2.6 (failure of infinitary confluence)**.** Consider the orthogonal TRS consisting of the *collapsing* rules $\rho_1\colon\ f(x) \to x$ and $\rho_2\colon\ g(x) \to x$ and the infinite term $t = g(f(g(f(\ldots))))$. We then obtain the reductions $S\colon\ t \xrightarrow{m}\!\!\!\!\!\rightarrow g^\omega$ and $T\colon\ t \xrightarrow{m}\!\!\!\!\!\rightarrow f^\omega$ by successively contracting all ρ_1- resp. ρ_2-redexes. However, there is no term s such that $g^\omega \xrightarrow{m}\!\!\!\!\!\rightarrow s \xleftarrow{m} f^\omega$ (or $g^\omega \xrightarrow{m} s \xleftarrow{m} f^\omega$) as both g^ω and f^ω can only be rewritten to themselves, respectively.

In the following section we discuss a method for obtaining an appropriate notion of transfinite reachability based on strong m-reachability which actually has the diamond property.

2.2 Meaningless Terms and Böhm Trees

At the end of the previous section we have seen that orthogonal TRSs are in general not infinitarily confluent. However, as Kennaway et al. [15] have shown, orthogonal TRSs are infinitarily confluent modulo so-called *hyper-collapsing* terms – in the sense that two forking strongly m-converging reductions $t \xrightarrow{m}\!\!\!\!\!\rightarrow t_1, t \xrightarrow{m}\!\!\!\!\!\rightarrow t_2$ can always be extended by two strongly m-converging reductions $t_1 \xrightarrow{m}\!\!\!\!\!\rightarrow t_3, t_2 \xrightarrow{m}\!\!\!\!\!\rightarrow t_3'$

such that the resulting terms t_3, t_3' only differ in the hyper-collapsing subterms they contain.

This result was later generalised by Kennaway et al. [16] to develop an axiomatic theory of *meaningless terms*. Intuitively, a set of meaningless terms in this setting consists of terms that are deemed meaningless since, from a term rewriting perspective, they cannot be distinguished from one another and they do not contribute any information to any computation. Kennaway et al. capture this by a set of axioms that characterise a set of meaningless terms. For orthogonal TRSs, one such set of terms, in fact the least such set, is the set of *root-active* terms [16]:

Definition 2.7 (root-activeness)**.** Let $\mathcal{R}$ be a TRS and $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$. Then t is called *root-active* if for each reduction $t \to^* t'$, there is a reduction $t' \to^* s$ to a redex s. The set of all root-active terms of $\mathcal{R}$ is denoted $\mathcal{RA}_\mathcal{R}$ or simply $\mathcal{RA}$ if $\mathcal{R}$ is clear from the context.

Intuitively speaking, as the name already suggests, root-active terms are terms that can be contracted at the root arbitrarily often, e.g. the terms f^ω and g^ω from Example 2.6.

In this paper we are only interested in this particular set of meaningless terms. So for the sake of brevity we restrict our discussion in this section to the set $\mathcal{RA}$ instead of the original more general axiomatic treatment by Kennaway et al. [16].

Since, operationally, root-active terms cannot be distinguished from each other it is appropriate to equate them [16]. This can be achieved by introducing a new constant symbol $\bot$ and making each root-active term equal to $\bot$. By adding rules which enable rewriting root-active terms to $\bot$, this can be encoded into an existing TRS [16]:

Definition 2.8 (Böhm extension)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS, and $\mathcal{U} \subseteq \mathcal{T}^\infty(\Sigma, \mathcal{V})$.

(i) A term $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ is called a $\bot, \mathcal{U}$*-instance* of a term $s \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ if t can be obtained from s by replacing each occurrence of $\bot$ in s with some term in $\mathcal{U}$.

(ii) $\mathcal{U}_\bot$ is the set of terms in $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ that have a $\bot, \mathcal{U}$-instance in $\mathcal{U}$.

(iii) The *Böhm extension* of $\mathcal{R}$ w.r.t. $\mathcal{U}$ is the TRS $\mathcal{B}_{\mathcal{R},\mathcal{U}} = (\Sigma_\bot, R \cup B)$, where

$$B = \{ t \to \bot \mid t \in \mathcal{U}_\bot \setminus \{\bot\} \}$$

We write $s \to_{\mathcal{U},\bot} t$ for a reduction step using a rule in B. If $\mathcal{R}$ and $\mathcal{U}$ are clear from the context, we simply write $\mathcal{B}$ and $\to_\bot$ instead of $\mathcal{B}_{\mathcal{R},\mathcal{U}}$ and $\to_{\mathcal{U},\bot}$, respectively.

A reduction that is strongly m-converging in the Böhm extension $\mathcal{B}$ is called *Böhm-converging*. A term t is called *Böhm-reachable* from s if there is a Böhm-converging reduction from s to t.

It is at this point where we, in fact, need the generality of allowing infinite terms on the left-hand side of rewrite rules: The additional rules of a Böhm extension allow possibly infinite terms $t \in \mathcal{U}_\bot \setminus \{\bot\}$ on the left-hand side.

Remark 2.9 (closure under substitution)**.** Note that, for orthogonal TRSs, $\mathcal{RA}$ is closed under substitutions and, hence, so is $\mathcal{RA}_\perp$ [16]. Therefore, whenever $C[t] \to_{\mathcal{RA},\perp} C[\perp]$, we can assume that $t \in \mathcal{RA}_\perp$.

With the additional rules provided by the Böhm extension, we gain infinitary confluence of orthogonal systems:

Theorem 2.10 (infinitary confluence of Böhm-converging reductions, [16])**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS. Then the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ is infinitarily confluent, i.e. $s_1 \mathrel{{}^{m}\!\!\twoheadleftarrow_{\mathcal{B}}} t \mathrel{\overset{m}{\twoheadrightarrow}_{\mathcal{B}}} s_2$ implies $s_1 \mathrel{\overset{m}{\twoheadrightarrow}_{\mathcal{B}}} t' \mathrel{{}^{m}\!\!\twoheadleftarrow_{\mathcal{B}}} s_2$.*

The lack of confluence for strongly m-converging reductions is resolved in Böhm extensions by allowing (sub-)terms, which where previously not joinable, to be contracted to $\perp$. Returning to Example 2.6, we can see that g^ω and f^ω can be rewritten to $\perp$ as both terms are root-active.

In fact, w.r.t. Böhm-convergence, every term of an orthogonal TRS has a normal form:

Theorem 2.11 (infinitary normalisation of Böhm-converging reductions, [16])**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS. Then the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ is infinitarily normalising, i.e. for each term t there is a $\mathcal{B}$-normal form Böhm-reachable from t.*

This means that each term t of an orthogonal, left-finite TRS $\mathcal{R}$ has a unique normal form in $\mathcal{B}_{\mathcal{R},\mathcal{RA}}$. This normal form is called the *Böhm tree* of t (w.r.t. $\mathcal{RA}$) [16].

The rest of this paper is concerned with establishing an alternative to the metric notion of convergence based on the partial order on terms that is equivalent to the Böhm extension approach.

3 Partial Order Infinitary Rewriting

In this section we introduce an alternative model of infinitary term rewriting which uses the partial order on terms to formalise convergence of transfinite reductions. To this end we will turn to partial terms which, like in the setting of Böhm extensions, have an additional constant symbol $\perp$. The result will be a more fine-grained notion of convergence in which, intuitively speaking, a reduction can be diverging in some positions but at the same time converging in other positions. The "diverging parts" are then indicated by a $\perp$-occurrence in the final term of the reduction:

Example 3.1. Consider the TRS consisting of the rules $h(x) \to h(g(x)), b \to g(b)$ and the term $t = f(a, b)$. In this system, we have the reduction

$$S\colon\ f(h(a), b) \to f(h(g(a)), b) \to f(h(g(a)), g(b)) \to f(h(g(g(a))), g(b)) \to \dots$$

which alternately contracts the redex in the left and in the right argument of f.

The reduction S weakly m-converges to the term $f(h(g^\omega), g^\omega)$. But it does not *strongly* m-converge as the depth at which contractions are performed does

not tend to infinity. However, this does only happen in the left argument of f, not in the other one. Within the partial order model we will still be able to obtain that S weakly converges to $f(h(g^\omega), g^\omega)$ but we will also obtain that it strongly converges to the term $f(\bot, g^\omega)$. That is, we will be able to identify that the reduction S strongly converges except at position $\langle 0 \rangle$, the first argument of f.

3.1 Partial Order Convergence

In order to formalise continuity and convergence in terms of the complete semilattice $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \leq_\bot)$ instead of the complete metric space $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \mathbf{d})$, we move from the limit of the metric space to the limit inferior of the complete semilattice:

Definition 3.2 (p-continuity/-convergence)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $S = (\varphi_\iota \colon t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ a non-empty reduction in $\mathcal{R}_\bot = (\Sigma_\bot, R)$. The reduction S is called

(i) *weakly p-continuous* in $\mathcal{R}$, written $S \colon t_0 \overset{p}{\to}_\mathcal{R} \ldots$, if $\liminf_{\iota\to\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$.

(ii) *strongly p-continuous* in $\mathcal{R}$, written $S \colon t_0 \overset{p}{\twoheadrightarrow}_\mathcal{R} \ldots$, if $\liminf_{\iota\to\lambda} c_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$, where $c_\iota = t_\iota[\bot]_{\pi_\iota}$. Each c_ι is called the *context* of the reduction step φ_ι, which we indicate by writing $\varphi_\iota \colon t_\iota \to_{c_\iota} t_{\iota+1}$.

(iii) *weakly p-converging* to t in $\mathcal{R}$, written $S \colon t_0 \overset{p}{\to}_\mathcal{R} t$, if it is weakly p-continuous and $t = \liminf_{\iota\to\widehat{\alpha}} t_\iota$.

(iv) *strongly p-converging* to t in $\mathcal{R}$, written $S \colon t_0 \overset{p}{\twoheadrightarrow}_\mathcal{R} t$, if it is strongly p-continuous and S is closed with $t = t_{\alpha+1}$ or $t = \liminf_{\iota\to\alpha} c_\iota$.

Whenever $S \colon t_0 \overset{p}{\to}_\mathcal{R} t$ or $S \colon t_0 \overset{p}{\twoheadrightarrow}_\mathcal{R} t$, we say that t is weakly resp. strongly *p-reachable* from t_0 in $\mathcal{R}$. By abuse of notation we use $\overset{p}{\to}_\mathcal{R}$ and $\overset{p}{\twoheadrightarrow}_\mathcal{R}$ as a binary relation to indicate weak resp. strong p-reachability. In order to indicate the length of S and the TRS $\mathcal{R}$, we write $S \colon t_0 \overset{p}{\to}^\alpha_\mathcal{R} t$ resp. $S \colon t_0 \overset{p}{\twoheadrightarrow}^\alpha_\mathcal{R} t$. The empty reduction $\langle\rangle$ is considered weakly/strongly p-continuous and p-convergent for any start and end term, i.e. $\langle\rangle \colon t \overset{p}{\twoheadrightarrow}_\mathcal{R} t$ for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$.

The definitions of weak p-continuity and weak p-convergence are straightforward "translations" from the metric setting to the partial order setting replacing the limit $\lim_{\iota\to\alpha}$ by the limit inferior $\liminf_{\iota\to\alpha}$. On the other hand, the definitions of the strong counterparts seem a bit different compared to the metric model: Whereas strong m-convergence simply adds a side condition regarding the depth $|\pi_\iota|$ of the reduction steps, strong p-convergence is defined in a different way compared to the weak variant. Instead of the terms t_ι of the reduction, it considers the contexts $c_\iota = t_\iota[\pi_\iota]_\bot$. However, one can surmise some similarity due to the face that the partial order model of strong convergence indirectly takes into account the position π_ι of each reduction step as well. Moreover, for the sake of understanding the intuition of strong p-convergence it is better to compare the contexts c_ι rather with the glb of two consecutive terms $t_\iota \sqcap_\bot t_{\iota+1}$ instead of the term t_ι itself. The following proposition allows precisely that.

Proposition 3.3 (limit inferior of open sequences)**.** *Let* $(a_\iota)_{\iota<\lambda}$ *be an open sequence in a complete semilattice. Then it holds that* $\liminf_{\iota<\lambda} a_\iota = \liminf_{\iota<\lambda}(a_\iota \sqcap a_{\iota+1})$.

Proof. Let $\overline{a} = \liminf_{\iota<\lambda} a_\iota$ and $\widehat{a} = \liminf_{\iota<\lambda}(a_\iota \sqcap a_{\iota+1})$. Since $a_\iota \sqcap a_{\iota+1} \leq a_\iota$ for each $\iota < \lambda$, we have $\widehat{a} \leq \overline{a}$. On the other hand, consider the sets $\overline{A}_\alpha = \{a_\iota \mid \alpha \leq \iota < \lambda\}$ and $\widehat{A}_\alpha = \{a_\iota \sqcap a_{\iota+1} \mid \alpha \leq \iota < \lambda\}$ for each $\alpha < \lambda$. Of course, we then have $\bigsqcap \overline{A}_\alpha \leq a_\iota$ for all $\alpha \leq \iota < \lambda$, and thus also $\bigsqcap \overline{A}_\alpha \leq a_\iota \sqcap a_{\iota+1}$ for all $\alpha \leq \iota < \lambda$. Hence, $\bigsqcap \overline{A}_\alpha$ is a lower bound of $\widehat{A}_\alpha$ which implies that $\bigsqcap \overline{A}_\alpha \leq \bigsqcap \widehat{A}_\alpha$. Consequently, $\overline{a} \leq \widehat{a}$ and, due to the antisymmetry of $\leq$, we can conclude that $\overline{a} = \widehat{a}$. □

With this in mind we can replace $\liminf_{\iota\to\lambda} t_\iota$ in the definition of weak p-convergence resp. p-continuity with $\liminf_{\iota\to\lambda} t_\iota \sqcap_\bot t_{\iota+1}$. From there it is easier to see the intention of moving from $t_\iota \sqcap_\bot t_{\iota+1}$ to the context $t_\iota[\bot]_{\pi_\iota}$ in order to model strong convergence:

What makes the notion of strong p-convergence (and p-continuity) *strong*, similar to the notion of strong m-convergence (resp. m-continuity), is the choice of taking the contexts $t_\iota[\bot]_{\pi_\iota}$ for defining the limit behaviour of reductions instead of the whole terms t_ι. The context $t_\iota[\bot]_{\pi_\iota}$ provides a conservative underapproximation of the shared structure $t_\iota \sqcap_\bot t_{\iota+1}$ of two consecutive terms t_ι and $t_{\iota+1}$ in a reduction step $\varphi_\iota\colon\ t_\iota \to_{\pi_\iota} t_{\iota+1}$. More specifically, we have that $t_\iota[\bot]_{\pi_\iota} \leq_\bot t_\iota \sqcap_\bot t_{\iota+1}$. That is, as in the metric model of strong convergence, the difference between two consecutive terms is overapproximated by using the position of the reduction step as an indicator. Likewise, strong p-convergence is simply weak p-convergence w.r.t. this underapproximation of $t_\iota \sqcap_\bot t_{\iota+1}$ [4]. If this approximation is actually precise, i.e. coincides with the actual value, both notions of p-convergence coincide.

Remark 3.4. As for the metric model, also in the partial order model, continuity can be defined solely in terms of convergence [4]. More precisely, we have for each reduction $S = (t_\iota \to t_{\iota+1})_{\iota<\alpha}$ that S is weakly p-continuous iff every (open) proper prefix of $S|_\beta$ weakly p-converges to t_β. Analogously, strong p-continuity can be characterised in terms of strong p-convergence. An easy consequence of this is that p-converging reductions are closed under concatenation, i.e. $S\colon\ s \hookrightarrow t$, $T\colon\ t \hookrightarrow u$ implies $S \cdot T\colon\ s \hookrightarrow u$ and likewise for strong p-convergence.

In order to understand the difference between weak and strong p-convergence let us look at a simple example:

Example 3.5. Consider the TRS with the single rule $f(x, y) \to f(y, x)$. This rule induces the following reduction:

$$S\colon\ f(a, f(g(a), g(b))) \to f(a, f(g(b), g(a))) \to f(a, f(g(a), g(b))) \to \ \ldots$$

S simply alternates between the terms $f(a, f(g(a), g(b)))$ and $f(a, f(g(b), g(a)))$ by swapping the arguments of the inner f occurrence. The reduction is depicted in Figure 2. The picture illustrates the parts of the terms that remain *unchanged* and those that remain completely *untouched* by the corresponding reduction step

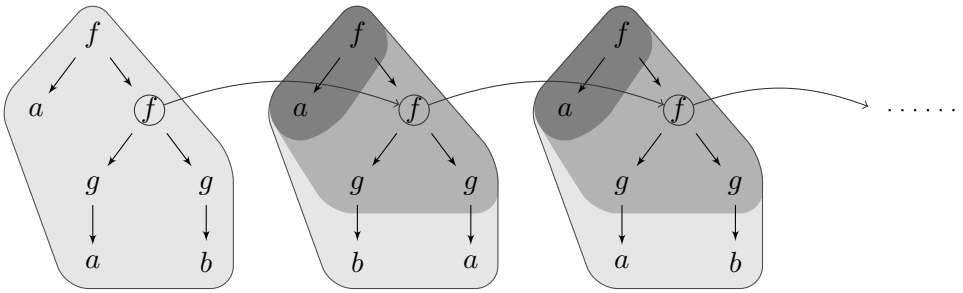


Figure 2: Reduction with stable context.

(a) Limit w.r.t. strong p-convergence.

(b) Limit w.r.t. strong p-convergence.

Figure 3: Limits of a p-converging reduction.

by using a lighter resp. a darker shade of grey. The unchanged part corresponds to the glb of the two terms of a reduction step, viz. for the first step

$$f(a, f(g(a), g(b))) \sqcap_\bot f(a, f(g(b), g(a))) = f(a, f(g(\bot), g(\bot)))$$

By symmetry, the glb of the terms of the second step is the same one. It is depicted in Figure 3a. Let $(t_i)_{i<\omega}$ be the sequence of terms of the reduction S. By definition, S weakly p-converges to $\liminf_{i<\omega} t_i$. According to Proposition 3.3, this is equal to $\liminf_{i<\omega}(t_i \sqcap_\bot t_{i+1})$. Since $t_i \sqcap_\bot t_{i+1}$ is constantly $f(a, f(g(\bot), g(\bot)))$, the reduction sequence weakly p-converges to $f(a, f(g(\bot), g(\bot)))$.

Similarly, the part of the term that remains untouched by the reduction step corresponds to the context. For the first step, this is $f(a, \bot)$. It is depicted in Figure 3b. By definition, S strongly p-converges to $\liminf_{i<\omega} c_i$ for $(c_i)_{i<\omega}$ the sequence of contexts of S. As one can see in Figure 2, the context constantly remains $f(a, \bot)$. Hence, S strongly p-converges to $f(a, \bot)$. The example sequence is a particularly simple one as both the glbs $t_i \sqcap_\bot t_{i+1}$ and the contexts c_i remain stable. In general, this is not necessary, of course.

One can clearly see from the definition that, as for their metric counterparts, weak resp. strong p-convergence implies weak resp. strong p-continuity. In contrast to the metric model, however, also the converse implication holds! Since the partial order $\leq_\bot$ on partial terms forms a complete semilattice, the limit inferior is defined for any non-empty sequence of partial terms. Hence, any weakly resp. strongly p-continuous reduction is also weakly resp. strongly p-convergent. This

is a major difference to m-convergence/-continuity. Nevertheless, p-convergence constitutes a meaningful notion of convergence: The final term of a p-convergent reduction contains a $\bot$ subterm at each position at which the reduction is "locally diverging" as we have seen in Example 3.1 and Example 3.13. In fact, as we will show in Section 4, whenever there are no '$\bot$'s involved, i.e. if there is no "local divergence", m-convergence and p-convergence coincide – both in the weak and the strong variant.

Recall that strong m-continuity resp. p-convergence implies weak m-continuity resp. m-convergence. This is not the case in the partial order setting. The reason for this is that strong p-convergence resp. p-continuity is defined differently compared to its weak variant. It uses the contexts instead of the terms in the reduction, whereas in the metric setting the strong notion of convergence is a mere restriction of the weak counterpart as we have observed earlier.

Example 3.6. Consider the TRS consisting of the rules $\rho_1\colon\ h(x) \to h(g(x))$, $\rho_2\colon\ f(x) \to g(x)$ and the following two reductions it induces:

$$S\colon\ f(h(a)) \to_{\rho_1} f(h(g(a))) \to_{\rho_1} f(h(g(g(a)))) \to_{\rho_1} \dots$$
$$T\colon\ f(\bot) \to_{\rho_2} g(\bot)$$

Then the reduction

$$S \cdot T\colon\ f(h(a)) \to_{\rho_1} f(h(g(a))) \to_{\rho_1} f(h(g(g(a)))) \to_{\rho_1} \dots\ f(\bot) \to_{\rho_2} g(\bot)$$

is clearly both strongly p-continuous and -convergent. On the other hand it is neither weakly p-continuous nor -convergent for the simple fact that S does not weakly p-converge to $f(\bot)$ but to $f(h(g^\omega)$.

Nevertheless, by observing that $\liminf_{\iota\to\alpha} c_\iota \leq_\bot \liminf_{\iota\to\alpha} t_\iota$ since $c_\iota \leq_\bot t_\iota$ for each $\iota < \alpha$, we obtain the following weaker relation between weak and strong p-convergence:

Proposition 3.7. *Let $\mathcal{R}$ be a left-linear TRS with $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$. Then there is a term $t' \geq_\bot t$ with $s \overset{p}{\to}_{\mathcal{R}} t'$.*

Proof. Let $S = (\varphi_\iota\colon\ t_\iota \to_{\rho_\iota} t_{\iota+1})_{\iota<\alpha}$ be a reduction strongly p-converging to t_α. By induction we construct for each prefix $S|_\beta$ of S a reduction $S'_\beta = (\varphi'_\iota\colon\ t'_\iota \to_{\rho_\iota} t'_{\iota+1})_{\iota<\beta}$ weakly p-converging to a term t'_β such that $t_\iota \leq_\bot t'_\iota$ for each $\iota \leq \alpha$. The proposition then follows from the case where $\beta = \alpha$.

The case $\beta = 0$ is trivial. If $\beta = \gamma + 1$, then by induction hypothesis we have a reduction $S'_\gamma\colon\ t'_0 \overset{p}{\to}_{\mathcal{R}} t'_\gamma$. Since $t_\gamma \leq_\bot t'_\gamma$ and t_γ is a ρ_γ-redex, also t'_γ is a ρ_γ-redex due to the left-linearity of $\mathcal{R}$. Hence, there is a reduction step $\varphi'_\gamma\colon\ t'_\gamma \to t'_\beta$. One can easily see that then $t_\beta \leq_\bot t'_\beta$. Hence, $S'_\beta = S'_\gamma \cdot \langle\varphi'_\gamma\rangle$ satisfies desired conditions.

If β is a limit ordinal, we can apply the induction hypothesis to obtain for each $\gamma < \beta$ a reduction $S'_\gamma = (\varphi'_\iota\colon\ t'_\iota \to_{\rho_\iota} t'_{\iota+1})_{\iota<\gamma}$ that weakly p-converges to $t'_\gamma \geq_\bot t_\gamma$. Hence, according to Remark 3.4, $S'_\beta = (\varphi'_\iota\colon\ t'_\iota \to_{\rho_\iota} t'_{\iota+1})_{\iota<\beta}$ is weakly p-continuous. Therefore, we obtain that S'_β weakly p-converges to $t'_\beta = \liminf_{\iota\to\beta} t'_\iota$. Moreover, since $c_\iota \leq_\bot t_\iota$ and $t_\iota \leq_\bot t'_\iota$ for each $\iota < \beta$, we can conclude that

$$t_\beta = \liminf_{\iota\to\beta} c_\iota \leq_\bot \liminf_{\iota\to\beta} t_\iota \leq_\bot \liminf_{\iota\to\beta} t'_\iota = t'_\beta.$$

□

And indeed, returning to Example 3.6, we can see that there is a reduction

$$f(h(a)) \to_{\rho_1} f(h(g(a))) \to_{\rho_1} f(h(g(g(a)))) \to_{\rho_1} \dots\ f(h(g^\omega)) \to_{\rho_2} g(h(g^\omega))$$

that, starting from $f(h(a))$, weakly p-converges to $g(h(g^\omega))$ which is strictly larger than $g(\bot)$.

A simple example shows that left-linearity is crucial for the above proposition:

Example 3.8. Let $\mathcal{R}$ be a TRS consisting of the rules

$$\rho_1\colon\ a \to a, \quad \rho_2\colon\ b \to b, \quad \rho_3\colon\ f(x,x) \to c.$$

We then get the strongly p-converging reduction

$$f(a,b) \to_{\rho_1} f(a,b) \to_{\rho_2} f(a,b) \to_{\rho_1} f(a,b) \to_{\rho_2} \dots\ f(\bot,\bot) \to_{\rho_3} c$$

Yet, there is no reduction in $\mathcal{R}$ that, starting from $f(a,b)$, weakly p-converges to c.

3.2 Strong p-Convergence

In this paper we are mainly focused on the strong notion of convergence. To this end, the rest of this section will be concerned exclusively with strong p-convergence. We will, however, revisit weak p-convergence in Section 4 when comparing it to weak m-convergence.

Note that in the partial order model we have to consider reductions over the extended signature $\Sigma_\bot$, i.e. reductions containing partial terms. Thus, from now on, we assume reductions in a TRS over Σ to be implicitly over $\Sigma_\bot$. When we want to make it explicit that a reduction S contains only total terms, we say that S is *total*. When we say that a strongly p-convergent reduction $S\colon\ s \overset{p}{\twoheadrightarrow} t$ is total, we mean that both the reduction S and the final term t are total.[1]

In order to understand the behaviour strong p-convergence, we need to look at how the lub and glb of a set of terms looks like. The following two lemmas provide some insight.

Lemma 3.9 (lub of terms)**.** *For each $T \subseteq \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ and $t = \bigsqcup^\bot T$, the following holds*

(i) $\mathcal{P}(t) = \bigcup_{s\in T} \mathcal{P}(s)$

(ii) $t(\pi) = f$ *iff there is some* $s \in T$ *with* $s(\pi) = f$ *for each* $f \in \Sigma \cup \mathcal{V}$*, and position* π*.*

Proof. Clause (i) follows straightforwardly from clause (ii). The "if" direction of (ii) follows from the fact that if $s \in T$, then $s \leq_\bot t$ and, therefore, $s(\pi) = f$ implies $t(\pi) = f$. For the "only if" direction assume that no $s \in T$ satisfies $s(\pi) = f$. Since, $s \leq_\bot t$ for each $s \in T$, we have $\pi \notin \mathcal{P}_\not\bot(s)$ for each $s \in T$. But then $t' = t[\bot]_\pi$ is an upper bound of T with $t' <_\bot t$. This contradicts the assumption that t is the least upper bound of T. □

[1] Note that if S is open, the final term t is not explicitly contained in S. Hence, the totality of S does not necessarily imply the totality of t.

Lemma 3.10 (glb of terms)**.** *Let $T \subseteq \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ and P a set of positions closed under prefixes such that all terms in T coincide in all occurrences in P, i.e. $s(\pi) = t(\pi)$ for all $\pi \in P$ and $s, t \in T$. Then the glb $\bar{t} = \bigsqcap^\bot T$ also coincides with all terms in T in all occurrences in P.*

Proof. Construct a term $\widehat{t}$ such that it coincides with all terms in T in all positions in P and has $\bot$ at all other positions. Then $\widehat{t}$ is a lower bound of T. By construction, $\widehat{t}$ coincides with all terms in T in all positions in P. Since $\widehat{t} \leq_\bot \bar{t}$, this property carries over to $\bar{t}$. □

Following the two lemmas above, we can observe that – intuitively speaking – the limit inferior $\liminf_{\iota\to\alpha} t_\iota$ of a sequence of terms is the term that contains those parts that become *eventually stable* in the sequence. Remaining holes in the term structure are filled with '$\bot$'s. Let us see what this means for strongly p-converging reductions:

Lemma 3.11 (non-$\bot$ symbols in open reductions)**.** *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $S\colon s \xrightarrow{p}{\!\!\!\!\!\!\rightarrow}^{\lambda}_{\mathcal{R}} t$ an open reduction with $S = (t_\iota \to_{\pi_\iota, c_\iota} t_{\iota+1})_{\iota<\lambda}$. Then the following statements are equivalent for all positions π:*

(a) $\pi \in \mathcal{P}_{\not\perp}(t)$.

(b) there is some $\alpha < \lambda$ such that $c_\iota(\pi) = t(\pi) \neq \bot$ for all $\alpha \leq \iota < \lambda$.

(c) there is some $\alpha < \lambda$ such that $t_\alpha(\pi) = t(\pi) \neq \bot$ and $\pi_\iota \not\leq \pi$ for all $\alpha \leq \iota < \lambda$.

(d) there is some $\alpha < \lambda$ such that $\pi \in \mathcal{P}_{\not\perp}(t_\alpha)$ and $\pi_\iota \not\leq \pi$ for all $\alpha \leq \iota < \lambda$.

Proof. At first consider the implication from (a) to (b). To this end, let $\pi \in \mathcal{P}_{\not\perp}(t)$ and $s_\gamma = \bigsqcap^\bot_{\gamma\leq\iota<\lambda} c_\iota$ for each $\gamma < \lambda$. Note that then $t = \bigsqcup^\bot_{\gamma<\lambda} s_\gamma$. Applying Lemma 3.9 yields that there is some $\alpha < \lambda$ such that $s_\alpha(\pi) = t(\pi)$. Moreover, for each $\alpha \leq \iota < \lambda$, we have $s_\alpha \leq_\bot c_\iota$ and, therefore, $s_\alpha(\pi) = c_\iota(\pi)$. Consequently, we obtain $c_\iota(\pi) = t(\pi)$ for all $\alpha \leq \iota < \lambda$.

Next consider the implication from (b) to (c). Let $\alpha < \lambda$ be such that $c_\iota(\pi) = t(\pi) \neq \bot$ for all $\alpha \leq \iota < \lambda$. Recall that $c_\iota = t_\iota[\bot]_{\pi_\iota}$ for all $\iota < \lambda$. Hence, the fact that $\pi \in \mathcal{P}_{\not\perp}(c_\iota)$ for all $\alpha \leq \iota < \lambda$ implies that $t_\alpha(\pi) = c_\alpha(\pi)$ and that $\pi_\iota \not\leq \pi$ for all $\alpha \leq \iota < \lambda$. Since $c_\alpha(\pi) = t(\pi) \neq \bot$, we also have $t_\alpha(\pi) = t(\pi) \neq \bot$.

The implication from (c) to (d) is trivial.

Finally, consider the implication from (d) to (a). For this purpose, let $\alpha < \lambda$ be such that (1) $\pi \in \mathcal{P}_{\not\perp}(t_\alpha)$ and (2) $\pi_\iota \not\leq \pi$ for all $\alpha \leq \iota < \lambda$. Consider the set P consisting of all positions in t_α that are prefixes of π. P is obviously closed under prefixes and, because of (2), all terms in the set $T = \{c_\iota \mid \alpha \leq \iota < \lambda\}$ coincide in all positions in P. According to Lemma 3.10, also $s_\alpha = \bigsqcap^\bot T$ coincides with all terms in T in all positions in P. Since $\pi \in P$ and $c_\alpha \in T$, we thereby obtain that $c_\alpha(\pi) = s_\alpha(\pi)$. As we also have $t_\alpha(\pi) = c_\alpha(\pi)$, due to (2), and $\pi \in \mathcal{P}_{\not\perp}(t_\alpha)$ we can infer that $\pi \in \mathcal{P}_{\not\perp}(s_\alpha)$. Since $s_\alpha \leq_\bot t$, we can then conclude $\pi \in \mathcal{P}_{\not\perp}(t)$. □

The above lemma is central for dealing with strongly p-convergent reductions. It also reveals how the final term of a strongly p-convergent reduction is constructed. According to the equality of (a) and (c), the final term has the non-$\bot$ symbol f at some position π iff some term t_α in the reduction also had this symbol

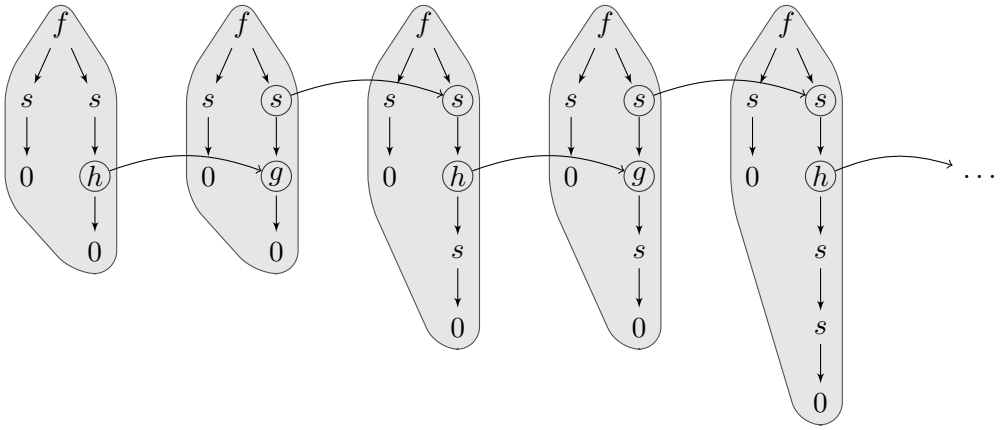


Figure 4: Reduction with two nested volatile positions.

f at this position π and no reduction after that term occurred at π or above. In this way, the final outcome of a strongly p-convergent reduction consists of precisely those parts of the intermediate terms which become *eventually persistent* during the reduction, i.e. are from some point on not subjected to contraction any more.

Now we turn to a characterisation of the parts that are not included in the final outcome of a strongly p-convergent reduction, i.e. those that do not become persistent. These parts either are either omitted or are filled by the placeholder $\bot$. We will call these positions *volatile*:

Definition 3.12 (volatility)**.** Let $\mathcal{R}$ be a TRS and $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ an open p-converging reduction in $\mathcal{R}$. A position π is said to be *volatile* in S if, for each ordinal $\beta < \lambda$, there is some $\beta \leq \gamma < \lambda$ such that $\pi_\gamma = \pi$. If π is volatile in S and no proper prefix of π is volatile in S, then π is called *outermost-volatile*.

In Example 3.1 the position $\langle 0 \rangle$ is outermost-volatile in the reduction S.

Example 3.13 (volatile positions)**.** Consider the TRS $\mathcal{R}$ consisting of the rules

$$\rho_1\colon\ h(x) \to g(x), \qquad \rho_2\colon\ s(g(x)) \to s(h(s(x)))$$

$\mathcal{R}$ admits the following reduction S of length ω:

$$\begin{aligned} S\colon\ f(s(0), s(h(0))) &\to_{\rho_1} f(s(0), s(g(0))) \to_{\rho_2} f(s(0), s(h(s(0)))) \\ &\to_{\rho_1} f(s(0), s(g(s(0)))) \to_{\rho_2} f(s(0), s(h(s(s(0))))) \end{aligned}$$

The reduction S p-converges to $f(s(0), \bot)$, i.e. we have $S\colon\ f(s(0), s(h(0))) \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} f(s(0), \bot)$. Figure 4 illustrates the reduction indicating the position of each reduction step by two circles and a reduction arrow in between. One can clearly see that both $\pi_1 = \langle 1 \rangle$ and $\pi_2 = \langle 1, 0 \rangle$ are volatile in S. Again and again reductions takes place at π_1 and π_2. Since these are the only volatile positions and π_1 is a prefix of π_2, we have that π_1 is an outermost-volatile position in S.

The following lemma shows that $\bot$ symbols are produced in open reductions precisely at outermost-volatile positions.

Lemma 3.14 ($\bot$ subterms in open reductions)**.** *Let $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ an open reduction p-converging to t_α in some TRS. Then, for every position π, we have the following:*

(i) If π is volatile in S, then $\pi \notin \mathcal{P}_\not\perp(t_\alpha)$.

(ii) $t_\alpha(\pi) = \bot$ iff

(a) π is outermost-volatile in S, or

(b) there is some $\beta < \alpha$ such that $t_\beta(\pi) = \bot$ and $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$.

(iii) Let t_ι be total for all $\iota < \alpha$. Then $t_\alpha(\pi) = \bot$ iff π is outermost-volatile in S.

Proof. (i) This follows from Lemma 3.11, in particular the equivalence of (a) and (c).

(ii) At first consider the "only if" direction. To this end, suppose that $t_\alpha(\pi) = \bot$. In order to show that then (iia) or (iib) holds, we will prove that (iib) must hold true whenever (iia) does not hold. For this purpose, we assume that π is not outermost-volatile in S. Note that no proper prefix π' of π can be volatile in S as this would imply, according to clause (i), that $\pi' \notin \mathcal{P}_\not\perp(t_\alpha)$ and, therefore, $\pi \notin \mathcal{P}(t_\alpha)$. Hence, π is also not volatile in S. In sum, no prefix of π is volatile in S. Consequently, there is an upper bound $\beta < \alpha$ on the indices of reduction steps taking place at π or above. But then $t_\beta(\pi) = \bot$ since otherwise Lemma 3.11 would imply that $t_\alpha(\pi) \neq \bot$. This shows that (iib) holds.

For the converse direction, we will show that both (iia) and (iib) independently imply that $t_\alpha(\pi) = \bot$:

(iia) Let π be outermost-volatile in S. By clause (i), this implies $\pi \notin \mathcal{P}_\not\perp(t_\alpha)$. Hence, it remains to be shown that $\pi \in \mathcal{P}(t_\alpha)$. If $\pi = \langle\rangle$, then this is trivial. Otherwise, π is of the form $\pi' \cdot i$. Since all proper prefixes of π are not volatile, there is some $\beta < \alpha$ such that $\pi_\beta = \pi$ and $\pi_\iota \not\leq \pi'$ for all $\beta \leq \iota < \alpha$. This implies that $\pi \in \mathcal{P}(t_\beta)$. Hence, $t_\beta(\pi') = f$ is a symbol having an arity of at least $i + 1$. Consequently, according to Lemma 3.11, also $t_\alpha(\pi') = f$. Since f's arity is at least $i + 1$, also $\pi = \pi' \cdot i \in \mathcal{P}(t_\alpha)$.

(iib) Let $\beta < \alpha$ such that $t_\beta(\pi) = \bot$ and $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. According to Proposition 1.1, we have that $t_\alpha = \bigsqcup^\bot_{\beta\leq\gamma<\alpha} \bigsqcap^\bot_{\gamma\leq\iota<\alpha} c_\iota$. Define $s_\gamma = \bigsqcap^\bot_{\gamma\leq\iota<\alpha} c_\iota$ for each $\gamma < \alpha$. Since from β onwards no reduction takes place at π or above, it holds that all c_ι, for $\beta \leq \iota < \alpha$, coincide in all prefixes of π. By Lemma 3.10, this also holds for all s_ι and c_ι with $\beta \leq \iota < \alpha$. Since $c_\beta(\pi) = t_\beta(\pi) = \bot$, this means that $s_\iota(\pi) = \bot$ for all $\beta \leq \iota < \alpha$. Recall that $t_\alpha = \bigsqcup^\bot_{\beta\leq\gamma<\alpha} s_\gamma$. Hence, according to Corollary 3.9, we can conclude that $t_\alpha(\pi) = \bot$.

(iii) is a special case of (ii): If each t_ι, $\iota < \alpha$, is total, then (iib) cannot be true. □

Clause (ii) shows that a $\bot$ subterm in the final term can only have its origin either in a preceding term which already contains this $\bot$ which then becomes stable, or in an outermost-volatile position. That is, it is exactly the outermost-volatile positions that generate '$\bot$'s.

We can apply this lemma to Example 3.13: As we have seen, the position $\pi_1 = \langle 1 \rangle$ is outermost-volatile in the reduction S mentioned in the example. Hence, S strongly p-converges to a term that has, according to Lemma 3.14, the symbol $\bot$ at position π_1. That is, S strongly p-converges to $f(s(0), \bot)$.

This characterisation of the final outcome of a p-converging reduction clearly shows that the partial order model captures the intuition of strong convergence in transfinite reductions even though it allows that every continuous reduction is also convergent: The final outcome only represents the parts of the reduction that *are* converging. Locally diverging parts are cut off and replaced by $\bot$.

In fact, the absence of such local divergence, or volatility, as we call it here, is equivalent to the absence of $\bot$:

Lemma 3.15 (total reductions)**.** *Let $\mathcal{R}$ be a TRS, s a total term in $\mathcal{R}$, and $S\colon s \xrightarrow{p}_{\mathcal{R}} t$. $S\colon s \xrightarrow{p}_{\mathcal{R}} t$ is total iff no prefix of S has a volatile position.*

Proof. The "only if" direction follows straightforwardly from Lemma 3.14.

We prove the "if" direction by induction on the length of S. If $|S| = 0$, then the totality of S follows from the assumption of s being total. If $|S|$ is a successor ordinal, then the totality of S follows from the induction hypothesis since single reduction steps preserve totality. If $|S|$ is a limit ordinal, then the totality of S follows from the induction hypothesis using Lemma 3.14. □

Moreover, as we shall show in the next section, if local divergences are excluded, i.e. if total reductions are considered, both the metric model and the partial order model coincide.

4 Comparing m-Convergence and p-Convergence

In this section we want to compare the metric and the partial order model of convergence. In particular, we shall show that the partial order model is only a conservative extension of the metric model: If we only consider total reductions, i.e. reductions over terms in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$, then m-convergence and p-convergence coincide in its weak and strong variant, respectively.

The first and rather trivial observation to this effect is that already on the level of single reduction steps the partial order model conservatively extends the metric model:

Fact 4.1. *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, $\mathcal{R}_\bot = (\Sigma_\bot, R)$, and $s, t \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$. Then we have*

$$s \to_{\mathcal{R},\pi} t \quad \text{iff} \quad s \to_{\mathcal{R}_\bot,\pi} t \text{ and } s \text{ is total.}$$

The next step is to establish that the underlying structures that are used to formalise convergence exhibit this behaviour as well. That is, the limit inferior in the complete semilattice $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \leq_\bot)$ is conservative extension of the limit in the complete metric space $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \mathbf{d})$. More precisely, we want to have that for a sequence $(t_\iota)_{\iota<\alpha}$ in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$

$$\liminf_{\iota\to\alpha} t_\iota = \lim_{\iota\to\alpha} t_\iota \qquad \text{whenever} \quad \begin{array}{l} \lim\limits_{\iota\to\alpha} \text{ is defined, or} \\ \liminf\limits_{\iota\to\alpha} t_\iota \text{ is total.} \end{array}$$

In Section 4.1 we shall establish the above property. This result is then used in Section 4.2 in order to show the desired property that p-convergence is a conservative extension of m-convergence in both their respective weak and strong variant.

4.1 Complete Semilattice vs. Complete Metric Space

In order to compare the complete semilattice of partial terms with the complete metric space of term, it is convenient to have an alternative characterisation of the similarity $\mathsf{sim}(s,t)$ of two terms s,t, which in turn provides an alternative characterisation of the metric $\mathbf{d}$ on terms. To this end we use the *truncation* of a term at a certain depth. This notion was originally used by Arnold and Nivat [2] to show that the $\mathbf{d}$ is a complete ultrametric on terms:

Definition 4.2 (truncation)**.** Let $d \in \mathbb{N} \cup \{\infty\}$ and $t \in \mathcal{T}^\infty(\Sigma_\perp, \mathcal{V})$. The *truncation* $t|d$ of t at depth d is defined inductively on d as follows

$$t|0 = \perp \qquad t|\infty = t$$
$$t|d+1 = \begin{cases} t & \text{if } t \in \mathcal{V} \cup \Sigma^{(0)} \\ f(t_1|d, \ldots, t_k|d) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

More concisely we can say that the truncation of a term t at depth d replaces all subterms at depth d with $\perp$. From this we can easily establish the following two properties of the truncation:

Proposition 4.3 (truncation)**.** *For each two $s,t \in \mathcal{T}^\infty(\Sigma_\perp, \mathcal{V})$ we have*

(i) $t|d \leq_\perp t$ *for all* $d \in \mathbb{N} \cup \{\infty\}$.

(ii) $s|d \leq_\perp t$ *implies* $s|d = t|d$ *for all* $d \in \mathbb{N} \cup \{\infty\}$.

(iii) $s|d = t|d$ *for all* $d \in \mathbb{N}$ *iff* $s = t$.

Proof. Straightforward. □

Recall that the similarity of two terms is the minimal depth at which they differ resp. ∞ if they are equal. However, saying that two terms differ at a certain minimal depth d is the same as saying that the truncation of the two terms at that depth d coincide. This provides an alternative characterisation of similarity:

Proposition 4.4 (characterisation of similarity)**.** *For each pair $s,t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ we have*

$$\mathsf{sim}(s,t) = \max\{d \in \mathbb{N} \cup \{\infty\} \mid s|d = t|d\}$$

Proof. Straightforward. □

We can use this characterisation to show the first part of the compatibility of the metric and the partial order:

Lemma 4.5 (metric limit equals limit inferior)**.** *Let $(t_\iota)_{\iota<\alpha}$ be a convergent sequence in $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \mathbf{d})$. Then $\lim_{\iota\to\alpha} t_\iota = \liminf_{\iota\to\alpha} t_\iota$.*

Proof. If α is a successor ordinal, this is trivial. Let α be a limit ordinal, $\widehat{t} = \lim_{\iota\to\alpha} t_\iota$, and $\bar{t} = \liminf_{\iota\to\alpha} t_\iota$. Then for each $\varepsilon \in \mathbb{R}^+$ there is a $\beta < \alpha$ such that $\mathbf{d}(\widehat{t}, t_\iota) < \varepsilon$ for all $\beta \leq \iota < \alpha$. Hence, for each $d \in \mathbb{N}$ there is a $\beta < \alpha$ such that $\mathsf{sim}(\widehat{t}, t_\iota) > d$ for all $\beta \leq \iota < \alpha$. According to Proposition 4.4, $\mathsf{sim}(\widehat{t}, t_\iota) > d$ implies $\widehat{t}|d = t_\iota|d$, which, according to Proposition 4.3, implies $\widehat{t}|d \leq_\bot t_\iota$. Therefore, $\widehat{t}|d$ is a lower bound of $T_\beta = \{t_\iota \,|\, \beta \leq \iota < \alpha\}$, i.e. $\widehat{t}|d \leq_\bot \bigsqcap^\bot T_\beta$. Since $\bar{t} = \bigsqcup^\bot_{\beta<\alpha} \bigsqcap^\bot T_\beta$, we also have that $\bigsqcap^\bot T_\beta \leq_\bot \bar{t}$. By transitivity, we obtain $\widehat{t}|d \leq_\bot \bar{t}$ for each $d \in \mathbb{N}$. According to Proposition 4.3, we can conclude $\widehat{t} = \bar{t}$ from this. □

Before we continue, we want introduce another characterisation of similarity which bridges the gap to the partial order $\leq_\bot$. In order to follow this approach, we need the to define the $\bot$-*depth* of a term $t \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$. It is the minimal depth of an occurrence of the subterm $\bot$ in t:

$$\bot\text{-}\mathsf{depth}(t) = \min\{|\pi| \,|\, t(\pi) = \bot\} \cup \{\infty\}$$

Intuitively, the glb $s \sqcap_\bot t$ of two terms s, t represents the common structure that both terms share. The similarity $\mathsf{sim}(s,t)$ is a much more condensed measure. It only provides the depth up two which the terms share a common structure. Using the $\bot$-depth we can directly condense the glb $s \sqcap_\bot t$ to the similarity $\mathsf{sim}(s,t)$:

Proposition 4.6 (characterisation of similarity)**.** *For each pair $s, t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ we have*

$$\mathsf{sim}(s,t) = \bot\text{-}\mathsf{depth}(s \sqcap_\bot t)$$

Proof. Follows from Lemma 3.10. □

We can employ this alternative characterisation of similarity to show the second part of the compatibility of the metric and the partial order:

Lemma 4.7 (total limit inferior implies Cauchy)**.** *Let $(t_\iota)_{\iota<\alpha}$ be a sequence in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ such that $\liminf_{\iota\to\alpha} t_\iota$ is total. Then $(t_\iota)_{\iota<\alpha}$ is Cauchy.*

Proof. For α a successor ordinal this is trivial. For the case that α is a limit ordinal, suppose that $(t_\iota)_{\iota<\alpha}$ is not Cauchy. That is, there is an $\varepsilon \in \mathbb{R}^+$ such that for all $\beta < \alpha$ there is a pair $\beta < \iota, \iota' < \alpha$ with $\mathbf{d}(t_\iota, t_{\iota'}) \geq \varepsilon$. Hence, there is a $d \in \mathbb{N}$ such that for all $\beta < \alpha$ there is a pair $\beta < \iota, \iota' < \alpha$ with $\mathsf{sim}(t_\iota, t_{\iota'}) \leq d$, which, according to Proposition 4.6, is equivalent to $\bot\text{-}\mathsf{depth}(t_\iota \sqcap_\bot t_{\iota'}) \leq d$. That is,

$$\text{for each } \beta < \alpha \text{ there are } \beta < \iota, \iota' < \alpha \text{ with } \bot\text{-}\mathsf{depth}(t_\iota \sqcap_\bot t_{\iota'}) \leq d \qquad (1)$$

Let $s_\beta = \bigsqcap^\bot_{\beta\leq\iota<\alpha} t_\iota$. Then $s_\beta \leq_\bot t_\iota \sqcap_\bot t_{\iota'}$ for all $\beta \leq \iota, \iota' < \alpha$, which implies $\bot\text{-}\mathsf{depth}(s_\beta) \leq \bot\text{-}\mathsf{depth}(t_\iota \sqcap_\bot t_{\iota'})$. By combining this with (1), we obtain $\bot\text{-}\mathsf{depth}(s_\beta) \leq d$. More precisely, we have that

$$\text{for each } \beta < \alpha \text{ there is a } \pi \in \mathcal{P}(s_\beta) \text{ with } |\pi| \leq d \text{ and } s_\beta(\pi) = \bot. \qquad (2)$$

Let $\bar{t} = \liminf_{\iota\to\alpha} t_\iota$. Note that $\bar{t} = \bigsqcup^\bot_{\beta<\alpha} s_\beta$. Since, according to Lemma 3.9, $\mathcal{P}(\bar{t}) = \bigcup_{\beta<\alpha} \mathcal{P}(s_\beta)$ we can reformulate (2) as follows:

$$\text{for each } \beta < \alpha \text{ there is a } \pi \in \mathcal{P}(\bar{t}) \text{ with } |\pi| \leq d \text{ and } s_\beta(\pi) = \bot. \qquad (2')$$

Since there are only finitely many positions in $\bar{t}$ of length at most d, there is some $\pi^* \in \mathcal{P}(\bar{t})$ such that

$$\text{for each } \beta < \alpha \text{ there is a } \beta \leq \gamma < \alpha \text{ with } s_\gamma(\pi^*) = \bot. \tag{3}$$

Since $s_\beta \leq_\bot s_\gamma$, whenever $\beta \leq \gamma$, we can rewrite (3) as follows:

$$s_\beta(\pi^*) = \bot \text{ for all } \beta < \alpha \text{ with } \pi^* \in \mathcal{P}(s_\beta). \tag{3'}$$

Since $\pi^* \in \mathcal{P}(\bar{t})$, we can employ Lemma 3.9 to obtain from (3') that $\bar{t}(\pi^*) = \bot$. This contradicts the assumption that $\bar{t} = \liminf_{\iota\to\alpha} t_\iota$ is total. □

The following proposition combines Lemma 4.5 and Lemma 4.7 in order to obtain the desired property that the metric and the partial order are compatible:

Proposition 4.8 (partial order conservatively extends metric)**.** *For every sequence $(t_\iota)_{\iota<\alpha}$ in $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ the following holds:*

$$\liminf_{\iota\to\alpha} t_\iota = \lim_{\iota\to\alpha} t_\iota \qquad \textit{whenever} \quad \begin{array}{l} \lim\limits_{\iota\to\alpha} \textit{ is defined, or} \\ \liminf\limits_{\iota\to\alpha} t_\iota \textit{ is total.} \end{array}$$

Proof. If $\lim_{\iota\to\alpha}$ is defined, the equality follows from Lemma 4.5. If $\liminf_{\iota\to\alpha} t_\iota$ is total, the sequence $(t_\iota)_{\iota<\alpha}$ is Cauchy by Lemma 4.7. Then, as the metric space $(\mathcal{T}^\infty(\Sigma,\mathcal{V}),\mathbf{d})$ is complete, $(t_\iota)_{\iota<\alpha}$ converges and we can apply Lemma 4.5 to conclude the equality. □

4.2 p-Convergence vs. m-Convergence

In the previous section we have established that the metric and the partial order on (partial) terms are compatible in the sense that the corresponding notions of limit and limit inferior coincide whenever the limit is defined or the limit inferior is a total term. As weak m-convergence and weak p-convergence are solely based on the limit in the metric space resp. the limit inferior in the partially ordered set, we can directly apply this result to show that both notions of convergence coincide on total reductions:

Theorem 4.9 (total weak p-convergence = weak m-convergence)**.** *For every reduction S in a TRS the following equivalences hold:*

(i) $S\colon s \xrightarrow{p} \ldots$ is total iff $S\colon s \xrightarrow{m} \ldots$, and (ii) $S\colon s \xrightarrow{p} t$ is total iff $S\colon s \xrightarrow{m} t$.

Proof. Both equivalences follow directly from Proposition 4.8 and Fact 4.1, both of which are applicable as we presuppose that each term in the reduction is total. □

In order to replicate Theorem 4.9 for the strong notions of convergence, we first need the following two lemmas that link the property of increasing contraction depth to volatile positions and the limit inferior, respectively:

Lemma 4.10 (strong m-convergence)**.** *Let $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ be an open reduction. Then $(|\pi_\iota|)_{\iota<\lambda}$ tends to infinity iff, for each position π, there is an ordinal $\alpha < \lambda$ such that $\pi_\iota \neq \pi$ for all $\alpha \leq \iota < \lambda$.*

Proof. The "only if" direction is trivial. For the converse direction, suppose that $|\pi_\iota|$ does not tend to infinity as ι approaches λ. That is, there is some depth $d \in \mathbb{N}$ such that there is no upper bound on the indices of reduction steps taking place at depth d. Let d^* be the minimal such depth. That is, there is some $\alpha < \lambda$ such that all reduction steps in $S|_{[\alpha,\lambda)}$ are at depth at least d^*, i.e. $|\pi_\iota| \geq d^*$ holds for all $\alpha \leq \iota < \lambda$. Of course, also in $S|_{[\alpha,\lambda)}$ the indices of steps at depth d^* are not bounded from above. As all reduction steps in $S|_{[\alpha,\lambda)}$ take place at depth d^* or below, $t_\iota|d^* = t_{\iota'}|d^*$ holds for all $\alpha \leq \iota, \iota' < \lambda$. That is, all terms in $S|_{[\alpha,\lambda)}$ have the same set of positions of length d^*. Let $P^* = \{\pi \in \mathcal{P}(t_n) \mid |\pi| = d^*\}$ be this set. Since there is no upper bound on the indices of steps in $S|_{[\alpha,\lambda)}$ taking place at a position in P^*, yet, P^* is finite, there has to be some position $\pi^* \in P^*$ for which there is also no such upper bound. This contradicts the assumption that there is always such an upper bound. □

Lemma 4.11 (limit inferior of truncations)**.** *Suppose $(t_\iota)_{\iota<\lambda}$ is a sequence in $\mathcal{T}^\infty(\Sigma_\perp, \mathcal{V})$ and $(d_\iota)_{\iota<\lambda}$ is a sequence in $\mathbb{N}$ such that λ is a limit ordinal and $(d_\iota)_{\iota<\lambda}$ tends to infinity. Then $\liminf_{\iota\to\lambda} t_\iota = \liminf_{\iota\to\lambda} t_\iota|d_\iota$.*

Proof. Let $\bar{t} = \liminf_{\iota\to\lambda} t_\iota|d_\iota$ and $\widehat{t} = \liminf_{\iota\to\lambda} t_\iota$. Since, according to Proposition 4.3, $t_\iota|d_\iota \leq_\perp t_\iota$ for each $\iota < \lambda$, we have that $\bar{t} \leq_\perp \widehat{t}$. Thus, it remains to be shown that also $\widehat{t} \leq_\perp \bar{t}$ holds. That is, we have to show that $\widehat{t}(\pi) = \bar{t}(\pi)$ holds for all $\pi \in \mathcal{P}_{\not\perp}(\widehat{t})$.

Let $\pi \in \mathcal{P}_{\not\perp}(\widehat{t})$. That is, $\widehat{t}(\pi) = f \neq \perp$. Hence, by Lemma 3.9, there is some $\alpha < \lambda$ with $(\bigsqcap^\perp_{\alpha\leq\iota<\lambda} t_\iota)(\pi) = f$. Let $P = \{\pi' \mid \pi' \leq \pi\}$ be the set of all prefixes of π. Note that $\bigsqcap^\perp_{\alpha\leq\iota<\lambda} t_\iota \leq_\perp t_\gamma$ for all $\alpha \leq \gamma < \lambda$. Hence, $\bigsqcap^\perp_{\alpha\leq\iota<\lambda} t_\iota$ and t_γ coincide in all occurrences in P for all $\alpha \leq \gamma < \lambda$. Because $(d_\iota)_{\iota<\lambda}$ tends to infinity, there is some $\alpha \leq \beta < \lambda$ such that $d_\gamma > |\pi|$ for all $\beta \leq \gamma < \lambda$. Consequently, since $t_\gamma|d_\gamma$ and t_γ coincide in all occurrences of length smaller than d_γ for all $\gamma < \lambda$, we have that $t_\gamma|d_\gamma$ and t_γ coincide in all occurrences in P for all $\beta \leq \gamma < \lambda$. Hence, $t_\gamma|d_\gamma$ and $\bigsqcap^\perp_{\alpha\leq\iota<\lambda} t_\iota$ coincide in all occurrences in P for all $\beta \leq \gamma < \lambda$. Hence, according to Lemma 3.10, $\bigsqcap^\perp_{\alpha\leq\iota<\lambda} t_\iota$ and $\bigsqcap^\perp_{\beta\leq\iota<\lambda} t_\iota|d_\iota$ coincide in all occurrences in P. Particularly, it holds that $(\bigsqcap^\perp_{\beta\leq\iota<\lambda} t_\iota|d_\iota)(\pi) = f$ which in turn implies by Lemma 3.9 that $\bar{t}(\pi) = f$. □

We now can prove the counterpart of Theorem 4.9 for strong convergences:

Theorem 4.12 (total strong p-convergence = strong m-convergence)**.** *For every reduction S in a TRS the following equivalences hold:*

(i) $S\colon s \overset{p}{\twoheadrightarrow} \dots$ is total iff $S\colon s \overset{m}{\twoheadrightarrow} \dots$, and (ii) $S\colon s \overset{p}{\twoheadrightarrow} t$ is total iff $S\colon s \overset{m}{\twoheadrightarrow} t$.

Proof. It suffices to only prove (ii) since (i) follows from (ii) according to Remark 3.4 resp. Remark 2.3.

Let $S = (\varphi_\iota\colon t_\iota \to_{\pi_\iota,c_\iota} t_{\iota+1})_{\iota<\alpha}$ be a reduction in a TRS $\mathcal{R}_\perp$. We continue the proof by induction on α. The case $\alpha = 0$ is trivial. If α is a successor ordinal $\beta + 1$, we can reason as follows

$$\begin{aligned} S\colon t_0 \overset{p}{\twoheadrightarrow} t_\alpha \text{ total} \quad &\text{iff} \quad S|_\beta\colon t_0 \overset{p}{\twoheadrightarrow} t_\beta \text{ and } t_\beta \to_{\mathcal{R}} t_\alpha && \text{(Remark 3.4, Fact 4.1)} \\ &\text{iff} \quad S|_\beta\colon t_0 \overset{m}{\twoheadrightarrow} t_\beta \text{ and } t_\beta \to_{\mathcal{R}} t_\alpha && \text{(ind. hyp.)} \\ &\text{iff} \quad S\colon t_0 \overset{m}{\twoheadrightarrow} t_\alpha && \text{(Remark 2.3)} \end{aligned}$$

Let α be a limit ordinal. At first consider the "only if" direction. That is, we assume that $S\colon t_0 \overset{p}{\twoheadrightarrow} t_\alpha$ is total. According to Remark 3.4, we have that $S|_\beta\colon t_0 \overset{p}{\twoheadrightarrow} t_\beta$ for each $\beta < \alpha$. Applying the induction hypothesis yields $S|_\beta\colon t_0 \overset{m}{\twoheadrightarrow} t_\beta$ for each $\beta < \alpha$. That is, following Remark 2.3, we have $S\colon t_0 \overset{m}{\twoheadrightarrow} \dots$. Since $c_\iota \leq_\perp t_\iota$ for all $\iota < \alpha$, we have that $t_\alpha = \liminf_{\iota\to\alpha} c_\iota \leq_\perp \liminf_{\iota\to\alpha} t_\iota$. Because t_α is total and, therefore, maximal w.r.t. $\leq_\perp$, we can conclude that $t_\alpha = \liminf_{\iota\to\alpha} t_\iota$. According to Proposition 4.8, this also means that $t_\alpha = \lim_{\iota\to\alpha} t_\iota$. For strong m-convergence it remains to be shown that $(|\pi_\iota|)_{\iota<\alpha}$ tends to infinity. So let us assume that this is not the case. By Lemma 4.10, this means that there is a position π such that, for each $\beta < \alpha$, there is some $\beta \leq \gamma < \alpha$ such that the step φ_γ takes place at position π. By Lemma 3.14, this contradicts the fact that t_α is a total term.

Now consider the converse direction and assume that $S\colon t_0 \overset{m}{\twoheadrightarrow} t_\alpha$. Following Remark 2.3 we obtain $S|_\beta\colon t_0 \overset{m}{\twoheadrightarrow} t_\beta$ for all $\beta < \alpha$, to which we can apply the induction hypothesis in order to get $S|_\beta\colon t_0 \overset{p}{\twoheadrightarrow} t_\beta$ for all $\beta < \alpha$ so that we have $S\colon t_0 \overset{p}{\twoheadrightarrow} \dots$, according to Remark 3.4. It remains to be shown that $t_\alpha = \liminf_{\iota\to\alpha} c_\iota$. Since S strongly m-converges to t_α, we have that (a) $t_\alpha = \lim_{\iota\to\alpha} t_\iota$, and that (b) the sequence of depths $(d_\iota = |\pi_\iota|)_{\iota<\alpha}$ tends to infinity. Using Proposition 4.8 we can deduce from (a) that $t_\alpha = \liminf_{\iota\to\alpha} t_\iota$. Due to (b), we can apply Lemma 4.11 to obtain

$$\liminf_{\iota\to\alpha} t_\iota = \liminf_{\iota\to\alpha} t_\iota|d_\iota \quad \text{and} \quad \liminf_{\iota\to\alpha} c_\iota = \liminf_{\iota\to\alpha} c_\iota|d_\iota.$$

Since $t_\iota|d_\iota = c_\iota|d_\iota$ for all $\iota < \alpha$, we can conclude that

$$t_\alpha = \liminf_{\iota\to\alpha} t_\iota = \liminf_{\iota\to\alpha} t_\iota|d_\iota = \liminf_{\iota\to\alpha} c_\iota|d_\iota = \liminf_{\iota\to\alpha} c_\iota.$$

□

The main result of this section is that we do not loose anything when switching from the metric model to the partial order model of infinitary term rewriting. Restricted to the domain of the metric model, i.e. total terms, both models coincide in the strongest possible sense as Theorem 4.9 and Theorem 4.12 confirm.

At the same time, however, the partial order model provides more structure. Whenever the metric model can only conclude divergence, the partial order model can qualify the degree of divergence. If a reduction p-converges to $\perp$, it can be considered completely divergent. If it p-converges to a term that only contains $\perp$ as proper subterms, it can be recognised as being only partially divergent with the diverging parts of the reduction indicated by '$\perp$'s, whereas complete absence of '$\perp$'s then indicates complete convergence.

In the rest of this paper we will put our focus on strong convergence. Theorem 4.12 will be one of the central tools in Section 6 where we shall discover that Böhm-reachability coincides with strong p-reachability in orthogonal systems. The other crucial tool that we will leverage is the existence and uniqueness of complete developments. This is the subject of the subsequent section.

5 Strongly p-Converging Complete Developments

The purpose of this section is to establish a theory of residuals and complete developments in the setting of strongly p-convergent reductions. Intuitively speaking, the residuals of a set of redexes are the remains of this set of redexes after a reduction, and a complete development of a set of redexes is a reduction which only contracts residuals of these redexes and ends in a term with no residuals.

Complete developments are a well-known tool for proving (finitary) confluence of orthogonal systems [23]. It has also been lifted to the setting of strongly m-convergent reductions in order to establish (restricted forms of) infinitary confluence of orthogonal systems [15]. As we have seen in Example 2.6, m-convergence in general does not have this property.

After introducing residuals and complete developments in Section 5.1, we will show in Section 5.2 resp. Section 5.3 that complete developments do always exist and that their final terms are uniquely determined. We then use this in Section 5.4 to show the Infinitary Strip Lemma for strongly p-converging reductions which is a crucial tool for proving our main result in Section 6.

5.1 Residuals

At first we need to formalise the notion of residuals. It is virtually equivalent to the definition for strongly m-convergent reduction by Kennaway et al. [15]:

Definition 5.1 (descendants, residuals)**.** Let $\mathcal{R}$ be a TRS, $S\colon t_0 \xrightarrow{p,\alpha}_{\mathcal{R}} t_\alpha$, and $U \subseteq \mathcal{P}_{\not\perp}(t_0)$. The *descendants* of U by S, denoted $U/\!/S$, is the set of positions in t_α inductively defined as follows:

(a) If $\alpha = 0$, then $U/\!/S = U$.

(b) If $\alpha = 1$, i.e. $S\colon t_0 \to_{\pi,\rho} t_1$ for some $\rho\colon l \to r$, take any $u \in U$ and define the set R_u as follows: If $\pi \not\leq u$, then $R_u = \{u\}$. If u is in the pattern of the ρ-redex, i.e. $u = \pi \cdot \pi'$ with $\pi' \in \mathcal{P}_\Sigma(l)$, then $R_u = \emptyset$. Otherwise, i.e. if $u = \pi \cdot w \cdot x$, with $l|_w \in \mathcal{V}$, then $R_u = \{\pi \cdot w' \cdot x \,|\, r|_{w'} = l|_w\}$. Define $U/\!/S = \bigcup_{u\in U} R_u$.

(c) If $\alpha = \beta + 1$, then $U/\!/S = (U/\!/S|_\beta)/\!/S|_{[\beta,\alpha)}$

(d) If α is a limit ordinal, then $U/\!/S = \mathcal{P}_{\not\perp}(t_\alpha) \cap \liminf_{\iota\to\alpha} U/\!/S|_\iota$
That is, $u \in U/\!/S$ iff $u \in \mathcal{P}_{\not\perp}(t_\alpha)$ and $\exists\beta < \alpha \forall\beta \leq \iota < \alpha\colon\ u \in U/\!/S|_\iota$

If, in particular, U is a set of redex occurrences, then $U/\!/S$ is also called the set of *residuals* of U by S. Moreover, by abuse of notation, we write $u/\!/S$ instead of $\{u\}/\!/S$.

Clauses (a), (b) and (c) are as in the finitary setting. Clause (d) lifts the definition to the infinitary setting. However, the only difference to the definition of Kennaway et al. is, that we consider partial terms here. Yet, for technical reasons, the notion of descendants has to be restricted to non-$\perp$ occurrences. Since $\perp$ cannot be a redex, this is not a restriction for residuals, though.

Remark 5.2. One can see that the descendants of a set of non-$\bot$-occurrences is again a set of non-$\bot$-occurrences. The restriction to non-$\bot$-occurrences has to be made explicit for the case of open reductions. In fact, without this explicit restriction the definition would yield descendants which might not even be occurrences in the final term t_α of the reduction. For example, consider the system with the single rule $f(x) \to x$ and the strongly p-convergent reduction

$$S\colon\ f^\omega \to f^\omega \to\ \dots\ \bot$$

in which each reduction step contracts the redex at the root of f^ω. Consider the set $U = \{\langle\rangle, \langle 0\rangle, \langle 0,0\rangle, \langle 0,0,0\rangle, \dots\}$ of all positions in t^ω. Without the abovementioned restriction, the descendants of U by S would be U itself as the descendants of U by each proper prefix of S is also U. However, none of the positions $\langle 0\rangle, \langle 0,0\rangle, \langle 0,0,0\rangle, \dots \in U$ is even a position in the final term $\bot$. The position $\langle\rangle \in U$ occurs in $\bot$, but only as a $\bot$-occurrence. With the restriction to non-$\bot$-occurrences we indeed get the expected result $U/\!/S = \emptyset$.

The definition of descendants of open reductions is quite subtle which makes it fairly cumbersome to use in proofs. The lemma below establishes an alternative characterisation which will turn out to be useful in later proofs:

Lemma 5.3 (descendants of open reductions)**.** *Let $\mathcal{R}$ be a TRS, $S\colon\ s \xrightarrow{p}{\twoheadrightarrow}^{\lambda}_{\mathcal{R}} t$ and $U \subseteq \mathcal{P}_{\not\perp}(s)$, with λ a limit ordinal and $S = (t_\iota \to_{\pi_\iota, c_\iota} t_{\iota+1})_{\iota<\lambda}$. Then it holds that for each position π*

$$\pi \in U/\!/S \quad \textit{iff} \quad \textit{there is some } \beta < \lambda \textit{ with } \pi \in U/\!/S|_\beta \textit{ and } \forall \beta \le \iota < \lambda\ \ \pi_\iota \not\le \pi.$$

Proof. We first prove the "only if" direction. To this end, assume that $\pi \in U/\!/S$. Hence, it holds that

$$\pi \in \mathcal{P}_{\not\perp}(t) \text{ and there is some } \gamma_1 < \lambda \text{ such that } \pi \in U/\!/S|_\iota \text{ for all } \gamma_1 \le \iota < \lambda \quad (1)$$

Particularly, we have that $t(\pi) \neq \bot$. Applying Lemma 3.11 then yields that

$$\text{there is some } \gamma_2 < \lambda \text{ such that } \pi_\iota \not\le \pi \text{ for all } \gamma_2 \le \iota < \lambda \quad (2)$$

Now take $\beta = \max\{\gamma_1, \gamma_2\}$. Then it holds that $\pi \in U/\!/S|_\beta$ and that $\pi_\iota \not\le \pi$ for all $\beta \le \iota < \lambda$ due to (1) and (2), respectively.

Next, consider the converse direction of the statement: Let $\beta < \lambda$ be such that $\pi \in U/\!/S|_\beta$ and $\pi_\iota \not\le \pi$ for all $\beta \le \iota < \lambda$. We will show that $\pi \in U/\!/S$ by proving the stronger statement that $\pi \in U/\!/S|_\gamma$ for all $\beta \le \gamma \le \lambda$. We do this by induction on γ.

For $\gamma = \beta$, this is trivial. Let $\gamma = \gamma' + 1 > \beta$. Note that, by definition, $U/\!/S|_\gamma = \big(U/\!/S|_{\gamma'}\big)\,/\!/S|_{[\gamma',\gamma)}$. Hence, since for the γ'-th step we have, by assumption, $\pi_{\gamma'} \not\le \pi$ and for the preceding reduction we have, by induction hypothesis, that $\pi \in U/\!/S|_{\gamma'}$, we can conclude that $\pi \in U/\!/S|_\gamma$.

Let $\gamma > \beta$ be a limit ordinal. By induction hypothesis, we have that $\pi \in U/\!/S|_\iota$ for each $\beta \le \iota < \gamma$. Particularly, this implies that $\pi \in \mathcal{P}_{\not\perp}(t_\beta)$. Together with the assumption that $\pi_\iota \not\le \pi$ for all $\beta \le \iota < \gamma$, this yields that $\pi \in \mathcal{P}_{\not\perp}(t_\gamma)$ according to Lemma 3.11. Hence, $\pi \in U/\!/S|_\gamma$. □

The following lemma confirms the expected monotonicity of descendants:

Lemma 5.4 (monotonicity of descendants)**.** *Let $\mathcal{R}$ be a TRS, $S\colon s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ and $U, V \subseteq \mathcal{P}_{\not\perp}(s)$. If $U \subseteq V$, then $U/\!/S \subseteq V/\!/S$.*

Proof. Straightforward induction on the length of S. □

This lemma can be generalised such that we can see that descendants are defined "pointwise":

Proposition 5.5 (pointwise definition of descendants)**.** *Suppose $\mathcal{R}$ is a TRS, $S\colon s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ and $U \subseteq \mathcal{P}_{\not\perp}(s)$. Then it holds that $U/\!/S = \bigcup_{u\in U} u/\!/S$.*

Proof. Let $S = (t_\iota \to_{\pi_\iota, c_\iota} t_{\iota+1})_{\iota<\alpha}$. For $\alpha = 0$ and $\alpha = 1$, the statement is trivially true. If $\alpha = \alpha' + 1 > 1$, then abbreviate $S|_{\alpha'}$ and $S|_{[\alpha',\alpha)}$ by S_1 and S_2, respectively, and reason as follows:

$$U/\!/S = (U/\!/S_1)/\!/S_2 \overset{IH}{=} (\underbrace{\bigcup_{u\in U} \overbrace{u/\!/S_1}^{V_u}}_{V})/\!/S_2 \overset{IH}{=} \bigcup_{u\in V} u/\!/S_2$$
$$= \bigcup_{u\in U}\bigcup_{v\in V_u} v/\!/S_2 \overset{IH}{=} \bigcup_{u\in U} V_u/\!/S_2 = \bigcup_{u\in U} (u/\!/S_1)/\!/S_2 = \bigcup_{u\in U} u/\!/S$$

Let α be a limit ordinal. The "$\supseteq$" direction of the equation follows from Lemma 5.4. For the converse direction, assume that $\pi \in U/\!/S$. By Lemma 5.3, there is some $\beta < \alpha$ such that $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$ and $\pi \in U/\!/S|_\beta$. Applying the induction hypothesis yields that $\pi \in \bigcup_{u\in U} u/\!/S|_\beta$, i.e. there is some $u^* \in U$ such that $\pi \in u^*/\!/S|_\beta$. By employing Lemma 5.3 again, we can conclude that $\pi \in u^*/\!/S$ and, therefore, that $\pi \in \bigcup_{u\in U} u/\!/S$. □

Note that the above proposition fails if we would include $\perp$-occurrences in our definition of descendants: Reconsider the example in Remark 5.2 and assume we would drop the restriction to non-$\perp$-occurrences. Then the residuals $u/\!/S$ of each occurrence $u \in U$ would be empty, whereas the residuals $U/\!/S$ of all occurrences would be the root occurrence $\langle\rangle$.

Proposition 5.6 (uniqueness of descendants)**.** *Let $\mathcal{R}$ be TRS, $S\colon s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ and $U, V \subseteq \mathcal{P}_{\not\perp}(s)$. If $U \cap V = \emptyset$, then $U/\!/S \cap V/\!/S = \emptyset$.*

Proof. We will prove the contraposition of the statement. To this end, suppose that there is some occurrence $w \in U/\!/S \cap V/\!/S$. By Proposition 5.5, there are occurrences $u \in U$ and $v \in V$ such that $w \in u/\!/S \cap v/\!/S$. We will show by induction on the length of S that then $u = v$ and, therefore, $U \cap V \neq \emptyset$. If S is empty, then this is trivial. If S is of successor ordinal length, then this follows straightforwardly from the induction hypothesis. If S is open, then $u = v$ follows from the induction hypothesis using Lemma 5.3. □

Remark 5.7. The two propositions above imply that each descendant $u' \in U/\!/S$ of a set U of occurrences is the descendant of a uniquely determined occurrence $u \in U$, i.e. $u' \in u/\!/S$ for exactly one $u \in U$. This occurrence u is also called the *ancestor* of u' by S.

The following proposition confirms a property of descendants that one expects intuitively: The descendants of descendants are again descendants. That is, the concept of descendants is composable.

Proposition 5.8 (descendants of sequential reductions)**.** *Suppose $\mathcal{R}$ is a TRS, $S\colon t_0 \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t_1$, $T\colon t_1 \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t_2$, and $U \subseteq \mathcal{P}_{\not\perp}(t_0)$. Then $U/\!/S \cdot T = (U/\!/S)/\!/T$.*

Proof. Straightforward proof by induction on the length of T. □

The following proposition confirms that the disjointness of occurrences is propagated through their descendants:

Proposition 5.9 (disjoint descendants)**.** *The descendants of a set of pairwise disjoint occurrences are pairwise disjoint as well.*

Proof. Let $S\colon s \overset{p}{\twoheadrightarrow}^{\alpha} t$ and let U be a set of pairwise disjoint occurrences in s. We show that $U/\!/S$ is also a set of pairwise disjoint occurrences by induction on α.

For α being 0, the statement is trivial, and, for α being a successor ordinal, the statement follows straightforwardly from the induction hypothesis. Let α be limit ordinal and suppose that there are two occurrences $u, v \in U/\!/S$ which are not disjoint. By definition, there are ordinals $\beta_1, \beta_2 < \alpha$ such that $u \in U/\!/S|_\iota$ for all $\beta_1 \leq \iota < \alpha$, and $v \in U/\!/S|_\iota$ for all $\beta_2 \leq \iota < \alpha$. Let $\beta = \max\{\beta_1, \beta_2\}$. Then we have that $u, v \in U/\!/S|_\beta$. This, however, contradicts the induction hypothesis which, in particular, states that $U/\!/S|_\beta$ is a set of pairwise disjoint occurrences. □

For the definition of complete developments it is important that the descendants of redex occurrences are again redex occurrences:

Proposition 5.10 (residuals)**.** *Let $\mathcal{R}$ be an orthogonal TRS, $S\colon s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ and U a set of redex occurrences in s. Then $U/\!/S$ is a set of redex occurrences in t.*

Proof. Let $S = (t_\iota \to_{\pi_\iota, c_\iota} t_{\iota+1})_{\iota<\alpha}$. We proceed by induction on α. For α being 0, the statement is trivial, and, for α a successor ordinal, the statement follows straightforwardly from the induction hypothesis.

So assume that α is a limit ordinal and that $\pi \in U/\!/S$. We will show that $t|_\pi$ is a redex. From Lemma 5.3 we obtain that

$$\text{there is some } \beta < \alpha \text{ with } \pi \in U/\!/S|_\beta \text{ and } \pi_\iota \not\leq \pi \text{ for all } \beta \leq \iota < \alpha. \tag{1}$$

By applying the induction hypothesis, we get that π is a redex occurrence in t_β. Hence, there is some rule $l \to r \in R$ such that $t_\beta|_\pi$ is an instance of l.

We continue this proof by showing the following stronger claim:

$$\text{for all } \beta \leq \gamma \leq \alpha \qquad t_\gamma|_\pi \text{ is an instance of } l, \text{ and} \tag{2}$$
$$c_\iota|_\pi \text{ is an instance of } l \text{ for all } \beta \leq \iota < \gamma \tag{3}$$

For the special case $\gamma = \alpha$ the above claim (2) implies that $t|_\pi$ is a redex.

We proceed by an induction on γ. For $\gamma = \beta$, part (2) of the claim has already been shown and (3) is vacuously true. Let $\gamma = \gamma' + 1 > \beta$. According to the induction hypothesis, (2) and (3) hold for γ'. Hence, it remains to be shown

that both $t_\gamma|_\pi$ and $c_{\gamma'}|_\pi$ are instances of l. At first consider $c_{\gamma'}|_\pi$. Recall that $c_{\gamma'} = t_{\gamma'}[\bot]_{\pi'_\gamma}$. At first consider the case where π and π'_γ are disjoint. Then $c_{\gamma'}|_\pi = t_{\gamma'}|_\pi$. Since, by induction hypothesis, $t_{\gamma'}|_\pi$ is an instance of l, so is $c_{\gamma'}|_\pi$. Next, consider the case where π and $\pi_{\gamma'}$ are not disjoint. Because of (1), we then have that $\pi < \pi_{\gamma'}$, i.e. there is some non-empty π' with $\pi_{\gamma'} = \pi \cdot \pi'$. Since $\mathcal{R}$ is non-overlapping, π' cannot be a position in the pattern of the redex $t_{\gamma'}|_\pi$ w.r.t. l. Therefore, also $c_{\gamma'}|_\pi$ is an instance of l. So in either case $c_{\gamma'}|_\pi$ is an instance of l. Since $c_{\gamma'} \leq_\bot t_\gamma$, also $t_\gamma|_\pi$ is an instance of l.

Let $\gamma > \beta$ be a limit ordinal. Part (3) of the claim follows immediately from the induction hypothesis. Hence, $c_\iota|_\pi$ is an instance of l for all $\beta \leq \iota < \gamma$. This and (1) implies that all terms in the set $T = \{c_\iota \mid \beta \leq \iota < \gamma\}$ coincide in all occurrences in the set

$$P = \left\{\pi' \,\middle|\, \pi' \leq \pi\right\} \cup \left\{\pi \cdot \pi' \,\middle|\, \pi' \in \mathcal{P}_\Sigma(l)\right\}$$

P is obviously closed under prefixes. Therefore, we can apply Lemma 3.10 in order to obtain that $\bigsqcap^\bot T$ coincides with all terms in T in all occurrences in P. Since $\bigsqcap^\bot T \leq_\bot t_\gamma$, this property carries over to t_γ. Consequently, also $t_\gamma|_\pi$ is an instance of l. □

Next we want to establish an alternative characterisation of descendants based on labellings. This is a well-known technique [23] that keeps track of descendants by labelling the symbols at the relevant positions in the initial term. In order to formalise this idea, we need to extend a given TRS such that it can also deal with terms that contain labelled symbols:

Definition 5.11 (labelled TRSs/terms)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS.

(i) The *labelled signature* Σ^l is defined as $\Sigma \cup \left\{f^l \,\middle|\, f \in \Sigma\right\}$. The arity of the function symbol f^l is the same as that of f. The symbols f^l are called *labelled*; the symbols $f \in \Sigma$ are called *unlabelled*. Terms over Σ^l are called *labelled terms*. Note that the symbol $\bot \in \Sigma_\bot$ has no corresponding labelled symbol $\bot^l$ in the labelled signature $\Sigma^l_\bot$.

(ii) Labelled terms can be projected back to the original unlabelled ones by removing the labels via the projection function $\|\cdot\|$:

$$\begin{aligned} \|\cdot\| \colon \mathcal{T}^\infty(\Sigma^l_\bot, \mathcal{V}) &\to \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}) \\ \|\bot\| = \bot \qquad \|x\| &= x \qquad \text{for all } x \in \mathcal{V}, \text{ and} \\ \left\|f^l(t_1, \ldots, t_k)\right\| = \|f(t_1, \ldots, t_k)\| &= f(\|t_1\|, \ldots, \|t_k\|) \quad \text{for all } f \in \Sigma^{(k)} \end{aligned}$$

(iii) The *labelled TRS* $\mathcal{R}^l$ is defined as (Σ^l, R^l) with $R^l = \{l \to r \mid \|l\| \to r \in R\}$.

(iv) For each rule $l \to r \in R^l$, we define its unlabelled original $\|l \to r\| = \|l\| \to r$ in R.

(v) Let $t \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ and $U \subseteq \mathcal{P}_\not\perp(t)$. The term $t^{(U)} \in \mathcal{T}^\infty(\Sigma^l_\bot, \mathcal{V})$ is defined by

$$t^{(U)}(\pi) = \begin{cases} t(\pi) & \text{if } \pi \notin U \\ t(\pi)^l & \text{if } \pi \in U \end{cases}$$

That is, $\left\|t^{(U)}\right\| = t$ and the labelled symbols in $t^{(U)}$ are exactly those at positions in U.

The key property which is needed in order to make the labelling approach work is that any reduction in a left-linear TRS that starts in some term t can be lifted for any labelling t' of t to a unique equivalent reduction in the corresponding labelled TRS that starts in t':

Proposition 5.12 (lifting reductions to labelled TRSs)**.** *Let $\mathcal{R} = (\Sigma, R)$ be a left-linear TRS, $S = (s_\iota \to_{\rho_\iota,\pi_\iota} s_{\iota+1})_{\iota<\alpha}$ a reduction strongly p-converging to s_α in $\mathcal{R}$, and $t_0 \in \mathcal{T}^\infty(\Sigma^l_\perp, \mathcal{V})$ a labelled term with $\|t_0\| = s_0$. Then there is a unique reduction $T = (t_\iota \to_{\rho'_\iota,\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ strongly p-converging to t_α in $\mathcal{R}^l$ such that*

(a) $\|t_\iota\| = s_\iota$, $\|\rho'_\iota\| = \rho_\iota$, for all $\iota < \alpha$, and

(b) $\|t_\alpha\| = s_\alpha$.

Proof. We prove this by an induction on α. For the case of α being zero, the statement is trivially true. For the case of α being a successor ordinal, the statement follows straightforwardly from the induction hypothesis (the argument is the same as for finite reductions; e.g. consult [23]).

Let α be a limit ordinal. By induction hypothesis, for each proper prefix $S|_\gamma$ of S there is a uniquely defined strongly p-convergent reduction T_γ in $\mathcal{R}^l$ satisfying (a) and (b). Since the sequence $(S|_\iota)_{\iota<\alpha}$ forms a chain w.r.t. the prefix order $\leq$, so does the corresponding sequence $(T_\iota)_{\iota<\alpha}$. Hence the sequence $T = \bigsqcup_{\iota<\alpha} T_\iota$ is well-defined. By construction, $T_\gamma \leq T$ holds for each $\gamma < \alpha$, and we can use the induction hypothesis to obtain part (a) of the proposition. In order to show $s_\alpha = \|t_\alpha\|$, we prove the two inequalities $s_\alpha \leq_\perp \|t_\alpha\|$ and $s_\alpha \geq_\perp \|t_\alpha\|$:

To show $\|t_\alpha\| \leq_\perp s_\alpha$, we take some $\pi \in \mathcal{P}_{\not\perp}(\|t_\alpha\|)$ and show that $\|t_\alpha\|(\pi) = s_\alpha(\pi)$. Let $f = \|t_\alpha\|(\pi)$. That is, either $t_\alpha(\pi) = f$ or $t_\alpha(\pi) = f^l$. In either case, we can employ Lemma 3.11 to obtain some $\beta < \alpha$ such that $t_\beta(\pi) = f$ resp. $t_\beta(\pi) = f^l$ and $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. Since, by (a), $s_\beta = \|t_\beta\|$, we have in both cases that $s_\beta(\pi) = f$. By applying Lemma 3.11 again, we get that $s_\alpha(\pi) = f$, too.

Lastly, we show the converse inequality $s_\alpha \leq_\perp \|t_\alpha\|$. For this purpose, let $\pi \in \mathcal{P}_{\not\perp}(s_\alpha)$ and $f = s_\alpha(\pi)$. By Lemma 3.11, there is some $\beta < \alpha$ such that $s_\beta(\pi) = f$ and $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. Since, by (a), $s_\beta = \|t_\beta\|$, we have that $t_\beta(\pi) \in \{f, f^l\}$. Applying Lemma 3.11 again then yields that $t_\alpha(\pi) \in \{f, f^l\}$ and, therefore, $\|t_\alpha\|(\pi) = f$. □

Having this, we can establish an alternative characterisation of descendants using labellings:

Proposition 5.13 (alternative characterisation of descendants)**.** *Let $\mathcal{R}$ be a left-linear TRS, $S\colon s_0 \overset{p}{\twoheadrightarrow}_\mathcal{R} s_\alpha$, and $U \subseteq \mathcal{P}_{\not\perp}(s_0)$. Following Proposition 5.12, let $T\colon t_0 \overset{p}{\twoheadrightarrow}_\mathcal{R} t_\alpha$ be the unique lifting of S to $\mathcal{R}^l$ starting with the term $t_0 = s_0^{(U)}$. Then it holds that $t_\alpha = s_\alpha^{(U//S)}$. That is, for all $\pi \in \mathcal{P}_{\not\perp}(s_\alpha)$, it holds that $t_\alpha(\pi)$ is labelled iff $\pi \in U//S$.*

Proof. Let $S = (s_\iota \to_{\pi_\iota} s_{\iota+1})_{\iota<\alpha}$ and $T = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$. We prove the statement by an induction on the length α of S. If $\alpha = 0$, then the statement is trivially true. If α is a successor ordinal, then a straightforward argument shows that the statement follows from the induction hypothesis. Here the restriction to left-linear systems is vital.

Let α be a limit ordinal and let $\pi \in \mathcal{P}_{\not\perp}(s_\alpha)$. We can then reason as follows:

$$
\begin{aligned}
&t_\alpha(\pi) \text{ is labelled} \\
&\quad \text{iff} \quad \exists \beta < \alpha\colon\ t_\beta(\pi) \text{ is labelled and } \forall \beta \leq \iota < \alpha\colon\ \pi_\iota \not\leq \pi && \text{(Lem. 3.11)} \\
&\quad \text{iff} \quad \pi \in U/\!/S|_\beta \text{ and } \forall \beta \leq \iota < \alpha\colon\ \pi_\iota \not\leq \pi && \text{(ind. hyp.)} \\
&\quad \text{iff} \quad \pi \in U/\!/S && \text{(Lem. 5.3)}
\end{aligned}
$$

□

5.2 Constructing Complete Developments

Complete developments are usually defined for (almost) orthogonal systems. This ensures that the residuals of redexes are again redexes. Since we are going to use complete developments for potentially overlapping systems as well, we need to make restrictions on the set of redex occurrences instead:

Definition 5.14 (conflicting redex occurrences)**.** Two distinct redex occurrences u, v in a term t are called *conflicting* if there is a position π such that $v = u \cdot \pi$ and π is a pattern position of the redex at u, or, vice versa, $u = v \cdot \pi$ and π is a pattern position of the redex at v. If this is not the case, then u and v are called *non-conflicting*.

One can easily see that in an orthogonal TRS any pair of redex occurrences is non-conflicting.

Definition 5.15 ((complete) development)**.** Let $\mathcal{R}$ be a left-linear TRS, s a partial term in $\mathcal{R}$, and U a set of pairwise non-conflicting redex occurrences in s.

(i) A *development* of U in s is a strongly p-converging reduction $S\colon\ s \overset{p}{\twoheadrightarrow}^\alpha t$ in which each reduction step $\varphi_\iota\colon\ t_\iota \to_{\pi_\iota} t_{\iota+1}$ contracts a redex at $\pi_\iota \in U/\!/S|_\iota$.

(ii) A development $S\colon\ s \overset{p}{\twoheadrightarrow} t$ of U in s is called *complete*, denoted $S\colon\ s \overset{p}{\twoheadrightarrow}_U t$, if $U/\!/S = \emptyset$.

This is a straightforward generalisation of complete developments known from the finitary setting and coincides with the corresponding formalisation for metric infinitary rewriting [15] if restricted to total terms.

The restriction to non-conflicting redex occurrences is essential in order guarantee that the redex occurrences are independent from each other:

Proposition 5.16 (non-conflicting residuals)**.** *Let $\mathcal{R}$ be a left-linear TRS, s a partial term in $\mathcal{R}$, U a set of pairwise non-conflicting redex occurrences in s, and $S\colon\ s \twoheadrightarrow_U t$ a development of U in s. Then also $U/\!/S$ is a set of pairwise non-conflicting redex occurrences.*

Proof. This can be proved by induction on the length of S. The part showing that the descendants are again redex occurrences can be copied almost verbatim from Proposition 5.10. Instead of referring to the non-overlappingness of the system one can refer to the non-conflictingness of the preceding residuals which can be assumed by the induction hypothesis. The part of the induction proof that shows non-conflictingness is analogous to Proposition 5.9. □

It is rather easy to show that complete developments of sets of non-conflicting redex occurrences do always exists in the partial order setting. The reason for this is that strongly p-continuous reductions do always strongly p-converge as well. This means that as long as there are (residuals of) redex occurrences left after an incomplete development, one can extend this development arbitrarily by contracting some of the remaining redex occurrences. The only thing that remains to be shown is that one can devise a reduction strategy which eventually contracts (all residuals of) all redexes. The proposition below shows that a parallel-outermost reduction strategy will always yield a complete development in a left-linear system.

Proposition 5.17 (complete developments)**.** *Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, and U a set of pairwise non-conflicting redex occurrences in t. Then U has a complete development in t.*

Proof. Let $t_0 = t$, $U_0 = U$ and V_0 the set of outermost occurrences in U_0. Furthermore, let $S_0\colon\ t_0 \overset{p}{\twoheadrightarrow}_{V_0} t_1$ be some complete development of V_0 in t_0. S_0 can be constructed by contracting the redex occurrences in V_0 in a left-to-right order. This step can be continued for each $i < \omega$ by taking $U_{i+1} = U_i//S_i$, where $S_i\colon\ t_i \overset{p}{\twoheadrightarrow}_{V_i} t_{i+1}$ is some complete development of V_i in t_i with V_i the set of outermost redex occurrences in U_i.

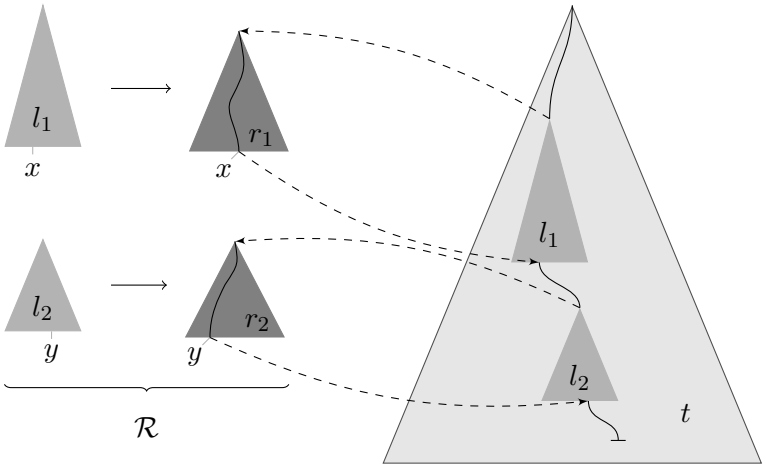
Note that then, by iterating Proposition 5.8, it holds that

$$U//S_0 \cdot \ldots \cdot S_{n-1} = U_n \quad \text{for all } n < \omega \tag{1}$$

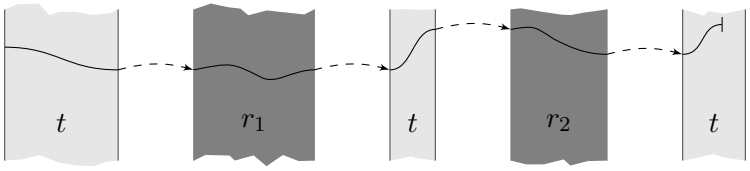
If there is some $n < \omega$ for which $U_n = \emptyset$, then $S_0 \cdot \ldots \cdot S_{n-1}$ is a complete development of U according to (1).

If this is not the case, consider the reduction $S = \prod_{i<\omega} S_i$, i.e. the concatenation of all 'S_i's. We claim that S is a complete development of U. Suppose that this is not the case, i.e. $U//S \neq \emptyset$. Hence, there is some $u \in U//S$. Since all 'U_i's are non-empty, so are the 'V_i's. Consequently, all 'S_i's are non-empty reductions which implies that S is a reduction of limit ordinal length, say λ. Therefore, we can apply Lemma 5.3 to infer from $u \in U//S$ that there is some $\alpha < \lambda$ such that $u \in U//S|_\alpha$ and all reduction steps beyond α do not take place at u or above. This is not possible due to the parallel-outermost reduction strategy that S adheres. □

This shows that complete developments of any set of redex occurrences do always exist in any (almost) orthogonal system. This is already an improvement over strongly m-converging reductions which only allow this if no collapsing rules are present or the considered set of redex occurrences does not contain a so-called *infinite collapsing tower*.



(a) Constructing a path in a term.



(b) The constructed path.

Figure 5: A path.

In the subsequent section we shall show that complete developments are also unique in the sense that the final outcome is uniquely determined by the initial set of redexes occurrences.

5.3 Uniqueness of Complete Developments

The goal of this section is to show that the final term of a complete development is uniquely determined by the initial set of redex occurrences U. There are several techniques to show that in the metric model. One of these approaches, introduced by Kennaway and de Vries [13] and detailed by Ketema and Simonsen [18, 19] for infinitary combinatory reduction systems, uses so-called *paths*. Paths are constructed such that they, starting from the root, run through the initial term t of the complete development, and whenever a redex occurrence of the development is encountered, the path jumps to the root of the right-hand side of the corresponding rule and jumps back to the term t when it reaches a variable in the right-hand side.

Figure 5a illustrates this idea. It shows a path in a term t that encounters two redex occurrences of the complete development. As soon as such a redex occurrence is encountered, the path jumps to the right-hand side of the corresponding rule as indicated by the dashed arrows. Then the path runs through the right-hand side. When a variable is encountered, the path jumps back to the position of the term t that matches the variable. This jump is again indicated by a dashed arrow. The path that is obtained by this construction is shown in

Figure 5b. With the collection of the thus obtained paths one can then construct the final term of the complete development. This technique – slightly modified – can also be applied in the present setting:

Definition 5.18 (path)**.** Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, and U a set of pairwise non-conflicting redex occurrence in t. A $U, \mathcal{R}$-*path* (or simply *path*) in t is a sequence of length at most ω containing so-called *nodes* and *edges* in an alternating manner like this:

$$\langle n_0, e_0, n_1, e_1, n_2, e_2, \ldots \rangle$$

where the 'n_i's are nodes and the 'e_i's are edges. A node is either a pair of the form $(\top, \pi)$ with $\pi \in \mathcal{P}(t)$ or a triple of the form (r, π, u) with r the right-hand side of a rule in $\mathcal{R}$, $\pi \in \mathcal{P}(r)$, and $u \in U$. Edges are denoted by arrows $\to$. Both edges and nodes might be labelled by elements in $\Sigma_\perp \cup \mathcal{V}$ and $\mathbb{N}$, respectively. We write paths as the one sketched above as

$$n_0 \to n_1 \to n_2 \to \cdots$$

or, when explicitly indicating labels, as

$$n_0{}^{l_0} \xrightarrow{l_1} n_1{}^{l_2} \xrightarrow{l_3} n_2{}^{l_4} \xrightarrow{l_5} \cdots$$

where empty labels are explicitly given by the symbol $\emptyset$. If a path has a segment of the form $n \to n'$, then we say there is an edge from n to n' or that n has an outgoing edge to n'.

Every path starts with the node $(\top, \langle\rangle)$ and is either infinitely long or ends with a node. For each node n having an outgoing edge to a node n', the following must hold:

(1) If n is of the form $(\top, \pi)$, then

  (a) $n' = (\top, \pi \cdot i)$ and the edge is labelled by i, with $\pi \cdot i \in \mathcal{P}(t)$ and $\pi \notin U$, or

  (b) $n' = (r, \langle\rangle, \pi)$ and the edge is unlabelled, with $t|_\pi$ a ρ-redex for $\rho\colon\ l \to r \in R$ and $\pi \in U$.

(2) If n is of the form (r, π, u), then

  (a) $n' = (r, \pi \cdot i, u)$ and the edge is labelled by i, with $\pi \cdot i \in \mathcal{P}(r)$, or

  (b) $n' = (\top, u \cdot \pi')$ and the edge is unlabelled, with $t|_u$ a ρ-redex for $\rho\colon\ l \to r \in R$, $r|_\pi$ a variable, and π' the unique occurrence of $r|_\pi$ in l. .

Additionally, the nodes of a path are supposed to be labelled in the following way:

(3) A node of the form $(\top, \pi)$ is unlabelled if $\pi \in U$ and is labelled by $t(\pi)$ otherwise.

(4) A node of the form (r, π, u) is unlabelled if $r|_\pi$ is a variable and labelled by $r(\pi)$ otherwise.

Remark 5.19. The above definition is actually a coinductive one. This is necessary to also define paths of infinite length. Also in [13] paths are considered to be possibly infinite, although they are defined inductively and are, therefore, finite.

The purpose of nodes of the form $(\top, \pi)$ and (r, π, u), respectively, is that they encode that the path is currently at position π in the term t resp. r. The additional component u provides the information that the path jumped to the right-hand side r from the redex $t|_u$. The labellings of the nodes represent the symbols at the current location of the path, unless it is a redex occurrence in the main term or a variable occurrence in a right-hand side. The labellings of the edges provide information on how the path moves through the terms: A labelling i represents a move along the i-th edge in the term tree from the current location whereas an empty labelling indicates a jump from or to a right-hand side of a rule.

Remark 5.20. Our definition of paths deviates slightly from the usual definition found in the literature [15, 19, 20]: In our setting, term nodes are of the form $(\top, \pi)$. The symbol $\top$ is used to indicate that we are in the host term t. In the definitions found in the literature, the term t itself is used for that, i.e. term nodes are of the form (t, π). Our definition of paths makes them less dependant on the term t they are constructed in. This makes it easier to construct a path in a host term from other paths in different host terms. This will become necessary in the proof of Lemma 5.33. However, we have to keep in mind that the node labels in a path are dependent on the host term under consideration. Thus, the labelling of a path might be different depending on which host term it is considered to be in.

Returning to the schematic example illustrated in Figure 5, we can observe how the construction of a path is carried out: The path starts with a segment in the term t. This segment is entirely regulated by the rule (1a); all its edges and nodes are labelled according to (1a) and (3). The jump to the right-hand side r_1 following that initial segment is justified by rule (1b). This jump consists of a node $(\top, u_1)$, unlabelled according to (3), corresponding to the redex occurrence u_1, and an unlabelled edge to the node $(r_1, \langle\rangle, u_1)$, corresponding to the root of the right-hand side r_1. The segment of the path that runs through the right-hand side r_1 is subject to rule (2a); again all its nodes and edges are labelled, now according to (2a) and (4). As soon as a variable is reached in the right-hand side term (in the schematic example it is the variable x) a jump to the main term t is performed as required by rule (2b). This jump consists of a node (r_1, π, u_1), unlabelled according to (4), where π is the current position in r_1, i.e. the variable occurrence, and an unlabelled edge to the node $(\top, u_1 \cdot \pi')$. The position π' is the occurrence of the variable x in the left-hand side. As we only consider left-linear systems, this occurrence is unique. Afterwards, the same behaviour is repeated: A segment in t is followed by a jump to a segment in the right-hand side r_2 which is in turn followed by a jump back to a final segment in t.

Note that paths do not need to be maximal. As indicated in the schematic example, the path ends somewhere within the main term, i.e. not necessarily at a constant symbol or a variable. What the example does not show, but which is

obvious from the definition, is that paths can also terminate within a right-hand side. A jump back to the main term is only required if variable is encountered.

The purpose of the concept of paths is to simulate the contraction of all redexes of the complete development in a locally restricted manner, i.e. only along some branch of the term tree. This locality will keep the proofs more concise and makes them easier to understand once we have grasped the idea behind paths. The strategy to prove our conjecture of uniquely determined final terms is to show that paths can be used to define a term and that a contraction of a redex of the complete development preserves a property of the collection of all paths which ensures that the induced term remains invariant. Then we only have to observe that the induced term of paths in a term with no redexes (in U) is the term itself.

The following fact is obvious from the definition of a path.

Fact 5.21. *Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t.*

(i) An edge in a $U,\mathcal{R}$-path in t is unlabelled iff the preceding node is unlabelled.

(ii) Any prefix of a $U,\mathcal{R}$-path in t that ends in a node is also a $U,\mathcal{R}$-path in t.

As we have already mentioned, collapsing rules and in particular so-called infinite collapsing towers play a significant role in m-convergent reductions as they obstruct complete developments. Also in our setting of p-convergent reductions they are important as they are responsible for volatile positions:

Definition 5.22 (collapsing rules)**.** Let $\mathcal{R}$ be a TRS.

(i) A rule $l \to r$ in $\mathcal{R}$ is called *collapsing* if r is a variable. The unique position of the variable r in l is called the *collapsing position* of the rule.

(ii) A ρ-redex is called *collapsing* if ρ is a collapsing rule.

(iii) A *collapsing tower* is a non-empty sequence $(u_i)_{i<\alpha}$ of collapsing redex occurrences in a term t such that $u_{i+1} = u_i \cdot \pi_i$ for each $i < \alpha$, where π_i is a collapsing position of the redex at u_i. It is called *maximal* if it is not a proper prefix of another collapsing tower.

One can easily see that, in orthogonal TRSs, maximal collapsing towers in the same term are uniquely determined by their topmost redex occurrence. That is, two maximal collapsing towers $(u_i)_{i<\alpha}, (v_i)_{i<\alpha}$ in the same term are equal iff $u_0 = v_0$.

As mentioned, we shall use the $U,\mathcal{R}$-paths in a term t in order to define the final term of a complete development of U in t. However, in order to do that, we only need the information that is available from the labellings. The inner structure of nodes is only used for the bookkeeping that necessary for defining paths. The following notion of traces defines projections to the labels of paths:

Definition 5.23 (trace)**.** Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, and U a set of pairwise non-conflicting redex occurrences in t.

(i) Let Π be a $U,\mathcal{R}$-path in t. The *trace* of Π, denoted $\mathsf{tr}_t(\Pi)$, is the projection of Π to the labelling of its nodes and edges ignoring empty labels and the node label $\bot$.

(ii) $\mathcal{P}(t, U, \mathcal{R})$ is used to denote the set of all $U, \mathcal{R}$-paths in t that end in a labelled node, or are infinite but have a finite trace. The set of traces of paths in $\mathcal{P}(t, U, \mathcal{R})$ is denoted by $\mathcal{T}r(t, U, \mathcal{R})$.

By Fact 5.21, the trace of a path is a sequence alternating between elements in $\Sigma \cup \mathcal{V}$ and $\mathbb{N}$, which, if non-empty, starts with an element in $\Sigma \cup \mathcal{V}$. Moreover, by definition, $\mathcal{T}r(t, U, \mathcal{R})$ is a set of finite traces of $U, \mathcal{R}$-paths in t.

As we have mentioned in Remark 5.20, the labelling of a path depends on the host term under consideration. Hence, also the trace of a path is depended on the host term. That is why we need to index the trace mapping $\mathsf{tr}_t(\cdot)$ with the corresponding host term t.

Example 5.24. Consider the term $t = g(f(g(h(\bot))))$ and the TRS $\mathcal{R}$ consisting of the two rules

$$f(x) \to h(x), \qquad h(x) \to x.$$

Furthermore, let U be the set of all redex occurrences in t, viz. $U = \{\langle 0\rangle, \langle 0\rangle^3\}$. The following path Π is a $U, \mathcal{R}$-path in t:

$$\begin{aligned}(\top, \langle\rangle)^g \xrightarrow{0} (\top, \langle 0\rangle)^{\emptyset} \xrightarrow{\emptyset} (r_1, \langle\rangle, \langle 0\rangle)^h \xrightarrow{0} (r_1, \langle 0\rangle, \langle 0\rangle)^{\emptyset} \xrightarrow{\emptyset} (\top, \langle 0\rangle^2)^g \\ \xrightarrow{0} (\top, \langle 0\rangle^3)^{\emptyset} \xrightarrow{\emptyset} (r_2, \langle\rangle, \langle 0\rangle^3)^{\emptyset} \xrightarrow{\emptyset} (\top, \langle 0\rangle^4)^{\bot}\end{aligned}$$

As a matter of fact, Π is the greatest path of t. Hence, according to Fact 5.21, the set of all prefixes of Π ending in a node is the set of all $U, \mathcal{R}$-paths in t. Note that since Π itself ends in a labelled node, it is contained in $\mathcal{P}(t, U, \mathcal{R})$. The trace $\mathsf{tr}_t(\Pi)$ of Π is the sequence

$$\langle g, 0, h, 0, g, 0\rangle$$

Now consider the term $t' = g(f(g(h^\omega)))$ and the set U' of all its redexes, viz. $U' = \{\langle 0\rangle\} \cup \{\langle 0\rangle^3, \langle 0\rangle^4, \dots\}$. Then the following path Π' is a $U, \mathcal{R}$-path in t':

$$\begin{aligned}(\top, \langle\rangle)^g \xrightarrow{0} (\top, \langle 0\rangle)^{\emptyset} \xrightarrow{\emptyset} (r_1, \langle\rangle, \langle 0\rangle)^h \xrightarrow{0} (r_1, \langle 0\rangle, \langle 0\rangle)^{\emptyset} \xrightarrow{\emptyset} (\top, \langle 0\rangle^2)^g \xrightarrow{0} (\top, \langle 0\rangle^3)^{\emptyset} \\ \xrightarrow{\emptyset} (r_2, \langle\rangle, \langle 0\rangle^3)^{\emptyset} \xrightarrow{\emptyset} (\top, \langle 0\rangle^4)^{\emptyset} \xrightarrow{\emptyset} (r_2, \langle\rangle, \langle 0\rangle^4)^{\emptyset} \xrightarrow{\emptyset} (\top, \langle 0\rangle^5)^{\emptyset} \xrightarrow{\emptyset} \dots\end{aligned}$$

Π' is the greatest path of t'. The trace $\mathsf{tr}_{t'}(\Pi')$ of Π' is the sequence

$$\langle g, 0, h, 0, g, 0\rangle$$

Since Π' is infinitely long but has a finite trace, it is contained in $\mathcal{P}(t', U, \mathcal{R})$.

The lemma below shows that there is a one-to-one correspondence between paths in $\mathcal{P}(t, U, \mathcal{R})$ and their traces in $\mathcal{T}r(t, U, \mathcal{R})$.

Lemma 5.25 ($\mathsf{tr}_t(\cdot)$ is a bijection)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. $\mathsf{tr}_t(\cdot)$ is a bijection from $\mathcal{P}(t, U, \mathcal{R})$ to $\mathcal{T}r(t, U, \mathcal{R})$.*

Proof. By definition, $\mathsf{tr}_t(\cdot)$ is surjective. Let Π_1, Π_2 be two paths having the same trace. We will show that then $\Pi_1 = \Pi_2$ by an induction on the length of the common trace.

Let $\mathsf{tr}_t(\Pi_1) = \langle\rangle$. Following Fact 5.21, there are two different cases: The first case is that $\Pi_1 = \Pi \cdot (\top, \pi)^\perp$, where the prefix Π corresponds to a finite maximal collapsing tower $(u_i)_{i \leq \alpha}$ starting at the root of t or Π is empty if such a collapsing tower does not exists. If the collapsing tower exists, then

$$\Pi = (\top, u_0)^\emptyset \xrightarrow{\emptyset} (r_0, \langle\rangle, u_0)^\emptyset \xrightarrow{\emptyset} (\top, u_1)^\emptyset \xrightarrow{\emptyset} (r_1, \langle\rangle, u_1)^\emptyset \xrightarrow{\emptyset} \ldots \xrightarrow{\emptyset} (\top, u_\alpha)^\emptyset \xrightarrow{\emptyset}$$

But then also Π_2 starts with the prefix $\Pi \cdot (\top, \pi)$ due to the uniqueness of the collapsing tower and the involved rules. In both cases, $\Pi_1 = \Pi_2$ follows immediately.

The second case is that Π_1 is infinite. Then there is an infinite collapsing tower $(u_i)_{i<\omega}$ starting at the root of t. Hence,

$$\Pi_1 = (\top, u_0)^\emptyset \xrightarrow{\emptyset} (r_0, \langle\rangle, u_0)^\emptyset \xrightarrow{\emptyset} (\top, u_1)^\emptyset \xrightarrow{\emptyset} (r_1, \langle\rangle, u_1)^\emptyset \xrightarrow{\emptyset} \ldots$$

$\Pi_1 = \Pi_2$ follows from the uniqueness of the infinite collapsing tower.

At first glance one might additionally find a third case where $\Pi_1 = \Pi \cdot (\top, \pi)^\emptyset \xrightarrow{\emptyset} (r, \langle\rangle, \pi)^\perp$ with Π a prefix corresponding to a collapsing tower as in the first case. However, this is not possible as it would require the occurrence of $\perp$ in a rule.

Let $\mathsf{tr}_t(\Pi_1) = f$. Then there are two cases: Either $\Pi_1 = \Pi \cdot (\top, \pi)^f$ or $\Pi_1 = \Pi \cdot (\top, \pi)^\emptyset \xrightarrow{\emptyset} (r, \langle\rangle, \pi)^f$, where the prefix Π corresponds to a finite maximal collapsing tower $(u_i)_{i \leq \alpha}$ starting at the root of t or Π is empty if such a collapsing tower does not exists. The argument is analogous to the argument employed for the first case of the induction base above.

Finally, we consider the induction step. Hence, there are the two cases: Either $\mathsf{tr}_t(\Pi_1) = T \cdot \langle i \rangle$ or $\mathsf{tr}_t(\Pi_1) = T \cdot \langle i, f \rangle$. For both cases, the induction hypothesis can be invoked by taking two prefixes Π_1' and Π_2' of Π_1 and Π_2, respectively, which both have the trace T and, therefore, are equal according to the induction hypothesis. The argument that the remaining suffixes of Π_1 and Π_2 are equal is then analogous to the argument for two base cases. □

As mentioned above, the traces of paths contain all information necessary to define a term which we will later identify to be the final term of the corresponding complete development. The following definition explains how such a term, called a *matching term*, is determined:

Definition 5.26 (matching term)**.** Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, and U a set of pairwise non-conflicting redex occurrences in t.

(i) The *position* of a trace $T \in \mathcal{T}r(t, U, \mathcal{R})$, denoted $\mathsf{pos}(T)$, is the subsequence of T containing only the edge labels. The set of all positions of traces in $\mathcal{T}r(t, U, \mathcal{R})$ is denoted $\mathcal{PT}r(t, U, \mathcal{R})$.

(ii) The *symbol* of a trace $T \in \mathcal{T}r(t, U, \mathcal{R})$, denoted $\mathsf{sym}_t(T)$, is f if T ends in a node label f, and is $\perp$ otherwise, i.e. whenever T is empty or ends in an edge label.

(iii) A term t' is said to *match* $\mathcal{T}r(t, U, \mathcal{R})$ if both $\mathcal{P}(t') = \mathcal{PT}r(t, U, \mathcal{R})$ and $t'(\mathsf{pos}(T)) = \mathsf{sym}_t(T)$ for all $T \in \mathcal{T}r(t, U, \mathcal{R})$.

Returning to the definition of paths, one can see that the label of a node is the symbol of the "current" position in a term. Similarly, the label of an edge says which edge in the term tree was taken at that point in the construction of the path. Hence, by projecting to the edge labels, we obtain the "history" of the path, i.e. the position. In the same way we obtain the symbol of that node by taking the label of the last node of the path, provided the corresponding path ends in a non-$\bot$-labelled node. In the other case that the trace does not end in a node label, the corresponding path either ends in a node labelled $\bot$ or is infinite. As we will see, infinite paths with finite traces correspond to infinite collapsing towers, which in turn yield volatile positions within the complete development. Eventually, these volatile positions will also give rise to $\bot$ subterms.

The following lemma shows that there is also a one-to-one correspondence between the traces in $\mathcal{T}r(t, U, \mathcal{R})$ and their positions in $\mathcal{PT}r(t, U, \mathcal{R})$:

Lemma 5.27 ($\mathsf{pos}(\cdot)$ is a bijection)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$ and U a set of redex occurrences in t. $\mathsf{pos}(\cdot)$ is a bijection from $\mathcal{T}r(t, U, \mathcal{R})$ to $\mathcal{PT}r(t, U, \mathcal{R})$.*

Proof. An argument similar to the one for Lemma 5.25 can be given in order to show that the composition $\mathsf{pos}(\cdot) \circ \mathsf{tr}_t(\cdot)$ is a bijection. Together with the bijectivity of $\mathsf{tr}_s(\cdot)$, according to Lemma 5.25, this yields the bijectivity of $\mathsf{pos}(\cdot)$. □

Having this lemma, the following proposition is an easy consequence of the definition of matching terms. It shows that matching terms do always exists and are uniquely determined:

Proposition 5.28 (unique matching term)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. Then there is a unique term, denoted $\mathcal{F}(t, U, \mathcal{R})$, that matches $\mathcal{T}r(t, U, \mathcal{R})$.*

Proof. Define the mapping $\varphi\colon \mathcal{PT}r(t, U, \mathcal{R}) \to \Sigma_\bot \cup \mathcal{V}$ by setting $\varphi(\mathsf{pos}(T)) = \mathsf{sym}_t(T)$ for each trace $T \in \mathcal{T}r(t, U, \mathcal{R})$. By Lemma 5.27, φ is well-defined. Moreover, it is easy to see from the definition of paths, that $\mathcal{PT}r(t, U, \mathcal{R})$ is closed under prefixes and that φ respects the arity of the symbols, i.e. $\pi \cdot i \in \mathcal{PT}r(t, U, \mathcal{R})$ iff $0 \leq i < \mathsf{ar}(\varphi(\pi))$. Hence, φ uniquely determines a term s with $s(\pi) = \varphi(\pi)$ for all $\pi \in \mathcal{PT}r(t, U, \mathcal{R})$. By construction, s matches $\mathcal{T}r(t, U, \mathcal{R})$. Moreover, any other term s' matching $\mathcal{T}r(t, U, \mathcal{R})$ must satisfy $s'(\pi) = \varphi(\pi)$ for all $\pi \in \mathcal{PT}r(t, U, \mathcal{R})$ and is therefore equal to s. □

It is also obvious that the matching term of a term t w.r.t. an empty set of redex occurrences is the term t itself.

Lemma 5.29 (matching term w.r.t. empty redex set)**.** *For any TRS $\mathcal{R}$ and any partial term t in $\mathcal{R}$, it holds that $\mathcal{F}(t, \emptyset, \mathcal{R}) = t$.*

Proof. Straightforward. □

Remark 5.30. Now it only remains to be shown that the matching term stays invariant during a development, i.e. that, for each development $S\colon t \xrightarrow{p} t'$ of U, the matching terms $\mathcal{F}(t, U, \mathcal{R})$ and $\mathcal{F}(t', U/\!/S, \mathcal{R})$ coincide. Since the matching term $\mathcal{F}(t, U, \mathcal{R})$ only depends on the set $\mathcal{T}r(t, U, \mathcal{R})$ of traces, it is sufficient to

show that $\mathcal{T}r(t, U, \mathcal{R})$ and $\mathcal{T}r(t', U//S, \mathcal{R})$ coincide. The key observation is that in each step $s \to s'$ in a development the paths in s' differ from the paths in s only in that they might omit some jumps. This can be seen in Figure 5a: In a step $s \to s'$ of a development, (some residual of) some redex occurrence in U is contracted. In the picture this corresponds to removing the pattern, say l_1, of the redex and replacing it by the corresponding right-hand side r_1 of the rule. One can see that, except for the jump to and from the right-hand side r_1 the path remains the same.

In order to establish the above observation formally, we need a means to simulate reduction steps in a development directly as an operation on paths. The following definition provides a tool for this.

Definition 5.31 (position and prefix of a path)**.** Let $\mathcal{R}$ be a left-linear TRS, t a partial term in $\mathcal{R}$, U a set of pairwise non-conflicting redex occurrences in t, and $\Pi \in \mathcal{P}(t, U, \mathcal{R})$.

(i) Π is said to *contain* a position $\pi \in \mathcal{P}(t)$ if it contains the node $(\top, \pi)$.

(ii) For each $u \in U$, the *prefix* of Π by u, denoted $\Pi^{(u)}$, is defined as Π whenever Π does not contain u and otherwise as the unique prefix of Π that ends in $(\top, \pi)$.

Remark 5.32. It is obvious from the definition that each prefix $\Pi^{(u)}$ of a path $\Pi \in \mathcal{P}(t, U, \mathcal{R})$ by an occurrence u is the maximal prefix of Π, that does not contain positions that are proper extensions of u. Equivalently, $\Pi^{(u)}$ is the maximal prefix of Π that only contains prefixes of u (including u itself).

The following lemma is the key step towards proving the invariance of matching terms in developments. It formalises the observation described in Remark 5.30.

Lemma 5.33 (preservation of traces)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, U a set of redex occurrences in t, and $S\colon t \overset{p}{\twoheadrightarrow} t'$ a development of U in t. There is a surjective mapping $\vartheta_S\colon \mathcal{P}(t, U, \mathcal{R}) \to \mathcal{P}(t', U//S, \mathcal{R})$ such that $\mathsf{tr}_t(\Pi) = \mathsf{tr}_{t'}(\vartheta_S(\Pi))$ for all $\Pi \in \mathcal{P}(t, U, \mathcal{R})$.*

Proof. Let $S = (t_\iota \to_{\pi_\iota, c_\iota} t_{\iota+1})_{\iota<\alpha}$. We prove the statement by an induction on α.

If $\alpha = 0$, then the statement is trivially true.

Suppose that α is a successor ordinal $\beta + 1$. Let $T\colon t_0 \overset{p}{\twoheadrightarrow}^{\beta} t_\beta$ be the prefix of S of length β and $\varphi_\beta\colon t_\beta \to_{\pi_\beta} t_\alpha$ the last step of S, i.e. $S = T \cdot \langle \varphi_\beta \rangle$. By the induction hypothesis, there is a surjective mapping $\vartheta_T\colon \mathcal{P}(t, U, \mathcal{R}) \to \mathcal{P}(t_\beta, U', \mathcal{R})$, with $U' = U//T$ and $\mathsf{tr}_t(\Pi) = \mathsf{tr}_{t_\beta}(\vartheta_T(\Pi))$ for all $\Pi \in \mathcal{P}(t, U, \mathcal{R})$. By a careful case analysis (as done in [20]), one can show that there is a surjective mapping $\vartheta\colon \mathcal{P}(t_\beta, U', \mathcal{R}) \to \mathcal{P}(t_\alpha, U'', \mathcal{R})$, with $U'' = U'//\langle \varphi_\beta \rangle = U//S$ and $\mathsf{tr}_{t_\beta}(\Pi) = \mathsf{tr}_{t_\alpha}(\vartheta(\Pi))$ for all $\Pi \in \mathcal{P}(t_\beta, U', \mathcal{R})$. Hence, the composition $\vartheta_S = \vartheta \circ \vartheta_T$ is a surjective mapping from $\mathcal{P}(t, U, \mathcal{R})$ to $\mathcal{P}(t', U//S, \mathcal{R})$ and satisfies $\mathsf{tr}_t(\Pi) = \mathsf{tr}_{t'}(\vartheta_S(\Pi))$ for all $\Pi \in \mathcal{P}(t, U, \mathcal{R})$.

Let α be a limit ordinal. By induction hypothesis, there is a surjective mapping $\vartheta_{S|_\iota}$ for each proper prefix $S|_\iota$ of S satisfying $\mathsf{tr}_{t_0}(\Pi) = \mathsf{tr}_{t_\iota}(\vartheta_{s|_\iota}(\Pi))$ for all $\Pi \in$

$\mathcal{P}(t, U, \mathcal{R})$. Let $\Pi \in \mathcal{P}(t, U, \mathcal{R})$ and $\Pi_\iota = \vartheta_{S|_\iota}(\Pi)$ for each $\iota < \alpha$. We define $\vartheta_S(\Pi)$ as follows:

$$\vartheta_S(\Pi) = \liminf_{\iota \to \alpha} \Pi_\iota^{(\pi_\iota)}$$

At first we have to show that ϑ_S is well-defined, i.e. that $\liminf_{\iota\to\alpha} \Pi_\iota^{(\pi_\iota)}$ is indeed a path in $\mathcal{P}(t', U//S, \mathcal{R})$, and that it preserves traces. There are two cases to be considered: If there is an outermost-volatile position π in S that is contained in Π_ι whenever $\pi_\iota = \pi$, then there is some $\beta < \alpha$ with $\pi_\iota \not< \pi$ for all $\beta \leq \iota < \alpha$. Hence, $\vartheta_S(\Pi) = \Pi_\beta^{(\pi)}$. By Lemma 3.11 and Lemma 3.14, we have that $\Pi_\beta^{(\pi)} \in \mathcal{P}(t', U//S, \mathcal{R})$, in particular because $t'(\pi) = \bot$. Since the suffix Π' with $\Pi_\beta = \Pi_\beta^{(\pi)} \cdot \Pi'$ follows an infinite collapsing tower and is therefore entirely unlabelled, it cannot contribute to the trace of Π_β. Consequently,

$$\mathsf{tr}_t(\Pi) \overset{IH}{=} \mathsf{tr}_{t_\beta}(\Pi_\beta) = \mathsf{tr}_{t'}(\Pi_\beta^{(\pi)}) = \mathsf{tr}_{t'}(\vartheta_S(\Pi)).$$

If, on the other hand, there is no such outermost-volatile position, then either the sequence $(\Pi_\iota^{(\pi_\iota)})_{\iota<\alpha}$ becomes stable at some point or the sequence $(\bigsqcap_{\iota<\gamma} \Pi_\iota^{(\pi_\iota)})_{\gamma<\alpha}$ grows monotonically towards the infinite path $\vartheta_S(\Pi)$. In either case, both well-definedness and preservation of traces follows easily from the induction hypothesis.

Lastly, we show the surjectivity of ϑ_S. To this end, assume some $\Pi \in \mathcal{P}(t', U//S, \mathcal{R})$. We show the existence of a path $\overline{\Pi} \in \mathcal{P}(t, U, \mathcal{R})$ with $\vartheta_S(\overline{\Pi}) = \Pi$ by distinguishing three cases:

(a) Π ends in a redex node (r, π, u). Hence, $u \in U//S$. According to Lemma 5.3, this means that there is some $\beta < \alpha$ such that

$$\pi_\iota \not\leq u \text{ for all } \beta \leq \iota < \alpha. \tag{1}$$

Consequently, all terms in $\{t_\iota \mid \beta \leq \iota < \alpha\}$ coincide in all prefixes of u, and each $v \in U//S$ with $v \leq u$ is in $U//S|_\iota$ for all $\beta \leq \iota < \alpha$. Hence, for all $\beta \leq \gamma < \alpha$ we have $\Pi \in \mathcal{P}(t_\gamma, U//S|_\gamma, \mathcal{R})$ with $\mathsf{tr}_{t'}(\Pi) = \mathsf{tr}_{t_\gamma}(\Pi)$. By induction hypothesis there is for each $\beta \leq \gamma < \alpha$ some $\Pi_\gamma \in \mathcal{P}(t, U, \mathcal{R})$ that is mapped to $\Pi \in \mathcal{P}(t_\gamma, U//S|_\gamma, \mathcal{R})$ by $\vartheta_{S|_\gamma}$ with $\mathsf{tr}_t(\Pi_\gamma) = \mathsf{tr}_{t_\gamma}(\Pi)$. Hence, $\mathsf{tr}_t(\Pi_\gamma) = \mathsf{tr}_{t'}(\Pi)$ which means that all paths Π_γ, with $\beta \leq \gamma < \alpha$, have the same trace in t and are therefore equal according to Lemma 5.25. Let us call this path $\overline{\Pi}$. That is, $\vartheta_{S|_\gamma}(\overline{\Pi}) = \Pi$ for all $\beta \leq \gamma < \alpha$. Since $\pi_\gamma \not\leq u$, we also have $(\vartheta_{S|_\gamma}(\overline{\Pi}))^{(\pi_\gamma)} = \Pi$. Consequently, $\vartheta_S(\overline{\Pi}) = \Pi$.

(b) Π ends in a term node $(\top, \pi)$. Let $f = t'(\pi)$. If $f \neq \bot$, then we can apply Lemma 3.11 to obtain some $\beta < \alpha$ such that $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. Then we can reason as in case (a) starting from (1). If $f = \bot$, then we have to distinguish two cases according to Lemma 3.14: If there is some $\beta < \alpha$ with $t_\beta(\pi) = \bot$ and $\pi_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$, then we can again employ the same argument as for case (a) starting from (1). Otherwise, i.e. if π is an outermost-volatile position in S, then we have some $\beta < \alpha$ such that $\pi_\iota \not< \pi$ for all $\beta \leq \iota < \alpha$ and such that

$$\text{for each } \beta \leq \gamma < \alpha \text{ there is some } \gamma \leq \gamma' < \alpha \text{ with } \pi'_\gamma = \pi. \tag{2}$$

Hence, we have for each $\beta \leq \gamma < \alpha$ some $\Pi_\gamma \in \mathcal{P}(t_\gamma, U//S|_\gamma, \mathcal{R})$ and an infinite collapsing tower $(u_i)_{i<\omega}$ in $U//S|_\gamma$ with $u_0 = \pi$ such that Π_γ is of the form

$$\Pi \cdot \xrightarrow{\emptyset} (r_0, \langle\rangle, u_0)^\emptyset \xrightarrow{\emptyset} (\top, u_1)^\emptyset \xrightarrow{\emptyset} (r_1, \langle\rangle, u_1)^\emptyset \xrightarrow{\emptyset} \dots$$

Therefore, $\mathsf{tr}_{t_\gamma}(\Pi_\gamma) = \mathsf{tr}_{t'}(\Pi)$. By induction hypothesis there is some $\overline{\Pi}_\gamma \in \mathcal{P}(t, T, \mathcal{R})$ with $\vartheta_{S|_\gamma}(\overline{\Pi}_\gamma) = \Pi_\gamma$ and $\mathsf{tr}_t(\overline{\Pi}_\gamma) = \mathsf{tr}_{t_\gamma}(\Pi_\gamma)$. Hence, $\mathsf{tr}_t(\overline{\Pi}_\gamma) = \mathsf{tr}_{t'}(\Pi)$, i.e. all $\overline{\Pi}_\gamma$ have the same trace in t and are therefore equal according to Lemma 5.25. Let us call this path $\overline{\Pi}$. Since $(\vartheta_{S|_\gamma}(\overline{\Pi}))^{(\pi)} = \Pi_\gamma^{(\pi)} = \Pi$ we can use (2) to obtain that $\vartheta_S(\overline{\Pi}) = \Pi$.

(c) Π is infinite. Hence, Π is of the form

$$\Pi' \cdot (\top, u_0)^\emptyset \xrightarrow{\emptyset} (r_0, \langle\rangle, u_0)^\emptyset \xrightarrow{\emptyset} (\top, u_1)^\emptyset \xrightarrow{\emptyset} (r_1, \langle\rangle, u_1)^\emptyset \xrightarrow{\emptyset} \dots$$

with $(u_i)_{i<\omega}$ an infinite collapsing tower in $U//S$. Consequently, according to Lemma 5.3, for each $u_i \in U//S$ there is some $\beta_i < \alpha$ such that

$$u_i \in U//S|_\gamma \text{ and } \pi_\gamma \not\leq u_\gamma \text{ for all } \beta_i \leq \gamma < \alpha. \tag{3}$$

Since $(u_i)_{i<\omega}$ is a chain (w.r.t. the prefix order), we can assume w.l.o.g. that $(\beta_i)_{i<\omega}$ is a chain as well. Following Remark 5.7, we obtain for each $u_i \in U//S$ its ancestor $v_i \in U$ with $v_i//S = u_i$. Let $\overline{\Pi}$ be the unique path in $\mathcal{P}(t, U, \mathcal{R})$ that contains each v_i and for each $j < \omega$ let Π_j be the unique path in $\mathcal{P}(t_{\beta_j}, U//S|_{\beta_j}, \mathcal{R})$ containing each $v_i//S|_{\beta_j}$. Clearly, $\vartheta_{S|_{\beta_j}}(\overline{\Pi}) = \Pi_j$. Note that we have for each $j < \omega$ that all paths $\vartheta_{S|_\iota}(\overline{\Pi})$ with $\beta_j \leq \iota < \alpha$ coincide in their prefix by u_j, which is a prefix of Π. Since additionally $(u_i)_{i<\omega}$ is a strict chain and because of (3), we can conclude that $\vartheta_S(\overline{\Pi}) = \Pi$.

□

The above lemma effectively establishes the invariance of matching terms during a development. Together with Lemma 5.29 this implies the uniqueness of final terms of complete developments of the same redex occurrences. As a corollary from this, we obtain that descendants are also unique among all complete developments:

Proposition 5.34 (final term and descendants of complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. Then the following holds:*

(i) Each complete development of U in t strongly p-converges to $\mathcal{F}(t, U, \mathcal{R})$.

(ii) For each set $V \subseteq \mathcal{P}_\not\perp(t)$ and two complete developments S and T of U in t, respectively, it holds that $V//S = V//T$.

Proof. (i) Let $S\colon t \xrightarrow{p}\!\!\!\!\!\rightarrow_U t'$ be a complete development of U in t strongly p-converging to t'. By Lemma 5.33, there is a surjective mapping $\vartheta\colon \mathcal{P}(t, U, \mathcal{R}) \to \mathcal{P}(t', U', \mathcal{R})$ with $\mathsf{tr}_t(\Pi) = \mathsf{tr}_{t'}(\vartheta(\Pi))$ for all $\Pi \in \mathcal{P}(t, U, \mathcal{R})$, where $U' = U//S$. Hence, it holds that $\mathcal{T}r(t, U, \mathcal{R}) = \mathcal{T}r(t', U', \mathcal{R})$ and, consequently, $\mathcal{F}(t, U, \mathcal{R}) =$

$\mathcal{F}(t', U', \mathcal{R})$. Since S is a complete development of U in t, we have that $U' = \emptyset$ which implies, according to Lemma 5.29, that $\mathcal{F}(t', U', \mathcal{R}) = t'$. Therefore, $\mathcal{F}(t, U, \mathcal{R}) = t'$.

(ii) Let $t' = t^{(V)}$. By Proposition 5.13, both reductions S and T can be uniquely lifted to reductions S' and T' in $\mathcal{R}^l$, respectively, such that $V//S$ and $V//T$ are determined by the final term of S' and T', respectively. It is easy to see that also $\mathcal{R}^l$ is an orthogonal TRS and that S' and T' are complete developments of U in t'. Hence, we can invoke clause (i) of this proposition to conclude that the final terms of S' and T' coincide and that, therefore, also $V//S$ and $V//T$ coincide. □

By the above proposition, the descendants of a complete development of a particular set of redex occurrences are unique. Therefore, we adopt the notation $U//V$ for the descendants $U//S$ of U by some complete development S of V. According to Proposition 5.17 and Proposition 5.34, $U//V$ is well-defined for any orthogonal TRS.

Furthermore, Proposition 5.34 yields the following corollary establishing the diamond property of complete developments:

Corollary 5.35 (diamond property of complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS and $t \xrightarrow{p}\!\!\!\!\rightarrow_U t_1$ and $t \xrightarrow{p}\!\!\!\!\rightarrow_V t_2$ be two complete developments of U respectively V in t. Then t_1 and t_2 are joinable by complete developments $t_1 \xrightarrow{p}\!\!\!\!\rightarrow_{V//U} t'$ and $t_2 \xrightarrow{p}\!\!\!\!\rightarrow_{U//V} t'$.*

Proof. By Proposition 5.5, it holds that

$$(U \cup V)//U = U//U \cup V//U = V//U.$$

Let $S\colon t \xrightarrow{p}\!\!\!\!\rightarrow_U t_1$, $T\colon t \xrightarrow{p}\!\!\!\!\rightarrow_V t_2$, $S'\colon t_1 \xrightarrow{p}\!\!\!\!\rightarrow_{V//U} t'$ and $T'\colon t_2 \xrightarrow{p}\!\!\!\!\rightarrow_{U//V} t''$. By the equation above and Proposition 5.8, we have that $S{\cdot}S'\colon t \xrightarrow{p}\!\!\!\!\rightarrow_U t_1 \xrightarrow{p}\!\!\!\!\rightarrow_{V//U} t'$ is a complete development of $U \cup V$. Analogously, we obtain that $T \cdot T'\colon t \xrightarrow{p}\!\!\!\!\rightarrow_V t_2 \xrightarrow{p}\!\!\!\!\rightarrow_{U//V} t''$ is a complete development of $U \cup V$, too. According to Proposition 5.34, this implies that both $S \cdot S'$ and $T \cdot T'$ strongly p-converge in the same term, i.e. $t' = t''$. □

In the next section we shall make use of complete developments in order to obtain the Infinitary Strip Lemma for p-converging reductions and a limited form of infinitary confluence for orthogonal systems.

5.4 The Infinitary Strip Lemma

In this section we use the results we have obtained for complete developments in the previous two sections in order to establish that a complete development of a set of pairwise disjoint redex occurrences commutes with any strongly p-convergent reduction:

Proposition 5.36 (Infinitary Strip Lemma)**.** *Suppose $\mathcal{R}$ is an orthogonal TRS, $S\colon t_0 \xrightarrow{p}\!\!\!\!\rightarrow^{\alpha} t_\alpha$ is a strongly p-convergent reduction, and $t_0 \xrightarrow{p}\!\!\!\!\rightarrow_U s_0$ is a complete development of a set U of pairwise disjoint redex occurrences in t_0. Then t_α and s_0 are joinable by a reduction $S/T\colon s_0 \xrightarrow{p}\!\!\!\!\rightarrow s_\alpha$ and a complete development $T/S\colon t_\alpha \xrightarrow{p}\!\!\!\!\rightarrow_{U//S} s_\alpha$.*

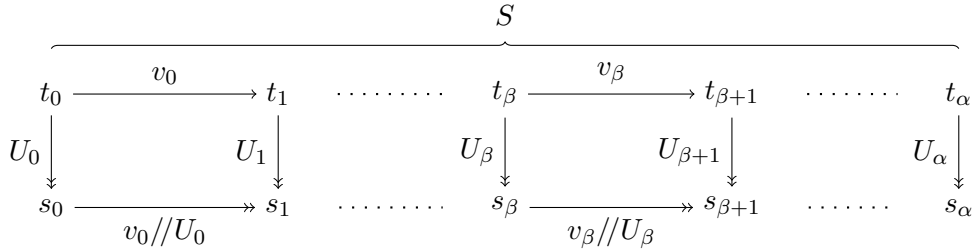


Figure 6: The Infinitary Strip Lemma.

Proof. We prove this statement by constructing the diagram shown in Figure 6. The 'U_ι's in the diagram are sets of redex occurrences: $U_\iota = U//S|_\iota$ for all $0 \leq \iota \leq \alpha$. In particular, $U_0 = U$. All arrows in the diagram represent complete developments of the indicated sets of redex occurrences. Particularly, in each ι-th step of S the redex at v_ι is contracted. We will construct the diagram by an induction on α.

If $\alpha = 0$, then the diagram is trivial. If α is a successor ordinal $\beta + 1$, then we can take the diagram for the prefix $S|_\beta$, which exists by induction hypothesis, and extend it to a diagram for S. The existence of the additional square that completes the diagram for S is affirmed by Corollary 5.35 since $U_{\beta+1} = U_\beta//v_\beta$.

Let α be a limit ordinal. Moreover, let s'_α be the uniquely determined final term of a complete development of U_α in t_α. By induction hypothesis, the diagram exists for each proper prefix of S. Let $T_\iota\colon\ s_0 \xrightarrow{p}\!\!\!\!\rightarrow s_\iota$ denote the reduction at the bottom of the diagram for the reduction $S|_\iota$ for each $\iota < \alpha$. The set of all T_ι is directed. Hence, $T = \bigsqcup_{\iota<\alpha} T_\iota$ exists. Since $T_\iota < T$ for each $\iota < \alpha$, the diagram for S with $T\colon\ s_0 \xrightarrow{p}\!\!\!\!\rightarrow s_\alpha$ at the bottom satisfies almost all required properties. Only the equality of s_α and s'_α remains to be shown.

Note that, by Proposition 5.9, the redex occurrences in U_α are pairwise disjoint. Let $\pi \in U_\alpha$. By Lemma 5.3 and the definition of descendants, there is some $\beta < \alpha$ such that $\pi \in U_\iota$ and $v_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. Hence, for all $\pi' \in v_\iota//U_\iota$ with $\beta \leq \iota < \alpha$, we also have $\pi' \not\leq \pi$. That is, in the remaining reductions $t_\beta \xrightarrow{p}\!\!\!\!\rightarrow t_\alpha$ and $t_\beta \xrightarrow{p}\!\!\!\!\rightarrow_{U_\beta} s_\beta \xrightarrow{p}\!\!\!\!\rightarrow s_\alpha$, no reduction takes place at a proper prefix of π. Hence, by Lemma 3.11, t_β coincides with t_α and s_α in all proper prefixes of π. Since in the reduction $t_\alpha \xrightarrow{p}\!\!\!\!\rightarrow_{U_\alpha} s'_\alpha$ also no reduction takes place at a proper prefix of π, we obtain that t_α and s'_α and, thus, also s_α and s'_α coincide in all proper prefixes of π.

Let $\rho\colon\ l \to r$ be the rule for the redex $t_\beta|_\pi$ and $C\langle,\dots,\rangle, D\langle,\dots,\rangle$ ground contexts such that $l = C\langle x_1,\dots,x_k\rangle$ and $r = D\langle x_{p(1)},\dots,x_{p(m)}\rangle$ for some pairwise distinct variables $x_1,\dots,x_k$ and an appropriate mapping $p\colon\ \{1,\dots,m\} \to \{1,\dots,k\}$. Moreover, let $t^\iota_1,\dots,t^\iota_k$ be terms such that $t_\iota = t_\iota[C\langle t^\iota_1,\dots,t^\iota_k\rangle]_\pi$ and $s_\iota = s_\iota[D\langle t^\iota_{p(1)},\dots,t^\iota_{p(m)}\rangle]_\pi$ for all $\beta \leq \iota \leq \alpha$. The argument in the previous paragraph justifies the assumption of these elements. From β onward, all horizontal reduction steps in the diagram take place within the contexts $t_\iota[\cdot]_\pi$ and $s_\iota[\cdot]_\pi$, respectively, or inside the terms t^ι_i, and all vertical reductions take place within the contexts $t_\iota[C\langle,\dots,\rangle]_\pi$ and $s_\iota[D\langle,\dots,\rangle]_\pi$, respectively. In particular, we have $t_\alpha = t_\alpha[C\langle t^\alpha_1,\dots,t^\alpha_k\rangle]_\pi$ and $s_\alpha = s_\alpha[D\langle t^\alpha_{p(1)},\dots,t^\alpha_{p(m)}\rangle]_\pi$. Let $t_\alpha \to_\pi t'_\alpha$. This reduction contracts the redex $C\langle t^\alpha_1,\dots,t^\alpha_k\rangle$ to the term $D\langle t^\alpha_{p(1)},\dots,t^\alpha_{p(m)}\rangle$ using

rule ρ. Note that a complete development $t_\alpha \overset{p}{\twoheadrightarrow}_{U_\alpha} s'_\alpha$ contracts, besides π, only redex occurrences disjoint with π. Hence, t'_α and s'_α coincide in all extensions of π. Since $t'_\alpha = t_\alpha[D\langle t^\alpha_{p(1)}, \ldots, t^\alpha_{p(k)}\rangle]_\pi$ (and $s_\alpha = s_\alpha[D\langle t^\alpha_{p(1)}, \ldots, t^\alpha_{p(m)}\rangle]_\pi$), we can conclude that s_α and s'_α coincide in all extensions of π.

Since the residual $\pi \in U_\alpha$ was chosen arbitrarily, the above holds for all elements in U_α. That is, s_α and s'_α coincide in all prefixes and all extensions of elements in U_α. It remains to be shown, that they also coincide in positions that are disjoint to all positions in U_α. To this end, we only need to show that t_α and s_α coincide in these positions since the complete development $t_\alpha \overset{p}{\twoheadrightarrow}_{U_\alpha} s'_\alpha$ keeps positions disjoint with all positions in U_α unchanged. Let π be such a position.

Suppose $t_\alpha(\pi) = f \neq \bot$. By Lemma 3.11, there is some $\beta < \alpha$ such that $t_\beta(\pi) = f$ and $v_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. Note that no prefix π' of π is in U_β since otherwise $\pi' \in U_\alpha$, by Lemma 5.3, which contradicts the assumption that π is disjoint to all positions in U_α. Hence, $s_\beta(\pi) = f$ and $\pi' \not\leq \pi$ for all $\pi' \in v_\iota // U_\iota$ and $\beta \leq \iota < \alpha$, which means that no reduction step in $s_\beta \overset{p}{\twoheadrightarrow} s_\alpha$ takes place at some prefix of π. Thus, we can conclude, according to Lemma 3.11, that $s_\alpha(\pi) = f$. Similarly, one can show that $s_\alpha(\pi) = f \neq \bot$ implies $t_\alpha(\pi) = f$.

Suppose $t_\alpha(\pi) = \bot$. Hence, according to Lemma 3.14, π is outermost-volatile in S or there is some $\beta < \alpha$ such that $t_\beta(\pi) = \bot$ and $v_\iota \not\leq \pi$ for all $\beta \leq \iota < \alpha$. For the latter case, we can argue as in the case for $t_\alpha(\pi) \neq \bot$ above. In the former case, π is outermost-volatile in T as well. Thus, by applying Lemma 3.14, we obtain that $s_\alpha(\pi) = \bot$. A similar argument can be employed for the reverse direction. □

The reduction S/T constructed in the proof above is called the *projection* of S by T. Likewise, the reduction T/S is called the *projection* of T by S. As a corollary we obtain the following semi-infinitary confluence result:

Corollary 5.37 (semi-infinitary confluence)**.** *In every orthogonal TRS, two reductions $t \overset{p}{\twoheadrightarrow} t_2$ and $t \to^* t_1$ can be joined by two reductions $t_2 \overset{p}{\twoheadrightarrow} t_3$ and $t_1 \overset{p}{\twoheadrightarrow} t_3$.*

Proof. This can be shown by an induction on the length of the reduction $t \to^* t_1$. If it is empty, the statement trivially holds. The induction step follows from Proposition 5.36. □

In the next section we shall, based on the Infinitary Strip Lemma, show that strong p-reachability coincides with Böhm-reachability, which then yields, amongst other things, full infinitary confluence of orthogonal systems.

6 Strong p-Convergence vs. Böhm-Convergence

In this section we shall show the core result of this paper: For orthogonal, left-finite TRSs, strong p-reachability and Böhm-reachability w.r.t. the set $\mathcal{RA}$ of root-active terms coincide. As corollaries of that, leveraging the properties of Böhm-convergence, we obtain both infinitary normalisation and infinitary confluence of orthogonal systems in the partial order model. Moreover, we will show that strong p-convergence also satisfies the compression property.

The central step of the proof of the equivalence of both models of infinitary rewriting is an alternative characterisation of root-active terms which is captured by the following definition:

Definition 6.1 (destructiveness, fragility)**.** Let $\mathcal{R}$ be a TRS.

(i) A reduction $S\colon t \overset{p}{\twoheadrightarrow} s$ is called *destructive* if $\langle\rangle$ is a volatile position in S.

(ii) A partial term t in $\mathcal{R}$ is called *fragile* if a destructive reduction starts in t.

Looking at the definition, fragility seems to be a more general concept than root-activeness: A term is fragile iff it admits a reduction in which infinitely often a redex at the root is contracted. For orthogonal TRSs, root-active terms are characterised in almost the same way. The difference is that only total terms are considered and that the stipulated reduction contracting infinitely many root redexes has to be of length ω. However, we shall show the set of total fragile terms to be equal to the set of root-active terms by establishing a compression lemma for destructive reductions.

Using Lemma 3.14 we can immediately derive the following alternative characterisations:

Fact 6.2 (destructiveness, fragility)**.** *Let $\mathcal{R}$ be a TRS.*

(i) A reduction $S\colon s \overset{p}{\twoheadrightarrow} t$ is destructive iff S is open and $t = \bot$

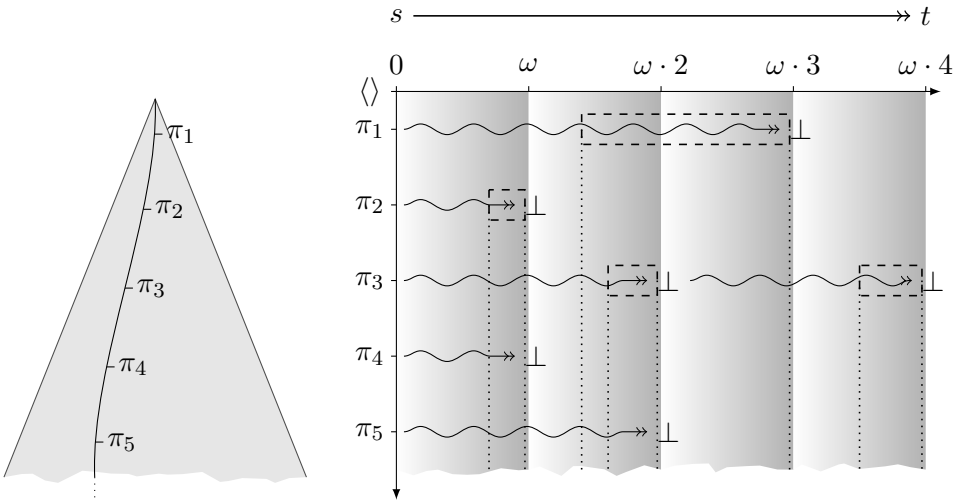
(ii) A partial term t in $\mathcal{R}$ is fragile iff there is an open strongly p-convergent reduction $t \overset{p}{\twoheadrightarrow} \bot$.

One has to keep in mind, however, that a closed reduction to $\bot$ is not destructive. Such a notion of destructiveness would include the empty reduction from $\bot$ to $\bot$, and reductions that end with the contraction of a collapsing redex as, for example, in the single step reduction $f(\bot) \to \bot$ induced by the rule $f(x) \to x$. Such reductions do not "produce" the term $\bot$. They are merely capable of "moving" an already existent subterm $\bot$ by a collapsing rule. In this sense, fragile terms are, according to Lemma 3.15, the only terms which can produce the term $\bot$. This is the key observation for studying the relation between strong p-convergence and Böhm-convergence.

In order to show that strong p-reachability and Böhm-reachability w.r.t. $\mathcal{RA}$ coincide we will proceed as follows: At first we will show that strong p-reachability implies Böhm-reachability w.r.t. the set of total fragile terms, i.e. the fragile terms in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. From this we will derive a compression lemma for destructive reductions. We will then use this to show that the set $\mathcal{RA}$ of root-active terms coincides with the set of total fragile terms. From this we conclude that strong p-reachability implies Böhm-reachability w.r.t. $\mathcal{RA}$. Finally, we then show the other direction of the equality.

6.1 From Strong p-Convergence to Böhm-Convergence

For the first step we have to transform a strongly p-converging reduction in to a Böhm-converging reduction w.r.t. the set of total fragile terms, i.e. a strongly m-converging reduction w.r.t. the corresponding Böhm extension $\mathcal{B}$. Recall that,



(a) Nested volatile positions. (b) Replacing nested destructive reductions by $\to_\bot$ steps.

Figure 7: Turning a p-converging reduction into a Böhm-converging reduction.

by Theorem 4.12, the only difference between strongly p-converging reductions and strongly m-converging reductions is the ability of the former to produce $\bot$ subterms. This happens, according to Lemma 3.14, precisely at volatile positions.

We can, therefore, proceed as follows: Given a strongly p-converging reduction we construct a Böhm-converging reduction by removing reduction steps which cause the volatility of a position in some open prefix of the reduction and then replacing them by a *single* $\to_\bot$-step.

The intuition of this construction is illustrated in Figure 7. It shows a strongly p-converging reduction of length $\omega \cdot 4$ from s to t. In order to maintain readability, we restrict the attention to a particular branch of the term (tree) as indicated in Figure 7a. The picture shows five positions which are volatile in some open prefix of the reduction. We assume that they are the only volatile positions at least in the considered branch. Note that the positions do not need to occur in all of the terms in the reduction. They might disappear and reappear repeatedly. Each of them, however, appears in infinitely many terms in the reduction, as, by definition of volatility, infinitely many steps take place at each of these positions. In Figure 7b, the prefixes of the reduction that contain a volatile position are indicated by a waved rewrite arrow pointing to a $\bot$. The level of an arrow indicates the position which is volatile. A prefix might have multiple volatile positions. For example, both π_2 and π_4 are volatile in the prefix of length ω. But a position might also be volatile for several prefixes. For instance, π_3 is volatile in the prefix of length $\omega \cdot 2$ and the prefix of length $\omega \cdot 4$.

By Lemma 3.14, outermost-volatile positions are responsible for the generation of $\bot$ subterms. By their nature, at some point there are no reductions taking place above outermost-volatile positions. The suffix where this is the case is a *nested* destructive reduction. The subterm where this suffix starts is, therefore, a fragile term and we can replace this suffix with a *single* $\to_\bot$-step. The segments which are replaced in this way are highlighted by dashed boxes in Figure 7b. As

indicated by the dotted lines, this then also includes reduction steps which occur below the outermost-volatile positions. Therefore, also volatile positions which are not outermost are removed as well. Eventually, we obtain a reduction without volatile positions, which is, by Lemma 3.15, a strongly m-converging reduction in the Böhm extension, i.e. a Böhm-converging reduction in the original system:

Proposition 6.3 (strong p-reachability implies Böhm-reachability)**.** *Let $\mathcal{R}$ be a TRS, $\mathcal{U}$ the set of fragile terms in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$, and $\mathcal{B}$ the Böhm extension of $\mathcal{R}$ w.r.t. $\mathcal{U}$. Then, for each strongly p-convergent reduction $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$, there is a Böhm-convergent reduction $s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$.*

Proof. Assume that there is a reduction $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ in $\mathcal{R}$ that strongly p-converges to t_α. We will construct a strongly m-convergent reduction $T\colon\ t_0 \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t_\alpha$ in $\mathcal{B}$ by removing reduction steps in S that take place at or below outermost-volatile positions of some prefix of S and replace them by $\to_\bot$-steps.

Let π be an outermost-volatile position of some prefix $S|_\lambda$. Then there is some ordinal $\beta < \lambda$ such that no reduction step between β and λ in S takes place strictly above π, i.e. $\pi_\iota \not< \pi$ for all $\beta \leq \iota < \lambda$. Such an ordinal β must exist since otherwise π would not be an outermost-volatile position in $S|_\lambda$. Hence, we can construct a destructive reduction $S'\colon\ t_\beta|_\pi \overset{p}{\twoheadrightarrow} \bot$ by taking the subsequence of the segment $S|_{[\beta,\lambda)}$ that contains the reduction steps at π or below. Note that $t_\beta|_\pi$ might still contain the symbol $\bot$. Since $\bot$ is not relevant for the applicability of rules in $\mathcal{R}$, each of the $\bot$ symbols in $t_\beta|_\pi$ can be safely replaced by arbitrary total terms, in particular by terms in $\mathcal{U}$. Let r be a term that is obtained in this way. Then there is a destructive reduction $S''\colon\ r \overset{p}{\twoheadrightarrow} \bot$ that applies the same rules at the same positions as in S'. Hence, $r \in \mathcal{U}$. By construction, r is a $\bot,\mathcal{U}$-instance of $t_\beta|_\pi$ which means that $t_\beta|_\pi \in \mathcal{U}_\bot$. Additionally, $t_\beta|_\pi \neq \bot$ since there is a non-empty reduction $S'\colon\ t_\beta|_\pi \overset{p}{\twoheadrightarrow} \bot$ starting in $t_\beta|_\pi$. Consequently, there is a rule $t_\beta|_\pi \to \bot$ in $\mathcal{B}$. Let T' be the reduction that is obtained from $S|_\lambda$ by replacing the β-th step, which we can assume w.l.o.g. to take place at π, by a step with the rule $t_\beta|_\pi \to \bot$ at the same position π and removing all reduction steps φ_ι taking place at π or below for all $\beta < \iota < \lambda$. Let t' be the term that the reduction T' strongly p-converges to. t_λ and t' can only differ at position π or below. However, by construction, we have $t'(\pi) = \bot$ and, by Lemma 3.14, $t_\lambda(\pi) = \bot$. Consequently, $t' = t_\lambda$.

This construction can be performed for all prefixes of S and their respective outermost-volatile positions. Thereby, we obtain a strongly p-converging reduction $T\colon\ t_0 \overset{p}{\twoheadrightarrow}_{\mathcal{B}} t_\alpha$ for which no prefix has a volatile position. By Lemma 3.15, T is a total reduction. Note that $\mathcal{B}$ is a TRS over the extended signature $\Sigma' = \Sigma \uplus \{\bot\}$, i.e. terms containing $\bot$ are considered total. Hence, by Theorem 4.12, $T\colon\ t_0 \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t_\alpha$. □

6.2 From Böhm-convergence to Strong p-Convergence

Next, we establish a compression lemma for destructive reductions, i.e. that each destructive reduction can be compressed to length ω. Before we continue with this, we need to mention the following lemma from Kennaway et al. [16]:

Lemma 6.4 (postponement of $\to_\bot$-steps)**.** *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $\mathcal{B}$ some Böhm extension of $\mathcal{R}$. Then $s \mathrel{\overset{m}{\twoheadrightarrow}_\mathcal{B}} t$ implies $s \mathrel{\overset{m}{\twoheadrightarrow}_\mathcal{R}} s' \mathrel{\overset{m}{\twoheadrightarrow}_\bot} t$ for some term s'.*[2]

In the next proposition we show that, excluding $\bot$ subterms, the final term of a strongly p-converging reduction can be approximated arbitrarily well by a finite reduction. This corresponds to Corollary 2.5 which establishes finite approximations for strongly m-convergent reductions.

Proposition 6.5 (finite approximation)**.** *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $s \mathrel{\overset{p}{\twoheadrightarrow}_\mathcal{R}} t$. Then, for each finite set $P \subseteq \mathcal{P}_\not\perp(t)$, there is a reduction $s \to^*_\mathcal{R} t'$ such that t and t' coincide in P.*

Proof. Assume that $s \mathrel{\overset{p}{\twoheadrightarrow}_\mathcal{R}} t$. Then, by Proposition 6.3, there is a reduction $s \mathrel{\overset{m}{\twoheadrightarrow}_\mathcal{B}} t$, where $\mathcal{B}$ is the Böhm extension of $\mathcal{R}$ w.r.t. the set of total, fragile terms of $\mathcal{R}$. By Lemma 6.4, there is a reduction $s \mathrel{\overset{m}{\twoheadrightarrow}_\mathcal{R}} s' \mathrel{\overset{m}{\twoheadrightarrow}_\bot} t$. Clearly, s' and t coincide in $\mathcal{P}_\not\perp(t)$. Let $d = \max\{|\pi| \mid \pi \in P\}$. Since P is finite, d is well-defined. By Corollary 2.5, there is a reduction $s \to^*_\mathcal{R} t'$ such that t' and s' coincide up to depth d and, thus, in particular they coincide in P. Consequently, since s' and t coincide in $\mathcal{P}_\not\perp(t) \supseteq P$, t and t' coincide in P, too. □

In order to establish a compression lemma for destructive reductions we need that fragile terms are preserved by finite reductions. We can obtain this from the following more general lemma showing that destructive reductions are preserved by forming projections as constructed in the Infinitary Strip Lemma:

Lemma 6.6 (preservation of destructive reductions by projections)**.** *Let $\mathcal{R}$ be an orthogonal TRS, $S\colon t_0 \mathrel{\overset{p}{\twoheadrightarrow}} t_\alpha$ a destructive reduction, and $T\colon t_0 \mathrel{\overset{p}{\twoheadrightarrow}_U} s_0$ a complete development of a set U of pairwise disjoint redex occurrences. Then the projection $S/T\colon s_0 \mathrel{\overset{p}{\twoheadrightarrow}} s_\alpha$ is also destructive.*

Proof. We consider the situation depicted in Figure 6. Since $S\colon t_0 \mathrel{\overset{p}{\twoheadrightarrow}} t_\alpha$ is destructive, we have, for each $\beta < \alpha$, some $\beta \leq \gamma < \alpha$ such that $v_\gamma = \langle\rangle$. If $v_\gamma = \langle\rangle$, then also $\langle\rangle \in v_\gamma // U_\gamma$ unless $\langle\rangle \in U_\gamma$. As by Proposition 5.9, U_γ is a set of pairwise disjoint positions, $\langle\rangle \in U_\gamma$ implies $U_\gamma = \{\langle\rangle\}$. This means that if $v_\gamma = \langle\rangle$ and $\langle\rangle \in U_\gamma$, then $U_\iota = \emptyset$ for all $\gamma < \iota < \alpha$. Thus, there is only at most one $\gamma < \alpha$ with $\langle\rangle \in U_\gamma$. Therefore, we have, for each $\beta < \alpha$, some $\beta \leq \gamma < \alpha$ such that $\langle\rangle \in v_\gamma // U_\gamma$. Hence, T is destructive. □

As a consequence of this preservation of destructiveness by forming projections, we obtain that the set of fragile terms is closed under finite reductions:

Lemma 6.7 (closure of fragile terms under finite reductions)**.** *In each orthogonal TRS, the set of fragile terms is closed under finite reductions.*

Proof. Let t be a fragile term and $T\colon t \to^* t'$ a finite reduction. Hence, there is a destructive reduction starting in t. A straightforward induction proof on the length of T, using Lemma 6.6, shows that there is a destructive reduction starting in t'. Thus, t' is fragile. □

[2] Strictly speaking, if s is not a total term, i.e. it contains $\bot$, then we have to consider the system that is obtained from $\mathcal{R}$ by extending its signature to $\Sigma_\bot$.

Now we can show that destructiveness does not need more that ω steps in orthogonal, left-finite TRSs. This property will be useful for proving the equivalence of root-activeness and fragility of total terms as well the Compression Lemma for strongly p-convergent reductions.

Proposition 6.8 (Compression Lemma for destructive reductions)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and t a partial term in $\mathcal{R}$. If there is a destructive reduction starting in t, then there is a destructive reduction of length ω starting in t.*

Proof. Let $S\colon t_0 \xrightarrow{p}{\twoheadrightarrow}^{\lambda} \bot$ be a destructive reduction starting in t_0. Hence, there is some $\alpha < \lambda$ such that $S|_\alpha\colon t_0 \xrightarrow{p}{\twoheadrightarrow} s_1$, where s_1 is a ρ-redex for some $\rho\colon l \to r \in R$. Let P be the set of pattern positions of the ρ-redex s_1, i.e. $P = \mathcal{P}_\Sigma(l)$. Due to the left-finiteness of $\mathcal{R}$, P is finite. Hence, by Proposition 6.5, there is a finite reduction $t_0 \to^* s_1'$ such that s_1 and s_1' coincide in P. Hence, because $\mathcal{R}$ is left-linear, also s_1' is a ρ-redex. Now consider the reduction $T_0\colon t_0 \to^* s_1' \to_{\rho,\langle\rangle} t_1$ ending with a contraction at the root. T_0 is of finite length and, according to Lemma 6.7, t_1 is fragile.

Since t_1 is again fragile, the above argument can be iterated arbitrarily often which yields for each $i < \omega$ a finite non-empty reduction $T_i\colon t_i \to^* t_{i+1}$ whose last step is a contraction at the root. Then the concatenation $T = \prod_{i<\omega} T_i$ of these reductions is a destructive reduction of length ω starting in t_0. □

The above proposition bridges the gap between fragility and root-activeness. Whereas the former concept is defined in terms of transfinite reductions, the latter is defined in terms of finite reductions. By Proposition 6.8, however, a fragile term is always finitely reducible to a redex. This is the key to the observation that fragility is not only quite similar to root-activeness but is, in fact, essentially the same concept.

Proposition 6.9 (root-activeness = fragility)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and t a total term in $\mathcal{R}$. Then t is root-active iff t is fragile.*

Proof. The "only if" direction is easy: If t is root-active, then there is a reduction S of length ω starting in t with infinitely many steps taking place at the root. Hence, $S\colon t \xrightarrow{p}{\twoheadrightarrow}^{\omega} \bot$ is a destructive reduction, which makes t a fragile term.

For the converse direction we assume that t is fragile and show that, for each reduction $t \to^* s$, there is a reduction $s \to^* t'$ to a redex t'. By Lemma 6.7, also s is fragile. Hence, there is a destructive reduction $S\colon s \xrightarrow{p}{\twoheadrightarrow} \bot$ starting in s. According to Proposition 6.8, we can assume that S has length ω. Therefore, there is some $n < \omega$ such that $S|_n\colon s \to^* t'$ for a redex t'. □

Before we prove the missing direction of the equality of strong p-reachability and Böhm-reachability we need the property that strongly m-convergent reductions consisting only of $\to_\bot$-steps can be compressed to length at most ω as well. In order to show this, we will make use of the following lemma from Kennaway et al. [16]:

Lemma 6.10 ($\bot,\mathcal{U}$-instances)**.** *Let $\mathcal{RA}$ be the root-active terms of an orthogonal, left-finite TRS and $t \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$. If some $\bot,\mathcal{RA}$-instance of t is in $\mathcal{RA}$, then every $\bot,\mathcal{RA}$-instance of t is.*

Lemma 6.11 (compression of $\to_\bot$-steps)**.** *Consider the Böhm extension of an orthogonal TRS w.r.t. its root-active terms and $S\colon\ s \stackrel{m}{\twoheadrightarrow}_\bot t$ with $s \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$, $t \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$. Then there is a strongly m-converging reduction $T\colon\ s \stackrel{m}{\twoheadrightarrow}_\bot t$ of length at most ω that is a complete development of a set of disjoint occurrences of root-active terms in s.*

Proof. The proof is essentially the same as that of Lemma 7.2.4 from Ketema [17].

Let $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ be the mentioned reduction strongly m-converging to t_α, and let π be a position at which some reduction step in S takes place. That is, there is some β such that $\pi_\beta = \pi$. We will prove by induction on β that $t_0|_\pi \in \mathcal{RA}$.

Consider the term $t_\beta|_\pi$. Since a $\to_\bot$-rule is applied here, we have, according to Remark 2.9, that $t_\beta|_\pi \in \mathcal{RA}_\bot$. Let $V = \mathcal{P}_\bot(t_\beta|_\pi)$. Hence, for each $v \in V$, there is some $\gamma < \beta$ such that $\pi_\gamma = \pi \cdot v$. Therefore, we can apply the induction hypothesis and get that $t_0|_{\pi\cdot v} \in \mathcal{RA}$ for all $v \in V$. It is clear that we can obtain $t_0|_\pi$ from $t_\beta|_\pi$ by replacing each $\bot$-occurrence at $v \in V$ with the corresponding term $t_0|_{\pi\cdot v}$. That is, $t_0|_\pi$ is a $\bot, \mathcal{RA}$-instance of $t_\beta|_\pi$. Because $t_\beta|_\pi \in \mathcal{RA}_\bot$, there is some $\bot, \mathcal{RA}$-instance of $t_\beta|_\pi$ in $\mathcal{RA}$. Thus, by Lemma 6.10, also $t_0|_\pi$ is in $\mathcal{RA}$. This closes the proof of the claim.

Now let $V = \mathcal{P}_\bot(t_\alpha)$. Clearly, all positions in V are pairwise disjoint. Moreover, for each $v \in V$, there is a step in S that takes place at v. Hence, by the claim shown above, V is a set of occurrences in t_0 of terms in $\mathcal{RA}$. A complete development of V in t_0 leads to t_α and can be performed in at most ω steps by an outermost reduction strategy. □

The important part of the above lemma is the statement that only terms in $\mathcal{RA}$ are contracted instead of the general case where a $\to_\bot$ -step contracts a term in $\mathcal{RA}_\bot \supset \mathcal{RA}$.

Finally, we have gathered all tools necessary in order to prove the converse direction of the equivalence of strong p-reachability and Böhm-reachability w.r.t. root-active terms.

Theorem 6.12 (strong p-reachability = Böhm-reachability w.r.t. $\mathcal{RA}$)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and $\mathcal{B}$ the Böhm extension of $\mathcal{R}$ w.r.t. its root-active terms. Then $s \stackrel{p}{\twoheadrightarrow}_\mathcal{R} t$ iff $s \stackrel{m}{\twoheadrightarrow}_\mathcal{B} t$.*

Proof. The "only if" direction follows from Proposition 6.9 and Proposition 6.3.

Now consider the converse direction: Let $s \stackrel{m}{\twoheadrightarrow}_\mathcal{B} t$ be a strongly m-convergent reduction in $\mathcal{B}$. W.l.o.g. we assume s to be total. Due to Lemma 6.4, there is a term $s' \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ such that there are strongly m-convergent reductions $S\colon\ s \stackrel{m}{\twoheadrightarrow}_\mathcal{R} s'$ and $T\colon\ s' \stackrel{m}{\twoheadrightarrow}_\bot t$. By Lemma 6.11, we can assume that in $s' \stackrel{m}{\twoheadrightarrow}_\bot t$ only pairwise disjoint occurrences of root-active terms are contracted. By Proposition 6.9, each root-active term $r \in \mathcal{RA}$ is fragile, i.e. we have a destructive reduction $r \stackrel{p}{\twoheadrightarrow}_\mathcal{R} \bot$ starting in r. Thus, following Remark 2.9, we can construct a strongly p-converging reduction $T'\colon\ s' \stackrel{p}{\twoheadrightarrow}_\mathcal{R} t$ by replacing each step $C[r] \to_\bot C[\bot]$ in T with the corresponding reduction $C[r] \stackrel{p}{\twoheadrightarrow}_\mathcal{R} C[\bot]$. By combining T' with the strongly m-converging reduction S, which, according to Theorem 4.12, is also strongly p-converging, we obtain the strongly p-converging reduction $S \cdot T'\colon\ s \stackrel{p}{\twoheadrightarrow}_\mathcal{R} t$. □

6.3 Corollaries

With the equivalence of strong p-reachability and Böhm-reachability established in the previous section, strongly p-convergent reductions inherit a number of important properties that are enjoyed by Böhm-convergent reductions:

Theorem 6.13 (infinitary confluence)**.** *Every orthogonal, left-finite TRS is infinitarily confluent. That is, for each orthogonal, left-finite TRS,* $s_1 \mathrel{{}^{p}\!\!\twoheadleftarrow} t \overset{p}{\twoheadrightarrow} s_2$ *implies* $s_1 \overset{p}{\twoheadrightarrow} t' \mathrel{{}^{p}\!\!\twoheadleftarrow} s_2$.

Proof. Leveraging Theorem 6.12, this theorem follows from Theorem 2.10. □

Returning to Example 2.6 again, we can see that, in the setting of strongly p-converging reduction, the terms g^ω and f^ω can now be joined by repeatedly contracting the redex at the root which yields two destructive reductions $g^\omega \overset{p}{\twoheadrightarrow} \bot$ and $f^\omega \overset{p}{\twoheadrightarrow} \bot$, respectively.

Theorem 6.14 (infinitary normalisation)**.** *Every orthogonal, left-finite TRS is infinitarily normalising. That is, for each orthogonal, left-finite TRS* $\mathcal{R}$ *and a partial term* t *in* $\mathcal{R}$*, there is an* $\mathcal{R}$*-normal form strongly* p*-reachable from* t*.*

Proof. This follows immediately from Theorem 6.12 and Theorem 2.11. □

Combining Theorem 6.13 and Theorem 6.14, we obtain that each term in an orthogonal TRS has a unique normal form w.r.t. strong p-convergence. Due to Theorem 6.12, this unique normal form is the Böhm tree w.r.t. root-active terms.

Since strongly p-converging reductions in orthogonal TRS can always be transformed such that they consist of a prefix which is a strongly m-convergent reduction and a suffix consisting of nested destructive reductions, we can employ the Compression Lemma for strongly m-convergent reductions (Theorem 2.4) and the Compression Lemma for destructive reductions (Proposition 6.8) to obtain the Compression Lemma for strongly p-convergent reductions:

Theorem 6.15 (Compression Lemma for strongly p-convergent reductions)**.** *For each orthogonal, left-finite TRS,* $s \overset{p}{\twoheadrightarrow} t$ *implies* $s \overset{p}{\twoheadrightarrow}^{\leq\omega} t$.

Proof. Let $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$. According to Theorem 6.12, we have $s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$ for the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ and, therefore, by Lemma 6.4, we have reductions $S\colon s \overset{m}{\twoheadrightarrow}_{\mathcal{R}} s'$ and $T\colon s' \overset{m}{\twoheadrightarrow}_{\bot} t$. Due to Theorem 2.4, we can assume S to be of length at most ω and, due to Theorem 4.12, to be strongly p-convergent, i.e $S\colon s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$. If T is the empty reduction, then we are done. If not, then T is a complete development of pairwise disjoint occurrences of root-active terms according to Lemma 6.11. Hence, each step is of the form $C[r] \to_\bot C[\bot]$ for some root-active term r. By Proposition 6.9, for each such term r, there is a destructive reduction $r \overset{p}{\twoheadrightarrow}_{\mathcal{R}} \bot$ which we can assume, in accordance with Proposition 6.8, to be of length ω. Hence, each step $C[r] \to_\bot C[\bot]$ can be replaced by the reduction $C[r] \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} C[\bot]$. Concatenating these reductions results in a reduction $T'\colon s' \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ of length at most $\omega \cdot \omega$. If $S\colon s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$ is of finite length, we can interleave the reduction steps in T' such that we obtain a reduction $T''\colon s' \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} t$ of length ω. Then we have $S \cdot T''\colon s \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} t$. If $S\colon s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$ has length ω, we construct a reduction $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ as follows: As illustrated above, T' consists of destructive

reductions taking place at some pairwise disjoint positions. These steps can be interleaved into the reduction S resulting into a reduction $s \xrightarrow{p}{\twoheadrightarrow}_{\mathcal{R}} t$ of length ω. The argument for that is similar to that employed in the successor case of the induction proof of the Compression Lemma of Kennaway et al. [15].

□

We do not know whether full orthogonality is essential for the Compression Lemma. However, as for strongly m-convergent reductions, the left-linearity part of it is:

Example 6.16 ([15])**.** Consider the TRS consisting of the rules $f(x,x) \to c, a \to g(a), b \to g(b)$. Then there is a strongly p-converging reduction

$$f(a,b) \to f(g(a),b) \to f(g(a),g(b)) \to f(g(g(a)),g(b)) \to \dots\ f(g^\omega,g^\omega) \to c$$

of length $\omega{+}1$. However, there is no strongly p-converging reduction $f(a,b) \xrightarrow{p}{\twoheadrightarrow}^{\leq\omega} c$ (since there is no such strongly m-converging reduction).

We can use the Compression Lemma for strongly p-convergent reductions to obtain a stronger variant of Theorem 4.12 for orthogonal TRSs:

Corollary 6.17 (strong m-reachability = strong p-reachability of total terms)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and $s,t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$. Then $s \xrightarrow{m}{\twoheadrightarrow} t$ iff $s \xrightarrow{p}{\twoheadrightarrow} t$.*

Proof. The "only if" direction follows immediately from Theorem 4.12. For the "if" direction assume a reduction $S\colon\ s \xrightarrow{p}{\twoheadrightarrow} t$. According to Theorem 6.15, there is a reduction $T\colon\ s \xrightarrow{p}{\twoheadrightarrow}^{\leq\omega} t$. Hence, since s is total and totality is preserved by single reduction steps, $T\colon\ s \xrightarrow{p}{\twoheadrightarrow}^{\leq\omega} t$ is total. Applying Theorem 4.12, yields that $T\colon\ s \xrightarrow{m}{\twoheadrightarrow}^{\leq\omega} t$. □

7 Conclusions

Infinitary term rewriting in the partial order model provides a more fine-grained notion of convergence. Formally, every meaningful, i.e. p-continuous, reduction is also p-converging. However, p-converging reductions can end in a term containing '$\bot$'s indicating positions of local divergence. Theorem 4.9, Theorem 4.12 and Corollary 6.17 show that the partial model coincides with the metric model but additionally allows a more detailed inspection of non-m-converging reductions. Instead of the coarse discrimination between convergence and divergence provided by the metric model, the partial order model allows different levels between full convergence (a total term as result) and full divergence ($\bot$ as result).

The equivalence of strong p-reachability and Böhm-reachability shows that the differences between the metric and the partial order model can be compensated by simply adding rules that allow to replicate destructive reductions by $\to_\bot$-steps. By this equivalence, we additionally obtain infinitary normalisation and infinitary confluence for orthogonal systems – a considerable improvement over strong m-convergence. Both strong p-convergence and Böhm-convergence are defined quite differently and have independently justified intentions, yet they still induce the

same notion of transfinite reachability. This suggests that this notion of transfinite reachability can be considered a "natural" choice also because of their properties that admit unique normal forms. Nevertheless, while achieving the same goals as Böhm-extensions, the partial order approach provides a more intuitive and more elegant model for transfinite reductions as it does not need the cumbersomely defined "shortcuts" provided by $\to_\perp$-steps which depend on allowing infinite left-hand sides in rewrite rules. Vice versa destructive reductions in the partial order model provide a justification for admitting these shortcuts.

7.1 Related Work

This study of partial order convergence is inspired by Blom [5] who investigated strong partial order convergence in lambda calculus and compared it to strong metric convergence. Similarly to our findings for orthogonal term rewriting systems, Blom has shown for lambda calculus that reachability in the metric model coincides with reachability in the partial order model modulo equating so-called 0-undefined terms.

Also Corradini [6] studied a partial order model. However, he uses it to develop a theory of parallel reductions which allows simultaneous contraction of a set of mutually independent redexes of left-linear rules. To this end, Corradini defines the semantics of redex contraction in a non-standard way by allowing a partial matching of left-hand sides. Our definition of complete developments also provides, at least for orthogonal systems, a notion of parallel reductions but does so using the standard semantics of redex contraction.

7.2 Future Work

While we have studied both weak and strong p-convergence and have compared it to the respective metric counterparts, we have put the focus on strong p-convergence. It would be interesting to find out whether the shift to the partial order model has similar benefits for weak convergence, which is known to be rather unruly in the metric model [22].

Moreover, we have focused on orthogonal systems in this paper. It should be easy to generalise our results to almost orthogonal systems. The only difficulty is to deal with the ambiguity of paths when rules are allowed to overlay. This could be resolved by considering equivalence classes of paths instead. The move to weakly orthogonal systems is much more complicated: For strong m-convergence Endrullis et al. [10] have shown that weakly orthogonal systems do not even satisfy the infinitary unique normal form property (UN^∞), a property that orthogonal systems do enjoy [15]. Due to Theorem 4.12, this means that also in the setting of strong p-convergence, weakly orthogonal systems do not satisfy UN^∞ and are therefore not infinitarily confluent either! Endrullis et al. [10] have shown that this can be resolved in the metric setting by prohibiting collapsing rules. However, it is not clear whether this result can be transferred to the partial order setting.

Another interesting direction to follow is the ability to finitely simulate transfinite reductions by term graph rewriting. For strong m-convergence this is possible, at least to some extent [14]. We think that a different approach to term graph

rewriting, viz. the *double-pushout approach* [9] or the *equational approach* [1], is more appropriate for the present setting of p-convergence [3, 7].

Acknowledgements

I want to thank Bernhard Gramlich for his constant support during the work on my master's thesis which made this work possible.

Bibliography

[1] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3-4):207–240, 1996. ISSN 0169-2968. doi: 10.3233/FI-1996-263401.

[2] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[3] P. Bahr. Infinitary Rewriting - Theory and Applications. Master's thesis, Vienna University of Technology, Vienna, 2009.

[4] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[5] S. Blom. An Approximation Based Approach to Infinitary Lambda Calculi. In V. van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin / Heidelberg, 2004. doi: 10.1007/b98160.

[6] A. Corradini. Term rewriting in CTΣ. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 468–484. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-56610-4_83.

[7] A. Corradini and F. Drewes. (Cyclic) Term Graph Rewriting is adequate for Rational Parallel Term Rewriting. Technical Report TR-14-97, Universita di Pisa, Dipartimento di Informatica, 1997.

[8] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90040-9.

[9] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory*, pages 167–180, Washington, DC, USA, 1973. IEEE Computer Society. doi: 10.1109/SWAT.1973.11.

[10] J. Endrullis, C. Grabmayer, D. Hendriks, J. W. Klop, and V. van Oostrom. Unique Normal Forms in Infinitary Weakly Orthogonal Rewriting. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 85–102, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.85.

[11] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1): 68–95, 1977. ISSN 0004-5411. doi: 10.1145/321992.321997.

[12] J. L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955. ISBN 0387901256.

[13] R. Kennaway and F.-J. de Vries. Infinitary Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 12, pages 668–711. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[14] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. ISSN 0164-0925. doi: 10.1145/177492.177577.

[15] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[16] R. Kennaway, V. van Oostrom, and F.-J. de Vries. Meaningless Terms in Rewriting. *Journal of Functional and Logic Programming*, 1999(1):1–35, 1999.

[17] J. Ketema. *Böhm-Like Trees for Rewriting*. PhD thesis, Vrije Universiteit Amsterdam, 2006.

[18] J. Ketema and J. G. Simonsen. On Confluence of Infinitary Combinatory Reduction Systems. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin / Heidelberg, 2005. doi: 10.1007/11591191_15.

[19] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems: Normalising Reduction Strategies. *Logical Methods in Computer Science*, 6 (1):7, 2010. doi: 10.2168/LMCS-6(1:7)2010.

[20] J. Ketema and J. G. Simonsen. Infinitary Combinatory Reduction Systems. *Information and Computation*, 209(6):893–926, 2011. ISSN 0890-5401. doi: 10.1016/j.ic.2011.01.007.

[21] P. H. Rodenburg. Termination and Confluence in Infinitary Term Rewriting. *The Journal of Symbolic Logic*, 63(4):1286–1296, 1998. ISSN 00224812.

[22] J. G. Simonsen. On confluence and residuals in Cauchy convergent transfinite rewriting. *Information Processing Letters*, 91(3):141–146, 2004. ISSN 0020-0190. doi: 10.1016/j.ipl.2004.03.018.

[23] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

Modes of Convergence for Term Graph Rewriting

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

Term graph rewriting provides a simple mechanism to finitely represent restricted forms of infinitary term rewriting. The correspondence between infinitary term rewriting and term graph rewriting has been studied to some extent. However, this endeavour is impaired by the lack of an appropriate counterpart of infinitary rewriting on the side of term graphs. We aim to fill this gap by devising two modes of convergence based on a partial order respectively a metric on term graphs. The thus obtained structures generalise corresponding modes of convergence that are usually studied in infinitary term rewriting.

We argue that this yields a common framework in which both term rewriting and term graph rewriting can be studied. In order to substantiate our claim, we compare convergence on term graphs and on terms. In particular, we show that the modes of convergence on term graphs are conservative extensions of the corresponding modes of convergence on terms and are preserved under unravelling term graphs to terms. Moreover, we show that many of the properties known from infinitary term rewriting are preserved. This includes the intrinsic completeness of both modes of convergence and the fact that convergence via the partial order is a conservative extension of the metric convergence.

Contents

1 Introduction

Non-terminating computations are not necessarily undesirable. For instance, the termination of a reactive system would be usually considered a critical failure. Even computations that, given an input x, should produce an output y are not necessarily terminating in nature either. For example, the various iterative approximation algorithms for π produce approximations of increasing accuracy

without ever terminating with the exact value of π. While such iterative approximation computations might not reach the exact target value, they are able to come arbitrary close to the correct value within finite time.

It is this kind of non-terminating computations which is the subject of infinitary term rewriting [23]. It extends the theory of term rewriting by giving a meaning to transfinite reductions instead of dismissing them as undesired and meaningless artifacts. Following the paradigm of iterative approximations, the result of a transfinite reduction is simply the term that is approximated by the reduction. In general, such a result term can be infinite. For example, starting from the term $\mathit{rep}(0)$, the rewrite rule $\mathit{rep}(x) \to x :: \mathit{rep}(x)$ produces a reduction

$$\mathit{rep}(0) \to 0 :: \mathit{rep}(0) \to 0 :: 0 :: \mathit{rep}(0) \to 0 :: 0 :: 0 :: \mathit{rep}(0) \to \dots$$

that approximates the infinite term $0 :: 0 :: 0 :: \dots$. Here, we use $::$ as a binary symbol that we write infix and assume to associate to the right. That is, the term $0 :: 0 :: \mathit{rep}(0)$ is parenthesised as $0 :: (0 :: \mathit{rep}(0))$. Think of the $::$ symbol as the list constructor *cons*.

Term graphs, on the other hand, allow us to explicitly represent and reason about sharing and recursion [3] by dropping the restriction to a tree structure, which we have for terms. Apart from that, term graphs also provide a finite representation of certain infinite terms, viz. *rational terms*. As Kennaway et al. [22, 24] have shown, this can be leveraged in order to finitely represent restricted forms of infinitary term rewriting using *term graph rewriting*.

In this paper, we extend the theory of infinitary term rewriting to the setting of term graphs. To this end, we devise modes of convergence that constrain reductions of transfinite length in a meaningful way. Our approach to convergence is twofold: we generalise the metric on terms that is used to define convergence for infinitary term rewriting [13] to term graphs. In a similar way, we generalise the partial order on terms that has been recently used to define a closely related notion of convergence for infinitary term rewriting [7]. The use of two different – but on terms closely related – approaches to convergence will allow us both to assess the appropriateness of the resulting infinitary calculi and to compare them against the corresponding infinitary calculi of term rewriting.

1.1 Motivation

1.1.1 Lazy Evaluation

Term rewriting is a useful formalism for studying declarative programs, in particular, functional programs. A functional program essentially consists of functions defined by a set of equations and an expression that is supposed to be evaluated according to these equations. The conceptual process of evaluating an expression is nothing else than term rewriting.

A particularly interesting feature of modern functional programming languages, such as Haskell [28], is the ability to use *conceptually* infinite computations and data structures. For example, the following definition of a function `from` constructs for each number n the infinite list of consecutive numbers starting from n:

```
from(n) = n :: from(s(n))
```

Here, we use the binary infix symbol `::` to denote the list constructor *cons* and `s` for the successor function. While we cannot use the infinite list generated by `from` directly – the evaluation of an expression of the form `from n` does not terminate – we can use it in a setting in which we only read a finite prefix of the infinite list conceptually defined by `from`. Functional languages such as Haskell allow this use of semantically infinite data structures through a *non-strict evaluation* strategy, which delays the evaluation of a subexpression until its result is actually required for further evaluation of the expression. This non-strict semantics is not only a conceptual neatness but in fact one of the major features that make functional programs highly modular [17].

The above definition of the function `from` can be represented as a term rewriting system with the following rule:

$$from(x) \to x :: from(s(x))$$

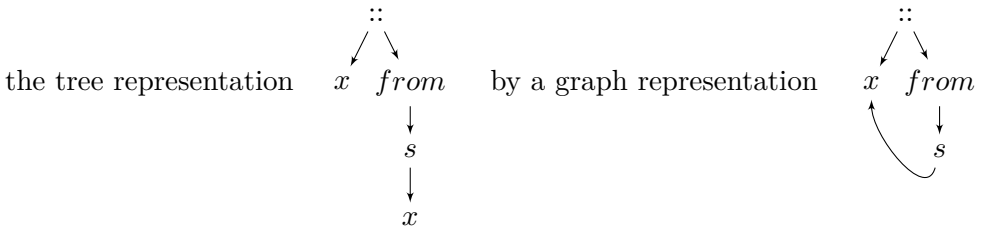
Starting with the term $from(0)$, we then obtain the following infinite reduction:

$$from(0) \to 0 :: from(s(0)) \to 0 :: s(0) :: from(s(s(0))) \to \ldots$$

Infinitary term rewriting [23] provides a notion of convergence that may assign a meaningful result term to such an infinite reduction provided there exists one. In this sense, the above reduction converges to the infinite term $0 :: s(0) :: s(s(0)) :: \ldots$, which represents the infinite list of numbers $0, 1, 2, \ldots$. Due to this extension of term rewriting with explicit limit constructions for non-terminating reductions, infinitary term rewriting allows us to directly reason about non-terminating functions and infinite data structures.

Non-strict evaluation is rarely found unescorted, though. Usually, it is implemented as *lazy evaluation* [16], which complements a non-strict evaluation strategy with *sharing*. The latter avoids duplication of subexpressions by using pointers instead of copying. For example, the function `from` above duplicates its argument `n` – it occurs twice on the right-hand side of the defining equation. A lazy evaluator simulates this duplication by inserting two pointers pointing to the actual argument. Sharing is a natural companion for non-strict evaluation as it avoids re-evaluation of expressions that are duplicated before they are evaluated.

The underlying formalism that is typically used to obtain sharing for functional programming languages is term graph rewriting [29, 30]. Term graph rewriting [10, 31] uses graphs to represent terms thus allowing multiple arcs to point to the same node. For example, term graphs allows us to change the representation of the term rewrite rule defining the function $from$ by replacing

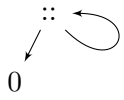


which shares the variable x by having two arcs pointing to it.

While infinitary term rewriting is used to model the non-strictness of lazy evaluation, term graph rewriting models the sharing part of it. By endowing term graph rewriting with a notion of convergence, we aim to unify the two formalisms into one calculus, thus allowing us to model both aspects withing the same calculus.

1.1.2 Rational Terms

Term graphs can do more than only share common subexpressions. Through cycles term graphs may also provide a finite representation of certain infinite terms – so-called *rational terms*. For example, the infinite term $0 :: 0 :: 0 :: \ldots$ can be represented as the finite term graph



Since a single node on a cycle in a term graph represents infinitely many corresponding subterms, the contraction of a single term graph redex may correspond to a transfinite term reduction that contracts infinitely many term redexes. For example, if we apply the rewrite rule $0 \to s(0)$ to the above term graph, we obtain a term graph that represents the term $s(0) :: s(0) :: s(0) :: \ldots$, which can only be obtained from the term $0 :: 0 :: 0 :: \ldots$ via a *transfinite* term reduction with the rule $0 \to s(0)$. Kennaway et al. [24] investigated this correspondence between cyclic term graph rewriting and infinitary term rewriting. Among other results they characterise a subset of transfinite term reductions – called *rational reductions* – that can be simulated by a corresponding finite term graph reduction. The above reduction from the term $0 :: 0 :: 0 :: \ldots$ is an example of such a rational reduction.

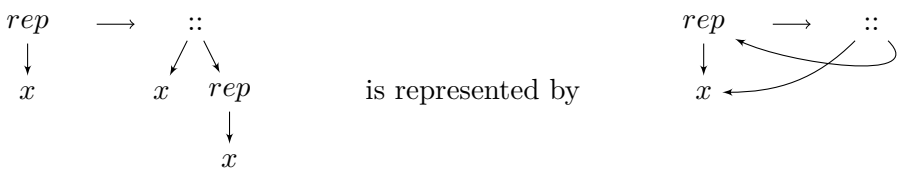
With the help of a unified formalism for infinitary and term graph rewriting, it should be easier to study the correspondence between infinitary term rewriting and finitary term graph rewriting further. The move from an infinitary term rewriting system to a term graph rewriting system only amounts to a change in the degree of sharing if we use infinitary term graph rewriting as a common framework.

Reconsider the term rewrite rule $rep(x) \to x :: rep(x)$, which defines a function rep that repeats its argument infinitely often:

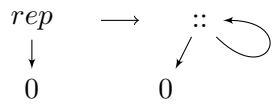
$$rep(0) \to 0 :: rep(0) \to 0 :: 0 :: rep(0) \to 0 :: 0 :: 0 :: rep(0) \to \quad \ldots \quad 0 :: 0 :: 0 :: \ldots$$

This reduction happens to be not a rational reduction in the sense of Kennaway et al. [24].

The move from the term rule $rep(x) \to x :: rep(x)$ to a term graph rule is a simple matter of introducing sharing of common subexpressions:



Instead of creating a fresh copy of the redex on the right-hand side, the redex is reused by placing an edge from the right-hand side of the rule to its left-hand side. This allows us to represent the infinite reduction approximating the infinite term $0 :: 0 :: 0 :: \ldots$ with the following single step term graph reduction induced by the above term graph rule:



Via its cyclic structure the resulting term graph represents the infinite term $0 :: 0 :: 0 :: \ldots.$

Since both transfinite term reductions and the corresponding finite term graph reductions can be treated within the same formalism, we hope to provide a tool for studying the ability of cyclic term graph rewriting to finitely represent transfinite term reductions.

1.2 Contributions & Related Work

1.2.1 Contributions

The main contributions of this paper are the following:

(i) We devise a partial order on term graphs based on a restricted class of graph homomorphisms. We show that this partial order forms a complete semi-lattice and thus is technically suitable for defining a notion of convergence (Theorem 5.15). Moreover, we illustrate alternative partial orders and show why they are not suitable for formalising convergence on term graphs.

(ii) Independently, we devise a metric on term graphs and show that it forms a complete ultrametric space on term graphs (Theorem 7.4).

(iii) Based on the partial order respectively the metric we define a notion of *weak convergence* for infinitary term graph rewriting. We show that – similar to the term rewriting case [7] – the metric calculus of infinitary term graph rewriting is the *total fragment* of the partial order calculus of infinitary term graph rewriting (Theorem 8.10).

(iv) We confirm that the partial order and the metric on term graphs generalise the partial order respectively the metric that is used for infinitary term rewriting (Proposition 5.19 and 6.16). Moreover, we show that the corresponding notions of convergence are preserved by unravelling term graphs to terms thus establishing the soundness of our notions of convergence on term graphs w.r.t. the convergence on terms (Theorems 9.9 and 9.11).

(v) We substantiate the appropriateness of our calculi by a number of examples that illustrate how increasing the sharing gradually reduces the number of steps necessary to reach the final result – eventually, from an infinite number of steps to a finite number (Sections 8 and 9).

1.2.2 Related Work

Calculi with explicit sharing and/or recursion, e.g. via *letrec*, can also be considered as a form of term graph rewriting. Ariola and Klop [3] recognised that adding such an explicit recursion mechanism to the lambda calculus may break confluence. In order to reconcile this, Ariola and Blom [1, 2] developed a notion of skew confluence that allows them to define an infinite normal form in the vein of Böhm trees.

Recently, we have investigated other notions of convergence for term graph rewriting [8, 9] that use simpler variants of the partial order and the metric that we use in this paper. Both of them have theoretically pleasing properties, e.g. the ideal completion and the metric completion of the set of finite term graphs both yield the set of all term graphs. However, the resulting notions of weak convergence are not fully satisfying and in fact counterintuitive for some cases. We will discuss this alternative approach and compare it to the present approach in more detail in Sections 5 and 6.

1.3 Overview

The structure of this paper is as follows: in Section 2, we provide the necessary background for metric spaces, partially ordered sets and term rewriting. In Section 3, we give an overview of infinitary term rewriting. Section 4 provides the necessary theory for graphs and term graphs. Sections 5 and 6 form the core of this paper. In these sections we study the partial order and the metric on term graphs that are the basis for the modes of convergence we propose in this paper. In Section 7, we then compare the two resulting modes of convergence. Moreover, in Section 8, we use these two modes of convergence to study two corresponding infinitary term graph rewriting calculi. In Section 9, we study the correspondences between infinitary term graph rewriting and infinitary term rewriting.

Some proofs have been omitted from the main body of the text. These proofs can be found in the appendix of this paper.

Contents

2 Preliminaries

We assume the reader to be familiar with the basic theory of ordinal numbers, orders and topological spaces [20], as well as term rewriting [33]. In order to make this paper self-contained, we briefly recall all necessary preliminaries.

2.1 Sequences

We use the von Neumann definition of ordinal numbers. That is, an *ordinal number* (or simply *ordinal*) α is the set of all ordinal numbers strictly smaller than α. In particular, each natural number $n \in \mathbb{N}$ is an ordinal number with $n = \{0, 1, \ldots, n-1\}$. The least infinite ordinal number is denoted by ω and is

the set of all natural numbers. Ordinal numbers will be denoted by lower case Greek letters $\alpha, \beta, \gamma, \lambda, \iota$.

A *sequence* S of *length* α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*. For a finite sequence $(a_i)_{i<n}$ we also use the notation $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. In particular, $\langle\rangle$ denotes the empty sequence. We write A^* for the set of all finite sequences in A.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a (proper) *prefix* of a sequence T, denoted $S \leq T$ (respectively $S < T$), if there is a (non-empty) sequence S' with $S \cdot S' = T$. The prefix of T of length $\beta \leq |T|$ is denoted $T|_\beta$. Similarly, a sequence S is a (proper) *suffix* of a sequence T if there is a (non-empty) sequence S' with $S' \cdot S = T$.

2.2 Metric Spaces

A pair $(M, \mathbf{d})$ is called a *metric space* if $\mathbf{d}$ is a *metric* on the set M. That is, $\mathbf{d}\colon M \times M \to \mathbb{R}_0^+$ is a function satisfying $\mathbf{d}(x, y) = 0$ iff $x = y$ (identity), $\mathbf{d}(x, y) = \mathbf{d}(y, x)$ (symmetry), and $\mathbf{d}(x, z) \leq \mathbf{d}(x, y)+\mathbf{d}(y, z)$ (triangle inequality), for all $x, y, z \in M$. If $\mathbf{d}$ instead of the triangle inequality, satisfies the stronger property $\mathbf{d}(x, z) \leq \max\{\mathbf{d}(x, y), \mathbf{d}(y, z)\}$ (strong triangle), then $(M, \mathbf{d})$ is called an *ultrametric space*.

Let $(a_\iota)_{\iota<\alpha}$ be a sequence in a metric space $(M, \mathbf{d})$. The sequence $(a_\iota)_{\iota<\alpha}$ *converges* to an element $a \in M$, written $\lim_{\iota\to\alpha} a_\iota$, if, for each $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that $\mathbf{d}(a, a_\iota) < \varepsilon$ for every $\beta < \iota < \alpha$; $(a_\iota)_{\iota<\alpha}$ is *continuous* if $\lim_{\iota\to\lambda} a_\iota = a_\lambda$ for each limit ordinal $\lambda < \alpha$. Intuitively speaking, $(a_\iota)_{\iota<\alpha}$ converges to a if the metric distance between the elements a_ι of the sequence and a tends to 0 as the index ι approaches α, i.e. they approximate a arbitrarily well. Accordingly, $(a_\iota)_{\iota<\alpha}$ is continuous if it does not leap to a distant object at limit ordinal indices.

The sequence $(a_\iota)_{\iota<\alpha}$ is called *Cauchy* if, for any $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that, for all $\beta < \iota < \iota' < \alpha$, we have that $\mathbf{d}(m_\iota, m_{\iota'}) < \varepsilon$. That is, the elements a_ι of the sequence move closer and closer to each other as the index ι approaches α.

A metric space is called *complete* if each of its non-empty Cauchy sequences converges. That is, whenever the elements a_ι of a sequence move closer and closer together, they in fact approximate an existing object of the metric space, viz. $\lim_{\iota\to\alpha} a_\iota$.

2.3 Partial Orders

A *partial order* $\leq$ on a set A is a binary relation on A such that $x \leq y, y \leq z$ implies $x \leq z$ (transitivity); $x \leq x$ (reflexivity); and $x \leq y, y \leq x$ implies $x = y$ (antisymmetry) for all $x, y, z \in A$. The pair $(A, \leq)$ is then called a *partially ordered set*. A subset D of the underlying set A is called *directed* if it is non-empty and each pair of elements in D has an upper bound in D. A partially

ordered set $(A, \leq)$ is called a *complete partial order* (*cpo*) if it has a least element and each directed set D has a *least upper bound* (*lub*) $\bigsqcup D$. A cpo $(A, \leq)$ is called a *complete semilattice* if every *non-empty* set B has *greatest lower bound* (*glb*) $\bigsqcap B$. In particular, this means that, in a complete semilattice, the *limit inferior* of any sequence $(a_\iota)_{\iota<\alpha}$, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha} \left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists.

There is also an alternative characterisation of complete semilattices: a partially ordered set $(A, \leq)$ is called *bounded complete* if each set $B \subseteq A$ that has an upper bound in A also has a least upper bound in A. Two elements $a, b \in A$ are called *compatible* if they have a common upper bound, i.e. there is some $c \in A$ with $a, b \leq c$.

Proposition 2.1 (bounded complete cpo = complete semilattice, [18])**.** *Given a cpo $(A, \leq)$ the following are equivalent:*

(i) $(A, \leq)$ *is a complete semilattice.*

(ii) $(A, \leq)$ *is bounded complete.*

(iii) Each two compatible elements in A have a least upper bound.

Given two partially ordered sets $(A, \leq_A)$ and $(B, \leq_B)$, a function $\phi\colon A \to B$ is called *monotonic* iff $a_1 \leq_A a_2$ implies $\phi(a_1) \leq_B \phi(a_2)$. In particular, we have that a sequence $(b_\iota)_{\iota<\alpha}$ in $(B, \leq_B)$ is monotonic if $b_\iota \leq_B b_\gamma$ for all $\iota \leq \gamma < \alpha$.

2.4 Terms

Since we are interested in the infinitary calculus of term rewriting, we consider the set $\mathcal{T}^\infty(\Sigma)$ of *infinitary terms* (or simply *terms*) over some *signature* Σ. A *signature* Σ is a countable set of symbols such that each symbol $f \in \Sigma$ is associated with an arity $\mathsf{ar}(f) \in \mathbb{N}$, and we write $\Sigma^{(n)}$ for the set of symbols in Σ that have arity n. The set $\mathcal{T}^\infty(\Sigma)$ is defined as the *greatest* set T such that $t \in T$ implies $t = f(t_1, \ldots, t_k)$ for some $f \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T$. For each constant symbol $c \in \Sigma^{(0)}$, we write c for the term $c()$. For a term $t \in \mathcal{T}^\infty(\Sigma)$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. $\mathcal{P}(t)$ is the least subset of $\mathbb{N}^*$ such that $\langle\rangle \in \mathcal{P}(t)$ and $\langle i\rangle \cdot \pi \in \mathcal{P}(t)$ if $t = f(t_0, \ldots, t_{k-1})$ with $0 \leq i < k$ and $\pi \in \mathcal{P}(t_i)$. For terms $s, t \in \mathcal{T}^\infty(\Sigma)$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, $t(\pi)$ for the function symbol in t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s. As positions are sequences, we use the prefix order $\leq$ defined on them. A position is also called an *occurrence* if the focus lies on the subterm at that position rather than the position itself. The set $\mathcal{T}(\Sigma)$ of *finite terms* is the subset of $\mathcal{T}^\infty(\Sigma)$ that contains all terms with a finite set of positions.

On $\mathcal{T}^\infty(\Sigma)$ a similarity measure $\mathsf{sim}(\cdot,\cdot)\colon \mathcal{T}^\infty(\Sigma) \times \mathcal{T}^\infty(\Sigma) \to \omega + 1$ is defined as follows

$$\mathsf{sim}(s,t) = \min\{|\pi| \mid \pi \in \mathcal{P}(s) \cap \mathcal{P}(t), s(\pi) \neq t(\pi)\} \cup \{\omega\} \qquad \text{for } s, t \in \mathcal{T}^\infty(\Sigma)$$

That is, $\mathsf{sim}(s,t)$ is the minimal depth at which s and t differ, respectively ω if $s = t$. Based on this similarity measure, a distance function $\mathbf{d}$ is defined by

$\mathbf{d}(s,t) = 2^{-\mathsf{sim}(s,t)}$, where we interpret $2^{-\omega}$ as 0. The pair $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$ is known to form a complete ultrametric space [4].

Partial terms, i.e. terms over signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$ with $\bot$ a fresh nullary symbol, can be endowed with a binary relation $\leq_\bot$ by defining $s \leq_\bot t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\bot$. Interpreting the term $\bot$ as denoting "undefined", $\leq_\bot$ can be read as "is less defined than". The pair $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ is known to form a complete semilattice [15]. To explicitly distinguish them from partial terms, we call terms in $\mathcal{T}^\infty(\Sigma)$ *total*.

2.5 Term Rewriting Systems

For term rewriting systems, we have to consider terms with variables. To this end, we assume a countably infinite set $\mathcal{V}$ of variables and extend a signature Σ to a signature $\Sigma_\mathcal{V} = \Sigma \uplus \mathcal{V}$ with variables in $\mathcal{V}$ as nullary symbols. Instead of $\mathcal{T}^\infty(\Sigma_\mathcal{V})$ we also write $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}^\infty(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ such that all variables occurring in r also occur in l. Note that both the left- and the right-hand side may be infinite. We usually use x, y, z and primed respectively indexed variants thereof to denote variables in $\mathcal{V}$. A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. Such a substitution σ can be uniquely lifted to a homomorphism from $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ to $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ mapping a term $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ to $t\sigma$ by setting $x\sigma = \sigma(x)$ if $x \in \mathcal{V}$ and $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ if $f \in \Sigma^{(n)}$.

As in the finitary setting, every TRS $\mathcal{R}$ defines a *rewrite relation* $\to_\mathcal{R}$ that indicates *rewrite steps*:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \sigma\colon\ s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

Instead of $s \to_\mathcal{R} t$, we sometimes write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π, or simply $s \to t$. The subterm $s|_\pi$ is called a *ρ-redex* or simply *redex*, $r\sigma$ its *contractum*, and $s|_\pi$ is said to be *contracted* to $r\sigma$.

3 Infinitary Term Rewriting

Before pondering over the right approach to an infinitary calculus of term graph rewriting, we want to provide a brief overview of infinitary term rewriting [7, 12, 23]. This should give an insight into the different approaches to dealing with infinite reductions. However, in contrast to the majority of the literature on infinitary term rewriting, which is concerned with strong convergence [23, 25], we will only consider weak notions of convergence in this paper; cf. [13, 19, 32]. This weak form of convergence, also called *Cauchy convergence*, is entirely based on the sequence of objects produced by rewriting without considering *how* the rewrite rules are applied.

A *(transfinite) reduction* in a term rewriting system $\mathcal{R}$, is a sequence $S = (t_\iota \to_\mathcal{R} t_{\iota+1})_{\iota<\alpha}$ of rewrite steps in $\mathcal{R}$. Note that the underlying sequence of terms $(t_\iota)_{\iota<\widehat{\alpha}}$ has length $\widehat{\alpha}$, where $\widehat{\alpha} = \alpha$ if S is open, and $\widehat{\alpha} = \alpha + 1$ if S is closed. The reduction S is called *m-continuous* in $\mathcal{R}$, written $S\colon\ t_0 \xrightarrow{m}_\mathcal{R} \ldots$, if the sequence of terms $(t_\iota)_{\iota<\widehat{\alpha}}$ is continuous in $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$, i.e. $\lim_{\iota\to\lambda} t_\iota = t_\lambda$ for

each limit ordinal $\lambda < \alpha$. The reduction S is said to *m-converge* to a term t in $\mathcal{R}$, written $S\colon\ t_0 \xrightarrow{m}_{\mathcal{R}} t$, if it is m-continuous and $\lim_{\iota \to \widehat{\alpha}} t_\iota = t$.

Example 3.1. Consider the TRS $\mathcal{R}$ containing the rule $\rho_1\colon\ a :: x \to b :: a :: x$. By repeatedly applying ρ_1, we obtain the infinite reduction

$$S\colon\ a :: c \to b :: a :: c \to b :: b :: a :: c \to \dots$$

The position at which two consecutive terms differ moves deeper and deeper during the reduction S, i.e. the **d**-distance between them tends to 0. Hence, S m-converges to the infinite term $s = b :: b :: b :: \dots$, i.e. $S\colon\ a :: c \xrightarrow{m} s$.

Now consider a TRS with the slightly different rule $\rho_2\colon\ a :: x \to a :: b :: x$. This TRS yields a reduction

$$S'\colon\ a :: c \to a :: b :: c \to a :: b :: b :: c \to \dots$$

Even though the rule ρ_2 is applied at the root of the term in each step of S', the **d**-distance between two consecutive terms tends to 0 again. The reduction S' m-converges to the infinite term $s' = a :: b :: b :: \dots$, i.e. $S'\colon\ a :: c \xrightarrow{m} s'$.

In contrast to the weak m-convergence that we consider here, strong m-convergence [23, 25] additionally requires that the depth of the contracted redexes tends to infinity as the reduction approaches a limit ordinal. Concerning Example 3.1 above, we have for instance that S also strongly m-converges – the rule is applied at increasingly deep redexes – whereas S' does not strongly m-converge – each step in S' results from a contraction at the root.

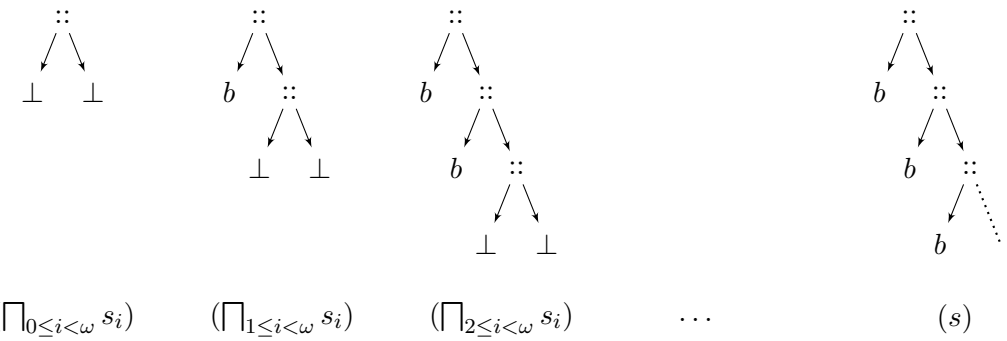
In the partial order model of infinitary rewriting [7], convergence is defined via the limit inferior in the complete semilattice $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$. Given a TRS $\mathcal{R} = (\Sigma, R)$, we extend it to $\mathcal{R}_\bot = (\Sigma_\bot, R)$ by adding the fresh constant symbol $\bot$ such that it admits all terms in $\mathcal{T}^\infty(\Sigma_\bot)$. A reduction $S = (t_\iota \to_{\mathcal{R}_\bot} t_{\iota+1})_{\iota<\alpha}$ in this system $\mathcal{R}_\bot$ is called *p-continuous* in $\mathcal{R}$, written $S\colon\ t_0 \xrightarrow{p}_{\mathcal{R}} \dots$, if $\liminf_{\iota \to \lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$. The reduction S is said to *p-converge* to a term t in $\mathcal{R}$, written $S\colon\ t_0 \xrightarrow{p}_{\mathcal{R}} t$, if it is p-continuous and $\liminf_{\iota \to \widehat{\alpha}} t_\iota = t$.

The distinguishing feature of the partial order approach is that each continuous reduction also converges due to the semilattice structure of partial terms. Moreover, p-convergence provides a conservative extension to m-convergence that allows rewriting modulo *meaningless terms* [7] by essentially mapping those parts of the reduction to $\bot$ that are divergent according to the metric mode of convergence.

Intuitively, the limit inferior in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ – and thus p-convergence – describes an approximation process that accumulates each piece of information that remains *stable* from some point onwards. This is based on the ability of the partial order $\leq_\bot$ to capture a notion of *information preservation*, i.e. $s \leq_\bot t$ iff t contains at least the same information as s does but potentially more. A monotonic sequence of terms $t_0 \leq_\bot t_1 \leq_\bot \dots$ thus approximates the information contained in $\bigsqcup_{i<\omega} t_i$. Given this reading of $\leq_\bot$, the glb $\bigsqcap T$ of a set of terms T captures the common (non-contradicting) information of the terms in T. Leveraging this observation, a sequence that is not necessarily monotonic can be turned into a monotonic sequence $t_j = \bigsqcap_{j \leq i < \omega} s_i$ such that each t_j contains exactly the

information that remains stable in $(s_i)_{i<\omega}$ from j onwards. Hence, the limit inferior $\liminf_{i\to\omega} s_i = \bigsqcup_{j<\omega} \bigsqcap_{j\le i<\omega} s_i$ is the term that contains the accumulated information that eventually remains stable in $(s_i)_{i<\omega}$. This is expressed as an approximation of the monotonically increasing information that remains stable from some point on.

Example 3.2. Reconsider the system from Example 3.1. The reduction S also p-converges to s. This can be seen by forming the sequence $(\bigsqcap_{j\le i<\omega} s_i)_{i<\omega}$ of stable information of the underlying sequence $(s_i)_{i<\omega}$ of terms in S:



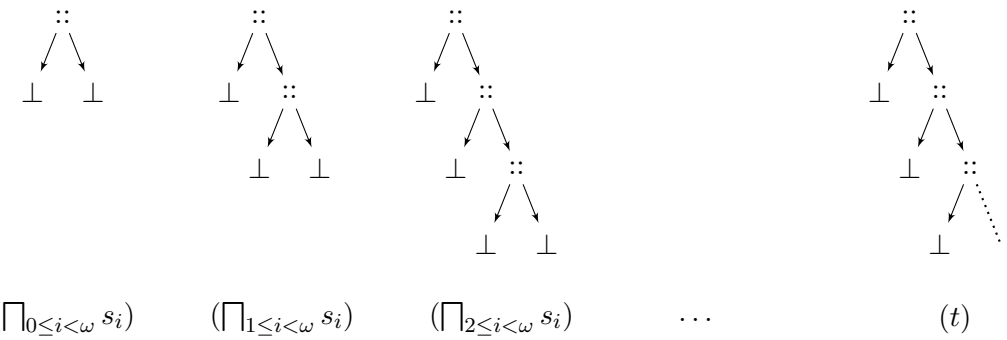
$(\bigsqcap_{0\le i<\omega} s_i)$ $(\bigsqcap_{1\le i<\omega} s_i)$ $(\bigsqcap_{2\le i<\omega} s_i)$ $\cdots$ (s)

This sequence approximates the term $s = b :: b :: b :: \ldots$.

Now consider the rule ρ_1 together with the rule $\rho_3\colon\ b :: x \to a :: b :: x$. Starting with the same term, but applying the two rules alternately at the root, we obtain the reduction sequence

$$T\colon\ a :: c \to b :: a :: c \to a :: b :: a :: c \to b :: a :: b :: a :: c \to \ldots$$

Now the differences between two consecutive terms occur right below the root symbol " :: ". Hence, T does not m-converge. This, however, only affects the left argument of each " :: ". Following the right argument position, the bare list structure becomes eventually stable. The sequence $(\bigsqcap_{j\le i<\omega} s_i)_{i<\omega}$ of stable information



$(\bigsqcap_{0\le i<\omega} s_i)$ $(\bigsqcap_{1\le i<\omega} s_i)$ $(\bigsqcap_{2\le i<\omega} s_i)$ $\cdots$ (t)

approximates the term $t = \bot :: \bot :: \bot \ldots$. Hence, T p-converges to t.

Note that in both the metric and the partial order setting continuity is simply the convergence of every proper prefix: a reduction $S = (t_\iota \to t_{\iota+1})_{\iota<\alpha}$ is m-continuous (respectively p-continuous) iff every proper prefix $S|_\beta$ m-converges (respectively p-converges) to t_β.

In order to define p-convergence, we had to extend terms with partiality. However, apart from this extension, both m- and p-convergence coincide. To describe this more precisely we use the following terms: a reduction $S\colon s \overset{p}{\rightarrow} \ldots$ is *p-continuous in $\mathcal{T}^\infty(\Sigma)$* iff each term in S is total, i.e. in $\mathcal{T}^\infty(\Sigma)$; a reduction $S\colon s \overset{p}{\rightarrow} t$ is called *p-convergent in $\mathcal{T}^\infty(\Sigma)$* iff t and each term in S is total. We then have the following theorem:

Theorem 3.3 (p-convergence in $\mathcal{T}^\infty(\Sigma) = m$-convergence, [5])**.** *For every reduction S in a TRS the following equivalences hold:*

(i) $S\colon s \overset{p}{\rightarrow} t$ *in* $\mathcal{T}^\infty(\Sigma)$ *iff* $S\colon s \overset{m}{\Rightarrow} t$
(ii) $S\colon s \overset{p}{\rightarrow} \ldots$ *in* $\mathcal{T}^\infty(\Sigma)$ *iff* $S\colon s \overset{m}{\Rightarrow} \ldots$

Example 3.2 illustrates the correspondence between p- and m-convergence: the reduction S p-converges in $\mathcal{T}^\infty(\Sigma)$ and m-converges whereas the reduction T p-converges but not in $\mathcal{T}^\infty(\Sigma)$ and thus does not m-converge.

Kennaway [21] and Bahr [6] investigated abstract models of infinitary rewriting based on metric spaces respectively partially ordered sets. We shall take these abstract models as a basis to formulate a theory of infinitary term graph reductions. The key question that we have to address is what an appropriate metric space respectively partial order on term graphs looks like.

4 Graphs & Term Graphs

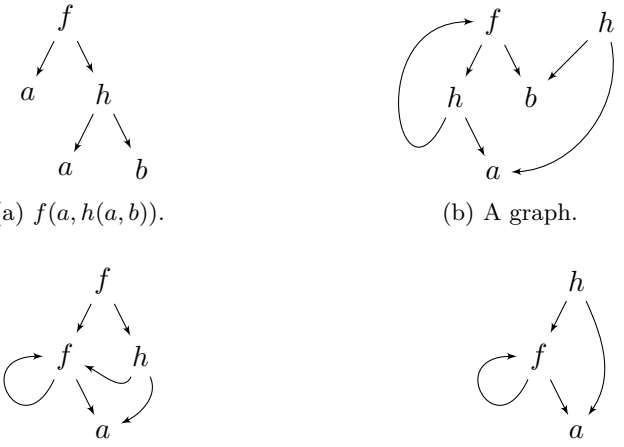
This section provides the basic notions for term graphs and more generally for graphs. Terms over a signature, say Σ, can be thought of as rooted trees whose nodes are labelled with symbols from Σ. Moreover, in these trees a node labelled with a k-ary symbol is restricted to have out-degree k and the outgoing edges are ordered. In this way the i-th successor of a node labelled with a symbol f is interpreted as the root node of the subtree that represents the i-th argument of f. For example, consider the term $f(a, h(a, b))$. The corresponding representation as a tree is shown in Figure 1a.

In term graphs, the restriction to a tree structure is abolished. The corresponding notion of term graphs we are using is taken from Barendregt et al. [10]. We begin by defining the underlying notion of graphs.

Definition 4.1 (graphs)**.** Let Σ be a signature. A *graph* over Σ is a tuple $g = (N, \mathsf{lab}, \mathsf{suc})$ consisting of a set N (of *nodes*), a *labelling function* $\mathsf{lab}\colon N \to \Sigma$, and a *successor function* $\mathsf{suc}\colon N \to N^*$ such that $|\mathsf{suc}(n)| = \mathsf{ar}(\mathsf{lab}(n))$ for each node $n \in N$, i.e. a node labelled with a k-ary symbol has precisely k successors. The graph g is called *finite* whenever the underlying set N of nodes is finite. If $\mathsf{suc}(n) = \langle n_0, \ldots, n_{k-1}\rangle$, then we write $\mathsf{suc}_i(n)$ for n_i. Moreover, we use the abbreviation $\mathsf{ar}_g(n)$ for the arity $\mathsf{ar}(\mathsf{lab}(n))$ of n.

Example 4.2. Let $\Sigma = \{f/2, h/2, a/0, b/0\}$ be a signature. The graph over Σ, depicted in Figure 1b, is given by the triple $(N, \mathsf{lab}, \mathsf{suc})$ with $N = \{n_0, n_1, n_2, n_3, n_4\}$, $\mathsf{lab}(n_0) = f, \mathsf{lab}(n_1) = \mathsf{lab}(n_4) = h, \mathsf{lab}(n_2) = b, \mathsf{lab}(n_3) = a$ and $\mathsf{suc}(n_0) = \langle n_1, n_2\rangle, \mathsf{suc}(n_1) = \langle n_0, n_3\rangle, \mathsf{suc}(n_2) = \mathsf{suc}(n_3) = \langle\rangle, \mathsf{suc}(n_4) = \langle n_2, n_3\rangle$.

Definition 4.3 (paths, reachability)**.** Let $g = (N, \mathsf{lab}, \mathsf{suc})$ be a graph and $n, m \in N$.



(a) $f(a, h(a, b))$. (b) A graph.

(c) A term graph g. (d) Sub-term graph of g.

Figure 1: Tree representation of a term and generalisation to (term) graphs.

(i) A *path* in g from n to m is a finite sequence $\pi \in \mathbb{N}^*$ such that either

- π is empty and $n = m$, or
- $\pi = \langle i \rangle \cdot \pi'$ with $0 \leq i < \mathsf{ar}_g(n)$ and the suffix π' is a path in g from $\mathsf{suc}_i(n)$ to m.

(ii) If there exists a path from n to m in g, we say that m is *reachable* from n in g.

Since paths are sequences, we may use the prefix order on sequences for paths as well. That is, we write $\pi_1 \leq \pi_2$ (respectively $\pi_1 < \pi_2$) if there is a (non-empty) path π_3 with $\pi_1 \cdot \pi_3 = \pi_2$.

Definition 4.4 (term graphs)**.** Given a signature Σ, a *term graph* g over Σ is a tuple $(N, \mathsf{lab}, \mathsf{suc}, r)$ consisting of an *underlying* graph $(N, \mathsf{lab}, \mathsf{suc})$ over Σ whose nodes are all reachable from the *root node* $r \in N$. The term graph g is called *finite* if the underlying graph is finite, i.e. the set N of nodes is finite. The class of all term graphs over Σ is denoted $\mathcal{G}^\infty(\Sigma)$; the class of all finite term graphs over Σ is denoted $\mathcal{G}(\Sigma)$. We use the notation N^g, lab^g, suc^g and r^g to refer to the respective components N,lab, suc and r of g. In analogy to subterms, term graphs have *sub-term graphs*. Given a graph or a term graph h and a node n in h, we write $h|_n$ to denote the sub-term graph of h rooted in n.

Example 4.5. Let $\Sigma = \{f/2, h/2, c/0\}$ be a signature. The term graph g over Σ, depicted in Figure 1c, is given by the quadruple $(N, \mathsf{lab}, \mathsf{suc}, r)$, where $N = \{r, n_1, n_2, n_3\}$, $\mathsf{suc}(r) = \langle n_1, n_2 \rangle$, $\mathsf{suc}(n_1) = \langle n_1, n_3 \rangle$, $\mathsf{suc}(n_2) = \langle n_1, n_3 \rangle$, $\mathsf{suc}(n_3) = \langle\rangle$ and $\mathsf{lab}(r) = \mathsf{lab}(n_1) = f$, $\mathsf{lab}(n_2) = h$, $\mathsf{lab}(n_3) = c$. Figure 1d depicts the sub-term graph $g|_{n_2}$ of g.

Paths in a graph are not absolute but relative to a starting node. In term graphs, however, we have a distinguished root node from which each node is reachable. Paths relative to the root node are central for dealing with term graphs:

Definition 4.6 (positions, depth, cyclicity, trees)**.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $n \in N^g$.

(i) A *position* of n in g is a path in the underlying graph of g from r^g to n. The set of all positions in g is denoted $\mathcal{P}(g)$; the set of all positions of n in g is denoted $\mathcal{P}_g(n)$.[1]

(ii) The *depth* of n in g, denoted $\mathsf{depth}_g(n)$, is the minimum of the lengths of the positions of n in g, i.e. $\mathsf{depth}_g(n) = \min\{|\pi| \mid \pi \in \mathcal{P}_g(n)\}$.

(iii) For a position $\pi \in \mathcal{P}(g)$, we write $\mathsf{node}_g(\pi)$ for the unique node $n \in N^g$ with $\pi \in \mathcal{P}_g(n)$ and $g(\pi)$ for its symbol $\mathsf{lab}^g(n)$.

(iv) A position $\pi \in \mathcal{P}(g)$ is called *cyclic* if there are paths $\pi_1 < \pi_2 \leq \pi$ with $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$, i.e. π passes a node twice. The non-empty path π' with $\pi_1 \cdot \pi' = \pi_2$ is then called a *cycle* of $\mathsf{node}_g(\pi_1)$. A position that is not cyclic is called *acyclic*. If g has a cyclic position, g is called cyclic; otherwise g is called acyclic.

(v) The term graph g is called a *term tree* if each node in g has exactly one position.

Note that the labelling function of graphs – and thus term graphs – is *total*. In contrast, Barendregt et al. [10] considered *open* (term) graphs with a *partial* labelling function such that unlabelled nodes denote holes or variables. This is reflected in their notion of homomorphisms in which the homomorphism condition is suspended for unlabelled nodes.

4.1 Homomorphisms

Instead of a partial node labelling function for term graphs, we chose a *syntactic* approach that is closer to the representation in terms: variables, holes and "bottoms" are represented as distinguished syntactic entities. We achieve this on term graphs by making the notion of homomorphisms dependent on a set of constant symbols Δ for which the homomorphism condition is suspended:

Definition 4.7 (Δ-homomorphisms)**.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$, and $g, h \in \mathcal{G}^\infty(\Sigma)$.

(i) A function $\phi\colon N^g \to N^h$ is called *homomorphic* in $n \in N^g$ if the following holds:

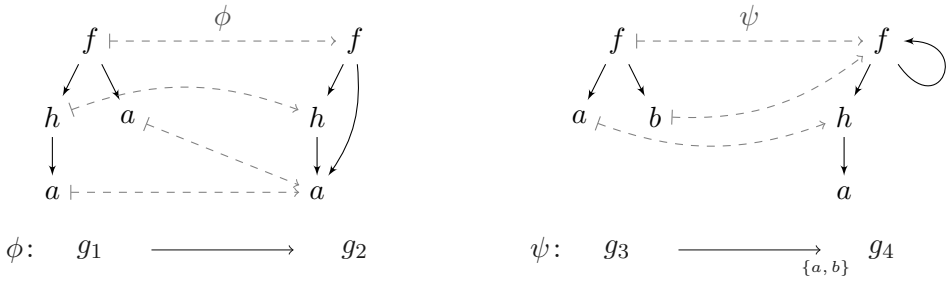
$$\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n)) \qquad \text{(labelling)}$$

$$\phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^h_i(\phi(n)) \quad \text{for all } 0 \leq i < \mathsf{ar}_g(n) \qquad \text{(successor)}$$

(ii) A Δ-*homomorphism* ϕ from g to h, denoted $\phi\colon g \to_\Delta h$, is a function $\phi\colon N^g \to N^h$ that is homomorphic in n for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$ and satisfies

$$\phi(r^g) = r^h \qquad \text{(root)}$$

[1] The notion/notation of positions is borrowed from terms: Every position π of a node n corresponds to the subterm represented by n occurring at position π in the unravelling of the term graph to a term.



(a) A homomorphism. (b) A $\{a, b\}$-homomorphism.

Figure 2: Δ-homomorphisms.

Note that, for $\Delta = \emptyset$, we get the usual notion of homomorphisms on term graphs (e.g. Barendsen [11]). The Δ-nodes can be thought of as holes in the term graphs that can be filled with other term graphs. For example, if we have a distinguished set of variable symbols $\mathcal{V} \subseteq \Sigma^{(0)}$, we can use $\mathcal{V}$-homomorphisms to formalise the matching step of term graph rewriting, which requires the instantiation of variables.

Example 4.8. Figure 2 depicts two functions ϕ and ψ. Whereas ϕ is a homomorphism, the function ψ is not a homomorphism since, for example, the node labelled a in g_3 is mapped to a node labelled h in g_3. Nevertheless, ψ is a $\{a, b\}$-homomorphism. Note that Δ-homomorphisms may introduce additional sharing in the target term graph by mapping several nodes in the source to the same node in the target.

Proposition 4.9 (Δ-homomorphism preorder)**.** *Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a category that is a preorder, i.e. there is at most one Δ-homomorphism from one term graph to another.*

Proof. The identity Δ-homomorphism is obviously the identity mapping on the set of nodes. Moreover, an easy equational reasoning reveals that the composition of two Δ-homomorphisms is again a Δ-homomorphism. Associativity of this composition is obvious as Δ-homomorphisms are functions.

To show that the category is a preorder, assume that there are two Δ-homomorphisms $\phi_1, \phi_2\colon g \to_\Delta h$. We prove that $\phi_1 = \phi_2$ by showing that $\phi_1(n) = \phi_2(n)$ for all $n \in N^g$ by induction on the depth of n in g.

Let $\mathsf{depth}_g(n) = 0$, i.e. $n = r^g$. By the root condition for ϕ, we have that $\phi_1(r^g) = r^h = \phi_2(r^g)$. Let $\mathsf{depth}_g(n) = d > 0$. Then n has a position $\pi \cdot \langle i \rangle$ in g such that $\mathsf{depth}_g(n') < d$ for $n' = \mathsf{node}_g(\pi)$. Hence, we can employ the induction hypothesis for n'. Moreover, since n' has at least one successor node, viz. n, it cannot be labelled with a nullary symbol and a fortiori not with a symbol in Δ. Therefore, the Δ-homomorphisms ϕ_1 and ϕ_2 are homomorphic in n' and we can thus reason as follows:

$$\begin{aligned} \phi_1(n) &= \mathsf{suc}_i^h(\phi_1(n')) && \text{(successor condition for } \phi_1) \\ &= \mathsf{suc}_i^h(\phi_2(n')) && \text{(ind. hyp.)} \\ &= \phi_2(n) && \text{(successor condition for } \phi_2) \end{aligned}$$

□

As a consequence, whenever there are two Δ-homomorphisms $\phi\colon g \to_\Delta h$ and $\psi\colon h \to_\Delta g$, they are inverses of each other, i.e. Δ-*isomorphisms*. If two term graphs are Δ-*isomorphic*, we write $g \cong_\Delta h$.

For the two special cases $\Delta = \emptyset$ and $\Delta = \{\sigma\}$, we write $\phi\colon g \to h$ respectively $\phi\colon g \to_\sigma h$ instead of $\phi\colon g \to_\Delta h$ and call ϕ a *homomorphism* respectively a σ-*homomorphism*. The same convention applies to Δ-isomorphisms.

The structure of positions permits a convenient characterisation of Δ-homomorphisms:

Lemma 4.10 (characterisation of Δ-homomorphisms)**.** *For $g, h \in \mathcal{G}^\infty(\Sigma)$, a function $\phi\colon N^g \to N^h$ is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff the following holds for all $n \in N^g$:*

$$\begin{array}{c} \textit{(a)}\ \mathcal{P}_g(n) \subseteq \mathcal{P}_h(\phi(n)), \quad \textit{and} \\ \textit{(b)}\ \mathsf{lab}^g(n) \notin \Delta \quad \Longrightarrow \quad \mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n)). \end{array}$$

Proof. For the "only if" direction, assume that $\phi\colon g \to_\Delta h$. (b) is the labelling condition and is therefore satisfied by ϕ. To establish (a), we show the equivalent statement

$$\forall \pi \in \mathcal{P}(g).\, \forall n \in N^g.\, \pi \in \mathcal{P}_g(n) \implies \pi \in \mathcal{P}_h(\phi(n))$$

We do so by induction on the length of π: if $\pi = \langle\rangle$, then $\pi \in \mathcal{P}_g(n)$ implies $n = r^g$. By the root condition, we have $\phi(r^g) = r^h$ and, therefore, $\pi = \langle\rangle \in \phi(r^g)$. If $\pi = \pi' \cdot \langle i \rangle$, then let $n' = \mathsf{node}_g(\pi')$. Consequently, $\pi' \in \mathcal{P}_g(n')$ and, by induction hypothesis, $\pi' \in \mathcal{P}_h(\phi(n'))$. Since $\pi = \pi' \cdot \langle i \rangle$, we have $\mathsf{suc}_i^g(n') = n$. By the successor condition we can conclude $\phi(n) = \mathsf{suc}_i^h(\phi(n'))$. This and $\pi' \in \mathcal{P}_h(\phi(n'))$ yields that $\pi' \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n))$.

For the "if" direction, we assume (a) and (b). The labelling condition follows immediately from (b). For the root condition, observe that since $\langle\rangle \in \mathcal{P}_g(r^g)$, we also have $\langle\rangle \in \mathcal{P}_h(\phi(r^g))$. Hence, $\phi(r^g) = r^h$. In order to show the successor condition, let $n, n' \in N^g$ and $0 \leq i < \mathsf{ar}_g(n)$ such that $\mathsf{suc}_i^g(n) = n'$. Then there is a position $\pi \in \mathcal{P}_g(n)$ with $\pi \cdot \langle i \rangle \in \mathcal{P}_g(n')$. By (a), we can conclude that $\pi \in \mathcal{P}_h(\phi(n))$ and $\pi \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n'))$ which implies that $\mathsf{suc}_i^h(\phi(n)) = \phi(n')$. □

By Proposition 4.9, there is at most one Δ-homomorphism between two term graphs. The lemma above uniquely defines this Δ-homomorphism: if there is a Δ-homomorphism from g to h, it is defined by $\phi(n) = n'$, where n' is the unique node $n' \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(n')$. Moreover, while it is not true for arbitrary Δ-homomorphisms, we have that homomorphisms are surjective.

Lemma 4.11 (homomorphisms are surjective)**.** *Every homomorphism $\phi\colon g \to h$, with $g, h \in \mathcal{G}^\infty(\Sigma)$, is surjective.*

Proof. Follows from an easy induction on the depth of the nodes in h. □

The $\{a, b\}$-homomorphism illustrated in Figure 2b, shows that the above lemma does not hold for Δ-homomorphisms in general.

4.2 Isomorphisms & Isomorphism Classes

When dealing with term graphs, in particular, when studying term graph transformations, we do not want to distinguish between isomorphic term graphs. Distinct but isomorphic term graphs do only differ in the naming of nodes and are thus an unwanted artifact of the definition of term graphs. In this way, equality up to isomorphism is similar to α-equivalence of λ-terms and has to be dealt with.

In this section, we shall characterise isomorphisms and, more generally, Δ-isomorphisms. From this we derive two canonical representations of isomorphism classes of term graphs. One is simply a subclass of the class of term graphs while the other one is based on the structure provided by the positions of term graphs. The relevance of the former representation is derived from the fact that we still have term graphs that can be easily manipulated whereas the latter is more technical and will be helpful for constructing term graphs up to isomorphism.

Note that a bijective Δ-homomorphism is not necessarily a Δ-isomorphism. To realise this, consider two term graphs g, h, each with one node only. Let the node in g be labelled with a and the node in h with b then the only possible a-homomorphism from g to h is clearly a bijection but not an a-isomorphism. On the other hand, bijective homomorphisms indeed are isomorphisms.

Lemma 4.12 (bijective homomorphisms are isomorphisms)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon\ g \to h$. Then the following are equivalent*

(a) ϕ is an isomorphism.

(b) ϕ is bijective.

(c) ϕ is injective.

Proof. The implication (a) $\Rightarrow$ (b) is trivial. The equivalence (b) $\Leftrightarrow$ (c) follows from Lemma 4.11. For the implication (b) $\Rightarrow$ (a), consider the inverse ϕ^{-1} of ϕ. We need to show that ϕ^{-1} is a homomorphism from h to g. The root condition follows immediately from the root condition for ϕ. Similarly, an easy equational reasoning reveals that ϕ^{-1} is homomorphic in N^h since ϕ is homomorphic in all $n \in N^g$. □

From the characterisation of Δ-homomorphisms in Lemma 4.10, we immediately obtain a characterisation of Δ-isomorphisms as follows:

Lemma 4.13 (characterisation of Δ-isomorphisms)**.** *For all $g, h \in \mathcal{G}^\infty(\Sigma)$, a function $\phi\colon\ N^g \to N^h$ is a Δ-isomorphism iff for all $n \in N^g$*

(a) $\mathcal{P}_h(\phi(n)) = \mathcal{P}_g(n)$, and

(b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ or $\mathsf{lab}^g(n), \mathsf{lab}^h(\phi(n)) \in \Delta$.

Proof. Immediate consequence of Lemma 4.10 and Proposition 4.9. □

Note that whenever Δ is a singleton set, the condition $\mathsf{lab}^g(n), \mathsf{lab}^h(\phi(n)) \in \Delta$ in the above lemma implies $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$. Therefore, we obtain the following corollary:

Corollary 4.14 (σ-isomorphism = isomorphism)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\sigma \in \Sigma^{(0)}$, we have $g \cong h$ iff $g \cong_\sigma h$.*

Note that the above equivalence does not hold for Δ-homomorphisms with more than one symbol in Δ: consider the term graphs $g = a$ and $h = b$ consisting of a single node labelled a respectively b. While g and h are Δ-isomorphic for $\Delta = \{a, b\}$, they are not isomorphic.

4.2.1 Canonical Term Graphs

From the Lemmas 4.12 and 4.13 we learned that isomorphisms between term graphs are bijections that preserve and reflect the positions as well as the labelling of each node. These findings motivate the following definition of canonical term graphs as candidates for representatives of isomorphism classes:

Definition 4.15 (canonical term graphs)**.** A term graph g is called *canonical* if $n = \mathcal{P}_g(n)$ holds for each $n \in N^g$. That is, each node is the set of its positions in the term graph. The set of all (finite) canonical term graphs over Σ is denoted $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ (respectively $\mathcal{G}_\mathcal{C}(\Sigma)$). Given a term graph $h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$, its *canonical representative* $\mathcal{C}(h)$ is the canonical term graph given by

$$N^{\mathcal{C}(h)} = \{\mathcal{P}_h(n) \mid n \in N\} \qquad r^{\mathcal{C}(h)} = \mathcal{P}_h(r)$$
$$\mathsf{lab}^{\mathcal{C}(h)}(\mathcal{P}_h(n)) = \mathsf{lab}^h(n) \quad \text{for all } n \in N$$
$$\mathsf{suc}_i^{\mathcal{C}(h)}(\mathcal{P}_h(n)) = \mathcal{P}_h(\mathsf{suc}_i^h(n)) \quad \text{for all } n \in N, 0 \leq i < \mathsf{ar}_h(n)$$

The above definition follows a well-known approach to obtain, for each term graph g, a canonical representative $\mathcal{C}(g)$ [31]. One can easily see that $\mathcal{C}(g)$ is a well-defined canonical term graph. With this definition we indeed capture a notion of canonical representatives of isomorphism classes:

Proposition 4.16 (canonical term graphs are isomorphism class representatives)**.** *Given $g \in \mathcal{G}^\infty(\Sigma)$, the term graph $\mathcal{C}(g)$ canonically represents the equivalence class $[g]_\cong$. More precisely, it holds that*

(i) $[g]_\cong = [\mathcal{C}(g)]_\cong$, *and* *(ii)* $[g]_\cong = [h]_\cong$ *iff* $\mathcal{C}(g) = \mathcal{C}(h)$.

In particular, we have, for all canonical term graphs g, h, that $g = h$ iff $g \cong h$.

Proof. Straightforward consequence of Lemma 4.13. □

4.2.2 Labelled Quotient Trees

Intuitively, term graphs can be thought of as "terms with sharing", i.e. terms in which occurrences of the same subterm may be identified. The representation of isomorphic term graphs as *labelled quotient trees*, which we shall study in this section, makes use of and formalises this intuition. To this end, we introduce an equivalence relation on the positions of a term graph that captures the sharing in a term graph:

Definition 4.17 (aliasing positions)**.** Given a term graph g and two positions $\pi_1, \pi_2 \in \mathcal{P}(g)$, we say that π_1 and π_2 *alias each other* in g, denoted $\pi_1 \sim_g \pi_2$, if $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$.

One can easily see that the thus defined relation $\sim_g$ on $\mathcal{P}(g)$ is an equivalence relation. Moreover, the partition on $\mathcal{P}(g)$ induced by $\sim_g$ is simply the set $\{\mathcal{P}_g(n) \mid n \in N^g\}$ that contains the sets of positions of nodes in g.

Example 4.18. For the term graph g_2 illustrated in Figure 2a, we have that $\langle 0,0\rangle \sim_{g_2} \langle 1\rangle$ as both $\langle 0,0\rangle$ and $\langle 1\rangle$ are positions of the a-node in g_2. For the term graph g_4 in Figure 2b, $\langle\rangle \sim_{g_4} \langle 1\rangle \sim_{g_4} \langle 1,1\rangle \sim_{g_4} \ldots$ as all finite sequences over 1 are positions of the f-node in g_4.

The characterisation of Δ-homomorphisms of Lemma 4.10 can be recast in terms of aliasing positions, which then yields the following characterisation of the *existence* of Δ-homomorphisms:

Lemma 4.19 (characterisation of Δ-homomorphisms)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, there is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff, for all $\pi, \pi' \in \mathcal{P}(g)$, we have*

$$\begin{array}{rcl} \textit{(a)}\ \pi \sim_g \pi' & \Longrightarrow & \pi \sim_h \pi',\ \textit{and} \\ \textit{(b)}\ g(\pi) \notin \Delta & \Longrightarrow & g(\pi) = h(\pi). \end{array}$$

Proof. For the "only if" direction, assume that ϕ is a Δ-homomorphism from g to h. Then we can use the properties (a) and (b) of Lemma 4.10, which we will refer to as (a') and (b') to avoid confusion. In order to show (a), assume $\pi \sim_g \pi'$. Then there is some node $n \in N^g$ with $\pi, \pi' \in \mathcal{P}_g(n)$. (a') yields $\pi, \pi' \in \phi(n)$ and, therefore, $\pi \sim_h \pi'$. To show (b), we assume some $\pi \in \mathcal{P}(g)$ with $g(\pi) \notin \Delta$. Then we can reason as follows:

$$g(\pi) = \mathsf{lab}^g(\mathsf{node}_g(\pi)) \overset{\text{(b')}}{=} \mathsf{lab}^h(\phi(\mathsf{node}_g(\pi))) \overset{\text{(a')}}{=} \mathsf{lab}^h(\mathsf{node}_h(\pi)) = h(\pi)$$

For the converse direction, assume that both (a) and (b) hold. Define the function $\phi\colon N^g \to N^h$ by $\phi(n) = m$ iff $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$ for all $n \in N^g$ and $m \in N^h$. To see that this is well-defined, we show at first that, for each $n \in N^g$, there is at most one $m \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$. Suppose there is another node $m' \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m')$. Since $\mathcal{P}_g(n) \neq \emptyset$, this implies $\mathcal{P}_h(m) \cap \mathcal{P}_h(m') \neq \emptyset$. Hence, $m = m'$. Secondly, we show that there is at least one such node m. Choose some $\pi^* \in \mathcal{P}_g(n)$. Since then $\pi^* \sim_g \pi^*$ and, by (a), also $\pi^* \sim_h \pi^*$ holds, there is some $m \in N^h$ with $\pi^* \in \mathcal{P}_h(m)$. For each $\pi \in \mathcal{P}_g(n)$, we have $\pi^* \sim_g \pi$ and, therefore, $\pi^* \sim_h \pi$ by (a). Hence, $\pi \in \mathcal{P}_h(m)$. So we know that ϕ is well-defined. By construction, ϕ satisfies (a'). Moreover, because of (b), it is also easily seen to satisfy (b'). Hence, ϕ is a homomorphism from g to h. □

Intuitively, Clause (a) states that h has at least as much sharing of nodes as g has, whereas Clause (b) states that h has at least the same non-Δ-labelling as g. In this sense, the above characterisation confirms the intuition about Δ-homomorphisms that we mentioned in Example 4.8, viz. Δ-homomorphisms may only introduce sharing and relabel Δ-nodes. This can be observed in the two Δ-homomorphisms illustrated in Figure 2.

From the above characterisations of the existence of Δ-homomorphisms, we can easily derive the following characterisation of Δ-isomorphisms using the uniqueness of Δ-homomorphisms between two term graphs:

Lemma 4.20 (characterisation of Δ-isomorphisms)**.** *For all $g, h \in \mathcal{G}^\infty(\Sigma)$, we have $g \cong_\Delta h$ iff*

$$\text{(a) } \sim_g = \sim_h, \text{ and} \qquad \text{(b) } g(\pi) = h(\pi) \text{ or } g(\pi), h(\pi) \in \Delta \quad \text{for all } \pi \in \mathcal{P}(g).$$

Proof. Immediate consequence of Lemma 4.19 and Proposition 4.9. □

Remark 4.21. Δ-homomorphisms can be naturally lifted to the set of isomorphism classes $\mathcal{G}^\infty(\Sigma)/_{\cong}$: we say that two Δ-homomorphisms $\phi\colon g \to_\Delta h$, $\phi'\colon g' \to_\Delta h'$, are isomorphic, written $\phi \cong \phi'$ iff there are isomorphisms $\psi_1\colon g \cong g'$ and $\psi_2\colon h \cong h'$ such that $\psi_2 \circ \phi = \phi' \circ \psi_1$. Given a Δ-homomorphism $\phi\colon g \to_\Delta h$ in $\mathcal{G}^\infty(\Sigma)$, $[\phi]_\cong\colon [g]_\cong \to_\Delta [h]_\cong$ is a Δ-homomorphism in $\mathcal{G}^\infty(\Sigma)/_{\cong}$. These Δ-homomorphisms then form a category which can easily be show to be isomorphic to the category of Δ-homomorphisms on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ via the mapping $[\cdot]_\cong$.

Lemma 4.20 has shown that term graphs can be characterised up to isomorphism by only giving the equivalence $\sim_g$ and the labelling $g(\cdot)\colon \pi \mapsto g(\pi)$ of the involved term graphs. This observation gives rise to the following definition:

Definition 4.22 (labelled quotient trees)**.** A *labelled quotient tree* over signature Σ is a triple $(P, l, \sim)$ consisting of a non-empty set $P \subseteq \mathbb{N}^*$, a function $l\colon P \to \Sigma$, and an equivalence relation $\sim$ on P that satisfies the following conditions for all $\pi, \pi' \in \mathbb{N}^*$ and $i \in \mathbb{N}$:

$$\pi \cdot \langle i \rangle \in P \implies \pi \in P \quad \text{and} \quad i < \mathsf{ar}(l(\pi)) \qquad \text{(reachability)}$$

$$\pi \sim \pi' \implies \begin{cases} l(\pi) = l(\pi') & \text{and} \\ \pi \cdot \langle i \rangle \sim \pi' \cdot \langle i \rangle & \text{for all } \ i < \mathsf{ar}(l(\pi)) \end{cases} \qquad \text{(congruence)}$$

In other words, a labelled quotient tree $(P, l, \sim)$ is a ranked tree domain P together with a congruence $\sim$ on it and a labelling function $l\colon P/_\sim \to \Sigma$ that honours the rank. Also note that since P must be non-empty, the reachability condition implies that $\langle\rangle \in P$.

Example 4.23. The term graph g_2 depicted in Figure 2a is represented up to isomorphism by the labelled quotient tree $(P, l, \sim)$ with $P = \{\langle\rangle, \langle 0\rangle, \langle 0, 0\rangle, \langle 1\rangle\}$, $l(\langle\rangle) = f$, $l(\langle 0\rangle) = h$, $l(\langle 0, 0\rangle) = l(\langle 1\rangle) = a$ and $\sim$ the least equivalence relation on P with $\langle 0, 0\rangle \sim \langle 1\rangle$.

The following lemma confirms that labelled quotient trees uniquely characterise any term graph up to isomorphism:

Lemma 4.24 (labelled quotient trees are canonical)**.** *Each term graph $g \in \mathcal{G}^\infty(\Sigma)$ induces a* canonical labelled quotient tree *$(\mathcal{P}(g), g(\cdot), \sim_g)$ over Σ. Vice versa, for each labelled quotient tree $(P, l, \sim)$ over Σ there is a unique canonical term graph $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ whose canonical labelled quotient tree is $(P, l, \sim)$, i.e. $\mathcal{P}(g) = P$, $g(\pi) = l(\pi)$ for all $\pi \in P$, and $\sim_g = \sim$.*

Proof. The first part is trivial: $(\mathcal{P}(g), g(\cdot), \sim_g)$ satisfies the conditions from Definition 4.22.

For the second part, let $(P, l, \sim)$ be a labelled quotient tree. Define the term graph $g = (N, \mathsf{lab}, \mathsf{suc}, r)$ by

$$\begin{array}{rrll} N = P/_\sim & \mathsf{lab}(n) = f & \text{iff} & \exists \pi \in n.\ l(\pi) = f \\ r = [\langle\rangle]_\sim & \mathsf{suc}_i(n) = n' & \text{iff} & \exists \pi \in n.\ \pi \cdot \langle i \rangle \in n' \end{array}$$

The functions lab and suc are well-defined due to the congruence condition satisfied by $(P, l, \sim)$. Since P is non-empty and closed under prefixes, it contains $\langle\rangle$. Hence, r is well-defined. Moreover, by the reachability condition, each node in N is reachable from the root node. An easy induction proof shows that $\mathcal{P}_g(n) = n$ for each node $n \in N$. Thus, g is a well-defined canonical term graph. The canonical labelled quotient tree of g is obviously $(P, l, \sim)$. Whenever there are two canonical term graphs with the same canonical labelled quotient tree $(P, l, \sim)$, they are isomorphic due to Lemma 4.20 and, therefore, have to be identical by Proposition 4.16. □

Labelled quotient trees provide a valuable tool for constructing canonical term graphs as we shall see. Nevertheless, the original graph representation remains convenient for practical purposes as it allows a straightforward formalisation of term graph rewriting and provides a finite representation of finite cyclic term graphs, which induce an infinite labelled quotient tree.

4.2.3 Terms, Term Trees & Unravelling

Before we continue, it is instructive to make the correspondence between terms and term graphs clear. First, note that, for each term tree t, the equivalence $\sim_t$ is the identity relation $\mathcal{I}_{\mathcal{P}(t)}$ on $\mathcal{P}(t)$, i.e. $\pi_1 \sim_t \pi_2$ iff $\pi_1 = \pi_2$. Consequently, we have the following one-to-one correspondence between canonical term *trees* and terms: each term $t \in \mathcal{T}^\infty(\Sigma)$ induces the canonical term tree given by the labelled quotient tree $(\mathcal{P}(t), t(\cdot), \mathcal{I}_{\mathcal{P}(t)})$. For example, the term tree depicted in Figure 1a corresponds to the term $f(a, h(a, b))$. We thus consider the set of terms $\mathcal{T}^\infty(\Sigma)$ as the subset of canonical term trees of $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$.

With this correspondence in mind, we can define the *unravelling* of a term graph g as the unique term t such that there is a homomorphism $\phi\colon t \to g$. The unravelling of cyclic term graphs yields infinite terms, e.g. in Figure 8 on page 345, the term h_ω is the unravelling of the term graph g_2. We use the notation $\mathcal{U}(g)$ for the unravelling of g.

5 A Rigid Partial Order on Term Graphs

In this section, we shall establish a partial order suitable for formalising convergence of sequences of canonical term graphs similarly to p-convergence on terms.

Recall that p-convergence in term rewriting systems is based on a partial order $\leq_\bot$ on the set $\mathcal{T}^\infty(\Sigma_\bot)$ of *partial* terms. The partial order $\leq_\bot$ instantiates occurrences of $\bot$ from left to right, i.e. $s \leq_\bot t$ iff t is obtained by replacing occurrences of $\bot$ in s by arbitrary terms in $\mathcal{T}^\infty(\Sigma_\bot)$.

Since we are considering term graph rewriting as a generalisation of term rewriting, our aim is to generalise the partial order $\leq_\bot$ on terms to term graphs.

That is, the partial order we are looking for should coincide with $\leq_\bot$ if restricted to term trees. Moreover, we also want to maintain the characteristic properties of the partial order $\leq_\bot$ when generalising to term graphs. The most important characteristic we are striving for is a complete semilattice structure in order to define p-convergence in terms of the limit inferior. Apart from that, we also want to maintain the intuition of the partial order $\leq_\bot$, viz. the intuition of information preservation, which $\leq_\bot$ captures on terms as we illustrated in Section 2. We will make this last guiding principle clearer as we go along.

Analogously to partial terms, we consider the class of *partial term graphs* simply as term graphs over the signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$. In order to generalise the partial order $\leq_\bot$ to term graphs, we need to formalise the instantiation of occurrences of $\bot$ in term graphs. Δ-homomorphisms, for $\Delta = \{\bot\}$ – or $\bot$-homomorphisms for short – provide the right starting point for that. A homomorphism $\phi\colon g \to h$ maps each node in g to a node in h while preserving the *local* structure of each node, viz. its labelling and its successors. In the case of a $\bot$-homomorphisms $\phi\colon g \to_\bot h$, the preservation of the labelling is suspended for nodes labelled $\bot$ thus allowing ϕ to instantiate each $\bot$-node in g with an arbitrary node in h.

Therefore, we shall use $\bot$-homomorphisms as the basis for generalising $\leq_\bot$ to canonical partial term graphs. This approach is based on the observation that $\bot$-homomorphisms characterise the partial order $\leq_\bot$ on terms. Considering terms as canonical term trees, we obtain the following equivalence:

$$s \leq_\bot t \iff \text{there is a } \bot\text{-homomorphism } \phi\colon s \to_\bot t.$$

Thus, $\bot$-homomorphisms constitute the ideal tool for defining a partial order on canonical partial term graphs that generalises $\leq_\bot$. In the following subsection, we shall explore different partial orders on canonical partial term graphs based on $\bot$-homomorphisms.

5.1 Partial Orders on Term Graphs

Consider the *simple partial order* $\leq_\bot^{\mathsf{S}}$ defined on term graphs as follows: $g \leq_\bot^{\mathsf{S}} h$ iff there is a $\bot$-homomorphism $\phi\colon g \to_\bot h$. This is a straightforward generalisation of the partial order $\leq_\bot$ to term graphs. In fact, this partial order forms a complete semilattice on $\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$ [9].

As we have explained in Section 2, p-convergence on terms is based on the ability of the partial order $\leq_\bot$ to capture *information preservation* between terms – $s \leq_\bot t$ means that t contains at least the same information as s does. The limit inferior – and thus p-convergence – comprises the accumulated information that eventually remains stable. Following the approach on terms, a partial order suitable as a basis for convergence for term graph rewriting, has to capture an appropriate notion of information preservation as well.

One has to keep in mind, however, that term graphs encode an additional dimension of information through *sharing* of nodes, i.e. the fact that nodes may have multiple positions. Since $\leq_\bot^{\mathsf{S}}$ specialises to $\leq_\bot$ on terms, it does preserve the information on the tree structure in the same way as $\leq_\bot$ does. The difficult part is to determine the right approach to the role of sharing.

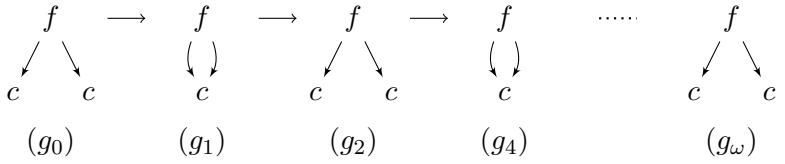


Figure 3: Limit inferior w.r.t. $\leq_\bot^\mathsf{S}$ in the presence of acyclic sharing.

Indeed, $\bot$-homomorphisms instantiate occurrences of $\bot$ and are thereby able to introduce new information. But while $\bot$-homomorphisms preserve the *local* structure of each node, they may change the *global* structure of a term graph by introducing sharing: for the term graphs g_0 and g_1 in Figure 3, we have an obvious $\bot$-homomorphism – in fact a homomorphism – $\phi\colon g_0 \to_\bot g_1$ and thus $g_0 \leq_\bot^\mathsf{S} g_1$.

There are at least two different ways to interpret the differences in g_0 and g_1. The first one dismisses $\leq_\bot^\mathsf{S}$ as a partial order suitable for our purposes: the term graphs g_0 and g_1 contain contradicting information. While in g_0 the two children of the f-node are distinct, they are identical in g_1. We will indeed follow this view in this paper and introduce a rigid partial order $\leq_\bot^\mathsf{R}$ that addresses this concern. There is, however, also a second view that does not see g_0 and g_1 in contradiction: both term graphs show the f-node with two successors, both of which are labelled with c. The term graph g_1 merely contains the additional piece of information that the two successor nodes of the f-node are identical. The simple partial order $\leq_\bot^\mathsf{S}$, which follows this view, is studied further in [9].

One consequence of the above behaviour of $\leq_\bot^\mathsf{S}$ is that total term graphs are not necessarily maximal w.r.t. $\leq_\bot^\mathsf{S}$, e.g. g_0 is total but not maximal. The second – more severe – consequence is that there can be no metric on total term graphs such that the limit w.r.t. that metric coincides with the limit inferior on total term graph. To see this consider the sequence $(g_i)_{i<\omega}$ of term graphs illustrated in Figure 3. Its limit inferior w.r.t. $\leq_\bot^\mathsf{S}$ is the total term graph g_ω. On the other hand, there is no metric w.r.t. which $(g_i)_{i<\omega}$ converges since the sequence alternates between two distinct term graphs. That is, the correspondence between metric and partial order convergence that we know from term rewriting, cf. Theorem 3.3, is impossible.

To avoid the introduction of sharing, we need to consider $\bot$-homomorphisms that preserve the structure of term graphs more rigidly, i.e. not only locally. Recall that by Lemma 4.24, the structure of a term graph is essentially given by the positions of nodes and their labelling. Labellings are already taken into consideration by $\bot$-homomorphisms. Thus, we can define a partial order $\leq_\bot^\mathsf{P}$ that preserves the structure of term graphs as follows: $g \leq_\bot^\mathsf{P} h$ iff there is a $\bot$-homomorphism $\phi\colon g \to_\bot h$ with $\mathcal{P}_h(\phi(n)) = \mathcal{P}_g(n)$ for all $n \in N^g$ with $\mathsf{lab}^g(n) \neq \bot$. While this would again yield a complete semilattice, it is unfortunately too restrictive. For example, consider the sequence of term graphs $(g_i)_{i<\omega}$ depicted in Figure 4. Due to the cycle, we have for each term graph g_i that $\bot$ is the only term graph strictly smaller than g_i w.r.t. $\leq_\bot^\mathsf{P}$. The reason for this is the fact that the only way to maintain the positions of the root node of the term graph g_i is to keep all nodes of the cycle in g_i. Hence, in order to obtain a term graph

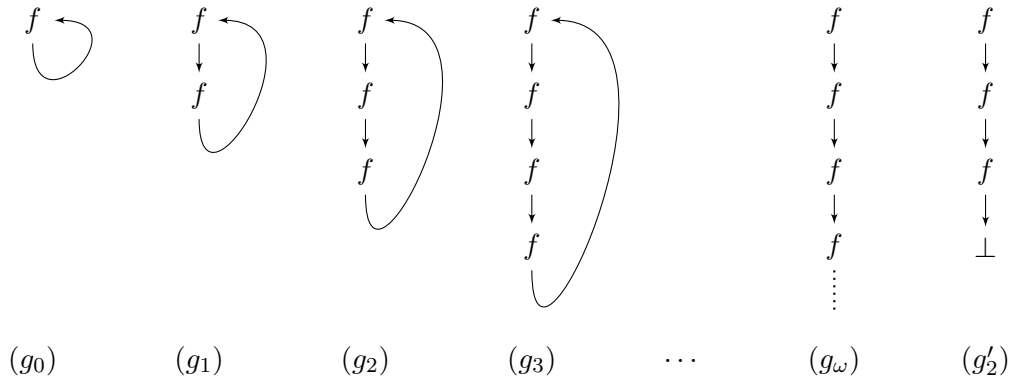


Figure 4: Varying acyclic sharing.

h with $h \leq^{\mathsf{P}}_{\bot} g_i$, we have to either keep the whole term graph g_i or collapse it completely, yielding $\bot$. For example, we neither have $g'_2 \leq^{\mathsf{P}}_{\bot} g_2$ nor $g'_2 \leq^{\mathsf{P}}_{\bot} g_3$ for the term graph g'_2 illustrated in Figure 4. As a consequence, the limit inferior of the sequence $(g_i)_{i<\omega}$ is $\bot$ and not the expected term graph g_ω.

The fact that the root nodes g_2 and g'_2 have different sets of positions is solely caused by the edge to the root node of g_2 that comes from *below* and thus closes a *cycle*. Even though the edge occurs below the root node, it affects its positions. Cutting off that edge, like in g'_2, changes the sharing. As a consequence, in the complete semilattice $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{P}}_{\bot})$, we do not obtain the intuitively expected convergence behaviour depicted in Figure 8c on page 345.

This observation suggests that we should only consider the *upward structure* of each node, ignoring the sharing that is caused by edges occurring *below* a node. We will see that by restricting our attention to *acyclic positions*, we indeed obtain the desired properties for a partial order on term graphs.

Recall that a position π in a term graph g is called cyclic iff there are positions π_1, π_2 with $\pi_1 < \pi_2 \leq \pi$ such that $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$, i.e. π passes a node twice. Otherwise it is called acyclic. We will use the notation $\mathcal{P}^a(g)$ for the set of all acyclic positions in g, and $\mathcal{P}^a_g(n)$ for the set of all acyclic positions of a node n in g. That is, $\mathcal{P}^a(g)$ is the set of positions in g that pass each node in g at most once. Clearly, every node has at least one acyclic position, i.e. $\mathcal{P}^a_g(n)$ is a non-empty set.

Definition 5.1 (rigidity)**.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$ and $g, h \in \mathcal{G}^\infty(\Sigma)$ such that $\phi\colon\ g \to_\Delta h$.

(i) Given $n \in N^g$, ϕ is said to be *rigid* in n if it satisfies the equation

$$\mathcal{P}^a_g(n) = \mathcal{P}^a_h(\phi(n)) \qquad \text{(rigid)}$$

(ii) ϕ is called a *rigid* Δ-homomorphism if it is rigid in all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$.

Proposition 5.2 (category of rigid Δ-homomorphisms)**.** *The rigid Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a subcategory of the category of Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$.*

Proof. Straightforward. □

Note that, for each node n in a term graph g, the positions in $\mathcal{P}_g^a(n)$ are minimal positions of n w.r.t. the prefix order. Rigid $\bot$-homomorphisms thus preserve the upward structure of each non-$\bot$-node and, therefore, provide the desired structure for a partial order that captures information preservation on term graphs:

Definition 5.3 (rigid partial order $\leq_\bot^{\mathsf{R}}$)**.** For every $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$, define $g \leq_\bot^{\mathsf{R}} h$ iff there is a rigid $\bot$-homomorphism $\phi\colon g \to_\bot h$.

Proposition 5.4 (partial order $\leq_\bot^{\mathsf{R}}$)**.** *The relation $\leq_\bot^{\mathsf{R}}$ is a partial order on $\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$.*

Proof. Reflexivity and transitivity of $\leq_\bot^{\mathsf{R}}$ follow immediately from Proposition 5.2. For antisymmetry, assume $g \leq_\bot^{\mathsf{R}} h$ and $h \leq_\bot^{\mathsf{R}} g$. By Proposition 4.9, this implies $g \cong_\bot h$. Corollary 4.14 then yields that $g \cong h$. Hence, according to Proposition 4.16, $g = h$. □

Example 5.5. Figure 8c on page 345 shows a sequence $(h_\iota)_{\iota<\omega}$ of term graphs and its limit inferior h_ω in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$: a cyclic list structure is repeatedly rewritten by inserting an element b in front of the a. We can see that in each step the newly inserted b (including the additional $::$-node) remains unchanged afterwards. In terms of positions, however, each of the nodes changes in each step since the length of the cycle in the term graph grows with each step. Since this affects only cyclic positions, we still get the following sequence $(\bigsqcap_{\beta\leq\iota<\omega} h_\iota)_{\beta<\omega}$ of canonical term trees:

$$\langle \bot :: \bot, b :: \bot :: \bot, b :: b :: \bot :: \bot, \ldots \rangle$$

The least upper bound of this sequence $(\bigsqcap_{\beta\leq\iota<\omega} h_\iota)_{\beta<\omega}$ and thus the limit inferior of $(h_\iota)_{\iota<\omega}$ is the infinite canonical term tree $h_\omega = b :: b :: b :: \ldots$. Since the cycle changes in each step and is thus cut through in each element of $(\bigsqcap_{\beta\leq\iota<\omega} h_\iota)_{\beta<\omega}$, the limit inferior has no cycles at all.

Note that we do not have this intuitively expected convergence behaviour for the partial order $\leq_\bot^{\mathsf{P}}$ based on positions: since the length of the cycle grows along the sequence $(h_\iota)_{\iota<\omega}$, we have that the set of positions of the root nodes changes constantly. Hence, the limit inferior of $(h_\iota)_{\iota<\omega}$ in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{P}})$ is $\bot$.

The partial order $\leq_\bot^{\mathsf{R}}$ based on rigid $\bot$-homomorphisms is defined in a rather non-local fashion as the definition of rigidity uses the set of *all* acyclic positions. This poses the question whether there is a more natural definition of a suitable partial order. One such candidate is the partial order $\leq_\bot^{\mathsf{I}}$, which uses injectivity in order to restrict the introduction of sharing: $g \leq_\bot^{\mathsf{I}} h$ iff there is a $\bot$-homomorphism $\phi\colon g \to_\bot h$ that is injective on non-$\bot$-nodes, i.e. $\phi(n) = \phi(m)$ and $\mathsf{lab}^g(n), \mathsf{lab}^g(m) \neq \bot$ implies $n = m$. While this yields indeed a cpo on $\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$, we do not get a complete semilattice. To see this, consider Figure 5. The two term graphs g_3, g_4 are two distinct maximal lower bounds of the two term graphs g_1, g_2 w.r.t. the partial order $\leq_\bot^{\mathsf{I}}$. Hence, the set $\{g_1, g_2\}$ does not have a greatest lower bound in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{I}})$, which is therefore not a complete

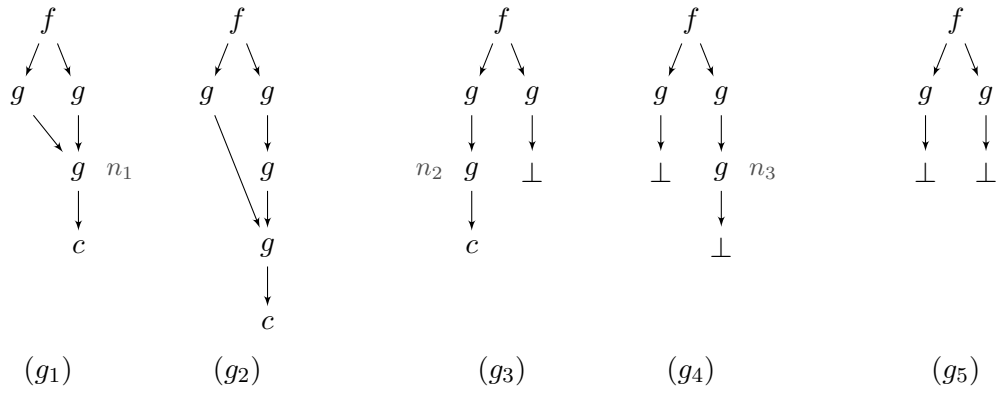


Figure 5: Term graphs g_1, g_2 with maximal lower bounds g_3, g_4 w.r.t. $\leq_\bot^{\mathsf{I}}$.

semilattice. The same phenomenon occurs if we consider a partial order derived from $\bot$-homomorphisms that are injective on all nodes.

The rigid partial order $\leq_\bot^{\mathsf{R}}$ resolves the issue of $\leq_\bot^{\mathsf{I}}$ illustrated in Figure 5: g_3 and g_4 are not lower bounds of g_1 and g_2 w.r.t. $\leq_\bot^{\mathsf{R}}$. The (unique) $\bot$-homomorphism from g_3 to g_1 is not rigid as it maps the node n_2 to n_1 and $\mathcal{P}^a_{g_3}(n_2) = \{\langle 0,0\rangle\}$ whereas $\mathcal{P}^a_{g_1}(n_1) = \{\langle 0,0\rangle, \langle 1,0\rangle\}$. Hence, $g_3 \not\leq_\bot^{\mathsf{R}} g_1$. Likewise, $g_4 \not\leq_\bot^{\mathsf{R}} g_1$ as the (unique) $\bot$-homomorphism from g_4 to g_1 maps n_3 to n_1, which again have different acyclic positions. We do find, however, a greatest lower bound of g_1 and g_2 w.r.t. $\leq_\bot^{\mathsf{R}}$, viz. g_5.

5.2 The Rigid Partial Order

In the remainder of this section, we will study the rigid partial order $\leq_\bot^{\mathsf{R}}$. In particular, we shall give a characterisation of rigidity in terms of labelled quotient trees analogous to Lemma 4.19, show that $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$ forms a complete semilattice, illustrate the resulting mode of convergence, and give a characterisation of term graphs that are maximal w.r.t. $\leq_\bot^{\mathsf{R}}$.

The partial order $\leq_\bot^{\mathsf{I}}$, derived from injective $\bot$-homomorphisms, failed to form a complete semilattice, which is why we abandoned that approach. The following lemma shows that rigidity is, in fact, a stronger property than injectivity on non-Δ-nodes. Hence, $\leq_\bot^{\mathsf{R}}$ is a restriction of $\leq_\bot^{\mathsf{I}}$.

Lemma 5.6 (rigid Δ-homomorphisms are injective for non-Δ-nodes)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon g \to_\Delta h$ rigid. Then ϕ is injective for all non-Δ-nodes in g. That is, for two nodes $n, m \in N^g$ with $\mathsf{lab}^g(n), \mathsf{lab}^g(m) \notin \Delta$ we have that $\phi(n) = \phi(m)$ implies $n = m$.*

Proof. Let $n, m \in N^g$ with $\mathsf{lab}^g(n), \mathsf{lab}^g(m) \notin \Delta$ and $\phi(n) = \phi(m)$. Since ϕ is rigid, it is rigid in n and m. That is, in particular we have $\mathcal{P}^a_h(\phi(n)) \subseteq \mathcal{P}_g(n)$ and $\mathcal{P}^a_h(\phi(m)) \subseteq \mathcal{P}_g(m)$. Moreover, because $\mathcal{P}^a_h(\phi(n)) = \mathcal{P}^a_h(\phi(m)) \neq \emptyset$, we can conclude that $\mathcal{P}_g(n) \cap \mathcal{P}_g(m) \neq \emptyset$ and, therefore, $m = n$. □

5.2.1 Characterising Rigidity

The goal of this subsection is to give a characterisation of rigidity in terms of labelled quotient trees. We will then combine this characterisation with Lemma 4.19 to obtain a characterisation of the partial order $\leq_{\perp}^{\mathsf{R}}$.

The following lemma provides a characterisation of rigid Δ-homomorphisms that reduces the proof obligations necessary to show that a Δ-homomorphism is rigid.

Lemma 5.7 (rigidity)**.** *Let $g, h \in \mathcal{G}^{\infty}(\Sigma)$, $\phi\colon g \to_{\Delta} h$. Then ϕ is rigid iff $\mathcal{P}_h^a(\phi(n)) \subseteq \mathcal{P}_g(n)$ for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$.*

Proof. The "only if" direction is trivial. For the "if" direction, suppose that ϕ satisfies $\mathcal{P}_h^a(\phi(n)) \subseteq \mathcal{P}_g(n)$ for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$. In order to prove that ϕ is rigid, we will show that $\mathcal{P}_h^a(\phi(n)) = \mathcal{P}_g^a(n)$ holds for each $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$.

We first show the inclusion $\mathcal{P}_h^a(\phi(n)) \subseteq \mathcal{P}_g^a(n)$. For this purpose, let $\pi \in \mathcal{P}_h^a(\phi(n))$. Due to the hypothesis, this implies that $\pi \in \mathcal{P}_g(n)$. Now suppose that π is cyclic in g, i.e. there are two positions π_1, π_2 of a node $m \in N^g$ with $\pi_1 < \pi_2 \leq \pi$. By Lemma 4.10, we can conclude that $\pi_1, \pi_2 \in \mathcal{P}_h(\phi(m))$. This is a contradiction to the assumption that π is acyclic in h. Hence, $\pi \in \mathcal{P}_g^a(n)$.

For the other inclusion, assume some $\pi \in \mathcal{P}_g^a(n)$. Using Lemma 4.10 we obtain that $\pi \in \mathcal{P}_h(\phi(n))$. It remains to be shown that π is acyclic in h. Suppose that this is not true, i.e. there are two positions π_1, π_2 of a node $m \in N^h$ with $\pi_1 < \pi_2 \leq \pi$. Note that since $\pi \in \mathcal{P}(g)$, also $\pi_1, \pi_2 \in \mathcal{P}(g)$. Let $m_i = \mathsf{node}_g(\pi_i)$, $i = 1, 2$. According to Lemma 4.10, we have that $\phi(m_1) = m = \phi(m_2)$. Moreover, observe that $g(\pi_1), g(\pi_2) \notin \Delta$: $g(\pi_1)$ cannot be a nullary symbol because $\pi_1 < \pi \in \mathcal{P}(g)$. The same argument applies for the case that $\pi_2 < \pi$. If this is not the case, then $\pi_2 = \pi$ and $g(\pi) \notin \Delta$ follows from the assumption that $\mathsf{lab}^g(n) \notin \Delta$. Thus, we can apply Lemma 5.6 to conclude that $m_1 = m_2$. Consequently, π is cyclic in g, which contradicts the assumption. Hence, $\pi \in \mathcal{P}_h^a(\phi(n))$. □

From the above lemma we learn that Δ-isomorphisms are also rigid Δ-homomorphisms.

Corollary 5.8 (Δ-isomorphisms are rigid)**.** *Let $g, h \in \mathcal{G}^{\infty}(\Sigma)$. If $\phi\colon g \cong_{\Delta} h$, then ϕ is a rigid Δ-homomorphism.*

Proof. This follows from Lemma 4.13 and Lemma 5.7. □

Similarly to Lemma 4.19, we provide a characterisation of rigid Δ-homomorphisms in terms of labelled quotient trees:

Lemma 5.9 (characterisation of rigid Δ-homomorphisms)**.** *Given $g, h \in \mathcal{G}^{\infty}(\Sigma)$, a Δ-homomorphism $\phi\colon g \to_{\Delta} h$ is rigid iff*

$$\pi \sim_h \pi' \quad \Longrightarrow \quad \pi \sim_g \pi' \quad \text{for all } \pi \in \mathcal{P}(g) \text{ with } g(\pi) \notin \Delta \text{ and } \pi' \in \mathcal{P}^a(h).$$

Proof. For the "only if" direction, assume that ϕ is rigid. Moreover, let $\pi \in \mathcal{P}(g)$ with $g(\pi) \notin \Delta$ and $\pi' \in \mathcal{P}^a(h)$ such that $\pi \sim_h \pi'$, and let $n = \mathsf{node}_g(\pi)$. By applying Lemma 4.10, we get that $\pi \in \mathcal{P}_h(\phi(n))$. Because of $\pi \sim_h \pi'$, also

$\pi' \in \mathcal{P}_h(\phi(n))$. Since, according to the assumption, π' is acyclic in h, we know in particular that $\pi' \in \mathcal{P}^a_h(\phi(n))$. Since ϕ is rigid and $\mathsf{lab}^g(n) \notin \Delta$, we know that ϕ is rigid in n which yields that $\pi' \in \mathcal{P}_g(n)$. Hence, $\pi \sim_g \pi'$.

For the converse direction, let $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$. We need to show that ϕ is rigid in n. Due to Lemma 5.7, it suffices to show that $\mathcal{P}^a_h(\phi(n)) \subseteq \mathcal{P}_g(n)$. Since $\mathcal{P}_g(n) \neq \emptyset$, we can choose some $\pi^* \in \mathcal{P}_g(n)$. Then, according to Lemma 4.10, also $\pi^* \in \mathcal{P}_h(\phi(n))$. Let $\pi \in \mathcal{P}^a_h(\phi(n))$. Then $\pi^* \sim_h \pi$ holds. Since π is acyclic in h and $g(\pi^*) \notin \Delta$, we can use the hypothesis to obtain that $\pi^* \sim_g \pi$ holds which shows that $\pi \in \mathcal{P}_g(n)$. □

Note that the above characterisation of rigidity is independent of the Δ-homomorphism at hand. This is expected since Δ-homomorphisms between a given pair of term graphs are unique.

By combining the above characterisation of rigidity with the corresponding characterisation of Δ-homomorphisms, we obtain the following compact characterisation of $\leq^{\mathsf{R}}_{\bot}$:

Corollary 5.10 (characterisation of $\leq^{\mathsf{R}}_{\bot}$)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$. Then $g \leq^R_\bot h$ iff the following conditions are met:*

(a) $\pi \sim_g \pi' \implies \pi \sim_h \pi'$ *for all* $\pi, \pi' \in \mathcal{P}(g)$

(b) $\pi \sim_h \pi' \implies \pi \sim_g \pi'$ *for all* $\pi \in \mathcal{P}(g)$ *with* $g(\pi) \in \Sigma$ *and* $\pi' \in \mathcal{P}^a(h)$

(c) $g(\pi) = h(\pi)$ *for all* $\pi \in \mathcal{P}(g)$ *with* $g(\pi) \in \Sigma$.

Proof. This follows immediately from Lemma 4.19 and Lemma 5.9. □

Note that for term trees (b) is always true and (a) follows from (c). Hence, on term trees, $\leq^{\mathsf{R}}_{\bot}$ is characterised by (c) alone. This observation shows that $\leq^{\mathsf{R}}_{\bot}$ is indeed a generalisation of $\leq_\bot$.

Corollary 5.11. *For all $s, t \in \mathcal{T}^\infty(\Sigma_\bot)$, we have that $s \leq^R_\bot t$ iff $s \leq_\bot t$.*

Proof. Follows from Corollary 5.10. □

5.2.2 Convergence

In the following, we shall show that $\leq^{\mathsf{R}}_{\bot}$ indeed forms a complete semilattice on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$. We begin by showing that it constitutes a complete partial order.

Theorem 5.12 ($\leq^{\mathsf{R}}_{\bot}$ is a cpo)**.** *The pair $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^R_\bot)$ forms a cpo. In particular, it has the least element $\bot$, and the least upper bound of a directed set G is given by the following labelled quotient tree $(P, l, \sim)$:*

$$P = \bigcup_{g \in G} \mathcal{P}(g) \qquad \sim = \bigcup_{g \in G} \sim_g \qquad l(\pi) = \begin{cases} f & \text{if } f \in \Sigma \text{ and } \exists g \in G.\ g(\pi) = f \\ \bot & \text{otherwise} \end{cases}$$

Proof. The least element of $\leq^{\mathsf{R}}_{\bot}$ is obviously $\bot$. Hence, it remains to be shown that each directed subset G of $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$ has a least upper bound w.r.t. $\leq^{\mathsf{R}}_{\bot}$. To this end, we show that the canonical term graph $\overline{g}$ given by the labelled quotient tree

$(P, l, \sim)$ described above is indeed the lub of G. We will make extensive use of Corollary 5.10 to do so. Therefore, we write (a), (b), (c) to refer to corresponding conditions of Corollary 5.10.

At first we need to show that l is indeed well-defined. For this purpose, let $g_1, g_2 \in G$ and $\pi \in \mathcal{P}(g_1) \cap \mathcal{P}(g_2)$ with $g_1(\pi), g_2(\pi) \in \Sigma$. Since G is directed, there is some $g \in G$ such that $g_1, g_2 \leq^{\mathsf{R}}_{\bot} g$. By (c), we can conclude $g_1(\pi) = g(\pi) = g_2(\pi)$.

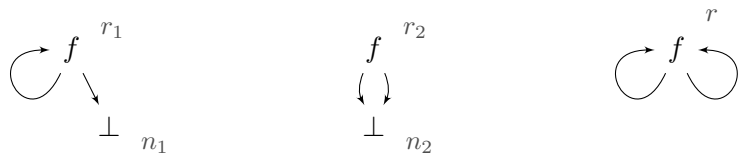
Next we show that $(P, l, \sim)$ is indeed a labelled quotient tree. Recall that $\sim$ needs to be an equivalence relation. For the reflexivity, assume that $\pi \in P$. Then there is some $g \in G$ with $\pi \in \mathcal{P}(g)$. Since $\sim_g$ is an equivalence relation, $\pi \sim_g \pi$ must hold and, therefore, $\pi \sim \pi$. For the symmetry, assume that $\pi_1 \sim \pi_2$. Then there is some $g \in G$ such that $\pi_1 \sim_g \pi_2$. Hence, we get $\pi_2 \sim_g \pi_1$ and, consequently, $\pi_2 \sim \pi_1$. In order to show transitivity, assume that $\pi_1 \sim \pi_2, \pi_2 \sim \pi_3$. That is, there are $g_1, g_2 \in G$ with $\pi_1 \sim_{g_1} \pi_2$ and $\pi_2 \sim_{g_2} \pi_3$. Since G is directed, we find some $g \in G$ such that $g_1, g_2 \leq^{\mathsf{R}}_{\bot} g$. By (a), this implies that also $\pi_1 \sim_g \pi_2$ and $\pi_2 \sim_g \pi_3$. Hence, $\pi_1 \sim_g \pi_3$ and, therefore, $\pi_1 \sim \pi_3$.

For the reachability condition, let $\pi \cdot \langle i \rangle \in P$. That is, there is a $g \in G$ with $\pi \cdot \langle i \rangle \in \mathcal{P}(g)$. Hence, $\pi \in \mathcal{P}(g)$, which in turn implies $\pi \in P$. Moreover, $\pi \cdot \langle i \rangle \in \mathcal{P}(g)$ implies that $i < \mathsf{ar}(g(\pi))$. Since $g(\pi)$ cannot be a nullary symbol and in particular not $\bot$, we obtain that $l(\pi) = g(\pi)$. Hence, $i < \mathsf{ar}(l(\pi))$.

For the congruence condition, assume that $\pi_1 \sim \pi_2$ and that $l(\pi_1) = f$. If $f \in \Sigma$, then there are $g_1, g_2 \in G$ with $\pi_1 \sim_{g_1} \pi_2$ and $g_2(\pi_1) = f$. Since G is directed, there is some $g \in G$ such that $g_1, g_2 \leq^{\mathsf{R}}_{\bot} g$. Hence, by (a) respectively (c), we have $\pi_1 \sim_g \pi_2$ and $g(\pi_1) = f$. Using Lemma 4.24 we can conclude that $g(\pi_2) = g(\pi_1) = f$ and that $\pi_1 \cdot i \sim_g \pi_2 \cdot i$ for all $i < \mathsf{ar}(g(\pi_1))$. Because $g \in G$, it holds that $l(\pi_2) = f$ and that $\pi_1 \cdot i \sim \pi \cdot i$ for all $i < \mathsf{ar}(l(\pi_1))$. If $f = \bot$, then also $l(\pi_2) = \bot$, for if $l(\pi_2) = f'$ for some $f' \in \Sigma$, then, by the symmetry of $\sim$ and the above argument (for the case $f \in \Sigma$), we would obtain $f = f'$ and, therefore, a contradiction. Since $\bot$ is a nullary symbol, the remainder of the condition is vacuously satisfied.

This shows that $(P, l, \sim)$ is a labelled quotient tree which, by Lemma 4.24, uniquely defines a canonical term graph. Next we show that the thus obtained term graph $\overline{g}$ is an upper bound for G. To this end, let $g \in G$. We will show that $g \leq^{\mathsf{R}}_{\bot} \overline{g}$ by establishing (a),(b) and (c). (a) and (c) are an immediate consequence of the construction. For (b), assume that $\pi_1 \in \mathcal{P}(g)$, $g(\pi_1) \in \Sigma$, $\pi_2 \in \mathcal{P}^a(\overline{g})$ and $\pi_1 \sim \pi_2$. We will show that then also $\pi_1 \sim_g \pi_2$ holds. Since $\pi_1 \sim \pi_2$, there is some $g' \in G$ with $\pi_1 \sim_{g'} \pi_2$. Because G is directed, there is some $g^* \in G$ with $g, g' \leq^{\mathsf{R}}_{\bot} g^*$. Using (a), we then get that $\pi_1 \sim_{g^*} \pi_2$. Note that since π_2 is acyclic in $\overline{g}$, it is also acyclic in g^*: Suppose that this is not the case, i.e. there are positions π_3, π_4 with $\pi_3 < \pi_4 \leq \pi_2$ and $\pi_3 \sim_{g^*} \pi_4$. But then we also have $\pi_3 \sim \pi_4$, which contradicts the assumption that π_2 is acyclic in $\overline{g}$. With this knowledge we are able to apply (b) to $\pi_1 \sim_{g^*} \pi_2$ in order to obtain $\pi_1 \sim_g \pi_2$.

In the final part of this proof, we will show that $\overline{g}$ is the least upper bound of G. For this purpose, let $\widehat{g}$ be an upper bound of G, i.e. $g \leq^{\mathsf{R}}_{\bot} \widehat{g}$ for all $g \in G$. We will show that $\overline{g} \leq^{\mathsf{R}}_{\bot} \widehat{g}$ by establishing (a), (b) and (c). For (a), assume that $\pi_1 \sim \pi_2$. Hence, there is some $g \in G$ with $\pi_1 \sim_g \pi_2$. Since, by assumption, $g \leq^{\mathsf{R}}_{\bot} \widehat{g}$, we can conclude $\pi_1 \sim_{\widehat{g}} \pi_2$ using (a). For (b), assume $\pi_1 \in P$, $l(\pi_1) \in \Sigma$,



(g) (h) $(g \sqcup h)$

Figure 6: Least upper bound $g \sqcup h$ of compatible term graphs g and h.

$\pi_2 \in \mathcal{P}^a(\widehat{g})$ and $\pi_1 \sim_{\widehat{g}} \pi_2$. That is, there is some $g \in G$ with $g(\pi_1) \in \Sigma$. Together with $g \leq^{\mathsf{R}}_{\bot} \widehat{g}$ this implies $\pi_1 \sim_g \pi_2$ by (b). $\pi_1 \sim \pi_2$ follows immediately. For (c), assume $\pi \in P$ and $l(\pi) = f \in \Sigma$. Then there is some $g \in G$ with $g(\pi) = f$. Applying (c) then yields $\widehat{g}(\pi) = f$ since $g \leq^{\mathsf{R}}_{\bot} \widehat{g}$. □

Remark 5.13. Following Remark 4.21, we define an order $\leq^{\mathsf{R}}_{\bot}$ on $\mathcal{G}^\infty(\Sigma_\bot)/_{\cong}$ which is isomorphic to the order $\leq^{\mathsf{R}}_{\bot}$ on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$. Define $[g]_{\cong} \leq^{\mathsf{R}}_{\bot} [h]_{\cong}$ iff there is a rigid $\bot$-homomorphism $\phi\colon\ g \to_\bot h$.

The extension of $\leq^{\mathsf{R}}_{\bot}$ to equivalence classes is easily seen to be well-defined: assume some rigid $\bot$-homomorphism $\phi\colon\ g \to_\bot h$ and two isomorphisms $g' \cong g$ and $h' \cong h$. Since, by Corollary 5.8, isomorphisms are also rigid ($\bot$-)homomorphisms, we have two rigid $\bot$-homomorphisms $\phi_1\colon\ g' \to_\bot g$ and $\phi_2\colon\ h \to_\bot h'$. Hence, by Proposition 5.2, $\phi_2 \circ \phi \circ \phi_1$ is a rigid $\bot$-homomorphism from g' to h'.

The isomorphism illustrated above allows us switch between the two partially ordered sets $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_{\bot})$ and $(\mathcal{G}^\infty(\Sigma_\bot)/_{\cong}, \leq^{\mathsf{R}}_{\bot})$ in order to use the structure that is more convenient for the given setting. In particular, the proof of Lemma 5.14 below is based on this isomorphism.

By Proposition 2.1, a cpo is a complete semilattice iff each two compatible elements have a least upper bound. Recall that compatible elements in a partially ordered set are elements that have a common upper bound. We make use of this proposition in order to show that $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_{\bot})$ is a complete semilattice. However, showing that each two term graphs $g, h \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$ with a common upper bound also have a least upper bound is not easy. The issue that makes the construction of the lub of compatible term graphs a bit more complicated than in the case of directed sets is illustrated in Figure 6. Note that the lub $g \sqcup h$ of the term graphs g and h has an additional cycle. The fact that in $g \sqcup h$ the second successor of r has to be r itself is enforced by g saying that the first successor of r_1 is r_1 itself and by h saying that the first and the second successor of r_2 must be identical. Because of the additional cycle in $g \sqcup h$, we have that the set of positions in $g \sqcup h$ is a proper superset of the union of the sets of positions in g and h. This makes the construction of $g \sqcup h$ using a labelled quotient tree quite intricate.

Our strategy to construct the lub is to form the disjoint union of the two term graphs in question and then identify nodes that have a common position w.r.t. the term graph they originate from. In our example, we have four nodes r_1, n_1, r_2 and n_2. At first r_1 and r_2 have to be identified as both have the position $\langle\rangle$. Next, r_1 and n_2 are identified as they share the position $\langle 0 \rangle$. And eventually,

also n_2 and n_1 are identified since they share the position $\langle 1 \rangle$. Hence, all four nodes have to be identified. The result is, therefore, a term graph with a single node r. The following lemma and its proof, given in Appendix A, show that, for any two compatible term graphs, this construction always yields their lub.

Lemma 5.14 (compatible elements have lub)**.** *Each pair g_1, g_2 of compatible term graphs in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\perp), \leq^R_\perp)$ has a least upper bound.*

Theorem 5.15. *The pair $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\perp), \leq^R_\perp)$ forms a complete semilattice.*

Proof. This is, according to Proposition 2.1, a consequence of Theorem 5.12 and Lemma 5.14. □

In particular, this means that the limit inferior is defined for every sequence of term graphs.

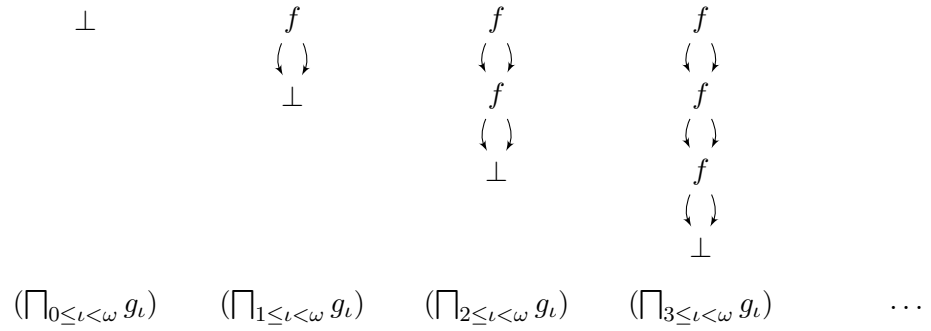
Corollary 5.16 (limit inferior of $\leq^\mathsf{R}_\perp$)**.** *Each sequence in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\perp), \leq^R_\perp)$ has a limit inferior.*

Recall that the intuition of the limit inferior on terms is that it contains the accumulated information that eventually remains stable in the sequence. This interpretation is, of course, based on the partial order $\leq_\perp$ on terms, which embodies the underlying notion of "information encoded in a term".

The same interpretation can be given for the limit inferior based on the rigid partial order $\leq^\mathsf{R}_\perp$ on term graphs. Given a sequence $(g_\iota)_{\iota<\alpha}$ of term graphs, its limit inferior $\liminf_{\iota\to\alpha} g_\iota$ is the term graph that contains the accumulation of all pieces of information that from some point onwards remain unchanged in $(g_\iota)_{\iota<\alpha}$.

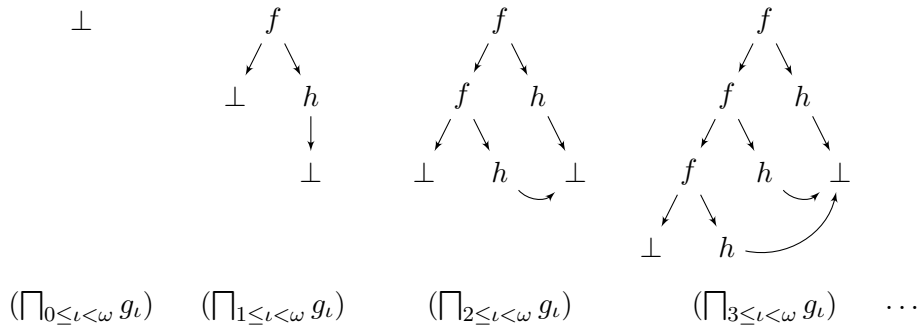
Example 5.17. 9d and 9e on page 347 each show a sequence of term graphs and its limit inferior in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\perp), \leq^\mathsf{R}_\perp)$.

(i) Figure 9d shows a simple example of how acyclic sharing is preserved by the limit inferior. The corresponding sequence $(\bigsqcap_{\beta\leq\iota<\omega} g_\iota)_{\beta<\omega}$ of greatest lower bounds is given as follows:



The least upper bound of this sequence of term graphs and thus the limit inferior of $(g_\iota)_{\iota<\omega}$ is the term graph g_ω depicted in Figure 9d.

(ii) The situation is slightly different in the sequence $(g_\iota)_{\iota<\omega}$ from Figure 9e. Here we also have acyclic sharing, viz. in the c-node. However, unlike in the previous example from Figure 9d, the acyclic sharing changes in each step. Hence, a lower bound of two distinct term graphs in $(g_\iota)_{\iota<\omega}$ cannot contain a c-node because a rigid $\bot$-homomorphism must map such a c-node to a c-node with the same acyclic sharing, i.e. the same acyclic positions. Consequently, the sequence of greatest lower bounds $(\bigsqcap_{\beta\leq\iota<\omega} g_\iota)_{\beta<\omega}$ looks as follows:



We thus get the term graph g_ω, depicted in Figure 9e, as the limit inferior of $(g_\iota)_{\iota<\omega}$. The $\bot$ labelling is necessary because of the change in acyclic sharing throughout the sequence.

While we have confirmed in Corollary 5.11 that the partial order $\leq_\bot^{\mathsf{R}}$ generalises the partial order $\leq_\bot$ on terms, we still have to show that this also carries over to the limit inferior. We can derive this property from the following simple lemma:

Lemma 5.18. *If $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$ and $t \in \mathcal{T}^\infty(\Sigma_\bot)$ with $g \leq_\bot^R t$, then $g \in \mathcal{T}^\infty(\Sigma_\bot)$.*

Proof. Since t is a term tree, $\sim_t$ is an identity relation. According to Corollary 5.10, $g \leq_\bot^{\mathsf{R}} t$ implies that $\sim_g \subseteq \sim_t$. Hence, also $\sim_g$ is an identity relation, which means that g is a term tree as well. □

Proposition 5.19. *Given a sequence $(t_\iota)_{\iota<\alpha}$ over $\mathcal{T}^\infty(\Sigma_\bot)$, the limit inferior of $(t_\iota)_{\iota<\alpha}$ in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ equals the limit inferior of $(t_\iota)_{\iota<\alpha}$ in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^R)$.*

Proof. Since both structures are complete semilattices, both limit inferiors exist. For each $\beta < \alpha$, let s_β be the glb of $T_\beta = \{t_\iota \mid \beta \leq \iota < \alpha\}$ in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ and g_β the glb of T_β in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$. Since then $g_\beta \leq_\bot^{\mathsf{R}} t_\beta$, we know by Lemma 5.18 that g_β is a term tree. By Corollary 5.11, this implies that g_β is the glb of T_β in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ as well, which means that $g_\beta = s_\beta$.

Let t and g be the limit inferior of $(t_\iota)_{\iota<\alpha}$ in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ and $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$, respectively. By the above argument, we know that t and g are the lub of the set $S = \{s_\beta \mid \beta < \alpha\}$ in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ respectively $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$. By Corollary 5.11, t is an upper bound of S in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$. Since g is the least such upper bound, we know that $g \leq_\bot^{\mathsf{R}} t$. According to Lemma 5.18, this implies that g is a term tree. Hence, by Corollary 5.11, g is an upper bound of S in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ and $g \leq_\bot t$. Since t is the least upper bound of S in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$, we can conclude that $t = g$. □

5.2.3 Maximal Term Graphs

Intuitively, partial term graphs represent partial results of computations where $\bot$-nodes act as placeholders denoting the uncertainty or ignorance of the actual "value" at that position. On the other hand, total term graphs do contain all the information of a result of a computation – they have the maximally possible information content. In other words, they are the maximal elements w.r.t. $\leq^{\mathsf{R}}_{\bot}$. The following proposition confirms this intuition.

Proposition 5.20 (total term graphs are maximal)**.** *Let Σ be a non-empty signature. Then $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ is the set of maximal elements in $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot}), \leq^{R}_{\bot})$.*

Proof. At first we need to show that each element in $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ is maximal. For this purpose, let $g \in \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ and $h \in \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot})$ such that $g \leq^{\mathsf{R}}_{\bot} h$. We have to show that then $g = h$. Since $g \leq^{\mathsf{R}}_{\bot} h$, there is a rigid $\bot$-homomorphism $\phi\colon g \to_{\bot} h$. As g does not contain any $\bot$-node, ϕ is even a rigid homomorphism. By Lemma 5.6, ϕ is injective and, therefore, according to Lemma 4.12, an isomorphism. Hence, we obtain that $g \cong h$ and, consequently, using Proposition 4.16, that $g = h$.

Secondly, we need to show that $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot})$ does not contain any other maximal elements besides those in $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$. Suppose there is a term graph $g \in \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot}) \setminus \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ which is maximal in $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot})$. Hence, there is a node $n^* \in N^g$ with $\mathsf{lab}^g(n^*) = \bot$. If Σ contains a nullary symbol c, construct a term graph h from g by relabelling the node n^* from $\bot$ to c. However, then $g <^{\mathsf{R}}_{\bot} h$, which contradicts the assumption that g is maximal w.r.t. $\leq^{\mathsf{R}}_{\bot}$. Otherwise, if $\Sigma^{(0)} = \emptyset$, let $\overline{n}$ be a fresh node (i.e. $\overline{n} \notin N^g$) and f some k-ary symbol in Σ. Define the term graph h by

$$N^h = N^g \uplus \{\overline{n}\} \qquad r^h = r^g$$

$$\mathsf{lab}^h(n) = \begin{cases} f & \text{if } n = n^* \\ \bot & \text{if } n = \overline{n} \\ \mathsf{lab}^g(n) & \text{otherwise} \end{cases} \qquad \mathsf{suc}^h(n) = \begin{cases} \langle \overline{n}, \ldots, \overline{n} \rangle & \text{if } n = n^* \\ \varepsilon & \text{if } n = \overline{n} \\ \mathsf{suc}^g(n) & \text{otherwise} \end{cases}$$

That is, h is obtained from g by relabelling n^* with f and setting the $\bot$-labelled node $\overline{n}$ as the target of all outgoing edges of n^*. We assume that $\overline{n}$ was chosen such that h is canonical (i.e. $\overline{n} = \mathcal{P}_h(\overline{n})$). Obviously, g and h are distinct. Define $\phi\colon N^g \to N^h$ by $n \mapsto n$ for all $n \in N^g$. Clearly, ϕ defines a rigid $\bot$-homomorphism from g to h. Hence, $g \leq^{\mathsf{R}}_{\bot} h$. This contradicts the assumption of g being maximal. Consequently, no element in $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot}) \setminus \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ is maximal. □

Note that this property does not hold for the simple partial order $\leq^{\mathsf{S}}_{\bot}$ that we have considered briefly in the beginning of this section. Figure 3 shows the total term graph g_0, which is strictly smaller than g_1 w.r.t. $\leq^{\mathsf{S}}_{\bot}$.

6 A Rigid Metric on Term Graphs

In this section, we pursue the metric approach to convergence in rewriting systems. To this end, we shall define a metric space on canonical term graphs. We base our approach to defining a metric distance on the definition of the metric

distance $\mathbf{d}$ on terms. In particular, we shall define a *truncation* operation on term graphs, which cuts off certain nodes depending on their depth in the term graph. Subsequently, we study the interplay of the truncation with Δ-homomorphisms and the depth of nodes within a term graph. Finally, we use the truncation operation to derive a metric on term graphs.

6.1 Truncating Term Graphs

Originally, Arnold and Nivat [4] used a truncation of terms to define the metric on terms. The truncation of a term t at depth $d \leq \omega$, denoted $t|d$, replaces all subterms at depth d by $\bot$:

$$t|0 = \bot, \quad f(t_1, \ldots, t_k)|d+1 = f(t_1|d, \ldots, t_k|d), \quad t|\omega = t$$

Recall that the metric distance $\mathbf{d}$ on terms is defined by $\mathbf{d}(s,t) = 2^{-\mathsf{sim}(s,t)}$. The underlying notion of similarity $\mathsf{sim}(\cdot,\cdot)$ can be characterised via truncations as follows:

$$\mathsf{sim}(s,t) = \max\{d \leq \omega \mid s|d = t|d\}$$

We adopt this approach for term graphs as well. To this end, we shall define a rigid truncation on term graphs. In Section 6.3 we will then show that this truncation indeed yields a complete metric space.

Definition 6.1 (rigid truncation of term graphs)**.** Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d < \omega$.

(i) Given $n, m \in N^g$, m is an *acyclic predecessor* of n in g if there is an acyclic position $\pi \cdot \langle i \rangle \in \mathcal{P}^a_g(n)$ with $\pi \in \mathcal{P}_g(m)$. The set of acyclic predecessors of n in g is denoted $\mathsf{Pre}^a_g(n)$.

(ii) The set of *retained nodes* of g at d, denoted $N^g_{<d}$, is the least subset M of N^g satisfying the following two conditions for all $n \in N^g$:

$$\text{(T1) } \mathsf{depth}_g(n) < d \implies n \in M \qquad \text{(T2) } n \in M \implies \mathsf{Pre}^a_g(n) \subseteq M$$

(iii) For each $n \in N^g$ and $i \in \mathbb{N}$, we use n^i to denote a fresh node, i.e. $\{n^i \mid n \in N^g, i \in \mathbb{N}\}$ is a set of pairwise distinct nodes not occurring in N^g. The set of *fringe nodes* of g at d, denoted $N^g_{=d}$, is defined as the singleton set $\{r^g\}$ if $d = 0$, and otherwise as the set

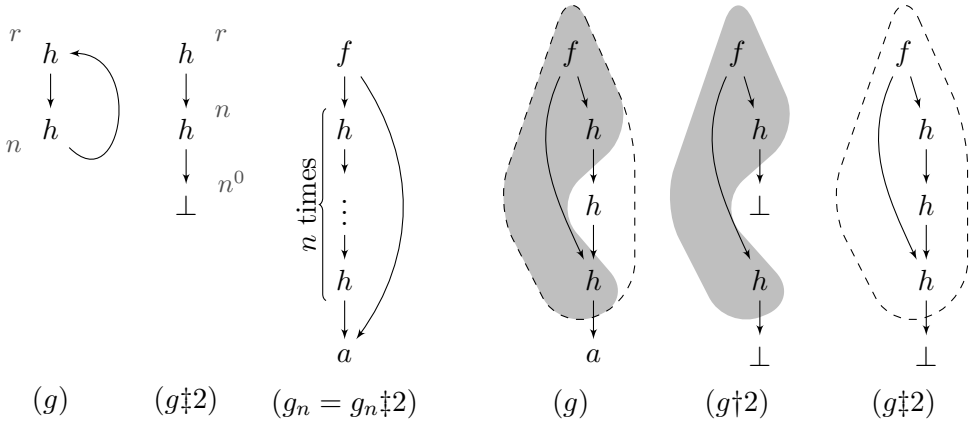
$$\left\{ n^i \;\middle|\; \begin{array}{rl} n \in N^g_{<d}, 0 \leq i < \mathsf{ar}_g(n) \text{ with} & \mathsf{suc}^g_i(n) \notin N^g_{<d} \\ \text{or} & \mathsf{depth}_g(n) \geq d-1, n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n)) \end{array} \right\}$$

(iv) The *rigid truncation* of g at d, denoted $g‡d$, is the term graph defined by

$$N^{g‡d} = N^g_{<d} \uplus N^g_{=d} \qquad r^{g‡d} = r^g$$

$$\mathsf{lab}^{g‡d}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases} \qquad \mathsf{suc}^{g‡d}_i(n) = \begin{cases} \mathsf{suc}^g_i(n) & \text{if } n^i \notin N^g_{=d} \\ n^i & \text{if } n^i \in N^g_{=d} \end{cases}$$

Additionally, we define $g‡\omega$ to be the term graph g itself.



(a) Cyclic vs. acyclic sharing.

(b) Comparison with simple truncation.

Figure 7: Examples of truncations.

Before discussing the intuition behind this definition of rigid truncation, let us have a look at the rôle of retained and fringe nodes: the set of retained nodes $N^g_{<d}$ contains the nodes that are *preserved* by the rigid truncation. All other nodes in $N^g \setminus N^g_{<d}$ are cut off. The "holes" that are thus created are filled by the fringe nodes in $N^g_{=d}$. This is expressed in the condition $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ which, if satisfied, yields a fringe node n^i. That is, a fresh fringe node is inserted for each successor of a retained node that is not a retained node itself. As fringe nodes function as a replacement for cut-off sub-term graphs, they are labelled with $\bot$ and have no successors.

But there is another circumstance that can give rise to a fringe node: if $\mathsf{depth}_g(n) \geq d-1$ and $n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$, we also get a fringe node n^i. This condition is satisfied whenever an outgoing edge from a retained node closes a cycle. The lower bound for the depth is chosen such that a successor node of n is not necessarily a retained node. An example is depicted in Figure 7a. For depth $d = 2$, the node n in the term graph g is just above the fringe, i.e. satisfies $\mathsf{depth}_g(n) \geq d-1$. Moreover, it has an edge to the node r that closes a cycle. Hence, the rigid truncation $g‡2$ contains the fringe node n^0 which is now the 0-th successor of n.

We chose this admittedly complicated notion of truncation in order to make it compatible with the partial order $\leq^{\mathsf{R}}_{\bot}$: first of all, the rigid truncation of a term graph is supposed to yield a smaller term graph w.r.t. the rigid partial order $\leq^{\mathsf{R}}_{\bot}$, i.e. $g‡d \leq^{\mathsf{R}}_{\bot} g$. Hence, whenever a node is kept as a retained node, also its acyclic positions have to be kept in order to preserve its upward structure. To achieve this, with each node also its *acyclic ancestors* have to be retained. The closure condition (T2) is enforced exactly for this purpose.

To see what this means, consider Figure 7b. It shows a term graph g and its truncation at depth 2, once without the closure condition (T2), denoted $g†2$, and once including (T2), denoted $g‡2$. The grey area highlights the nodes that are at depth smaller than 2, i.e. the nodes contained in $N^g_{<2}$ due to (T1). The nodes within the area surrounded by a dashed line are all the nodes in $N^g_{<2}$. One

can observe that with the *simple truncation* $g\dagger d$ without (T2), we do not have $g\dagger 2 \leq_\bot^{\mathsf{R}} g$. The reason in this particular example is the bottommost h-node whose acyclic sharing in g differs from that in the simple truncation $g\dagger 2$ as one of its predecessors was removed due to the truncation. This effect is avoided in our definition of rigid truncation, which always includes all acyclic predecessors of a node.

Nevertheless, the simple truncation $g\dagger d$ has its benefits. It is much easier to work with and provides a natural counterpart for the simple partial order $\leq_\bot^{\mathsf{S}}$ [9].

The following lemma confirms that we were indeed successful in making the truncation of term graphs compatible with the rigid partial order $\leq_\bot^{\mathsf{R}}$:

Lemma 6.2 (rigid truncation is smaller)**.** *Given $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \leq \omega$, we have that $g\ddagger d \leq_\bot^{\mathsf{R}} g$.*

Proof. The cases $d = \omega$ and $d = 0$ are trivial. Assume $0 < d < \omega$ and define the function ϕ as follows:

$$\begin{aligned} \phi\colon\ & N^{g\ddagger d} \to N^g \\ & N^g_{<d} \ni n \mapsto n \\ & N^g_{=d} \ni n^i \mapsto \mathsf{suc}^g_i(n) \end{aligned}$$

We will show that ϕ is a rigid $\bot$-homomorphism from $g\ddagger d$ to g and, thereby, $g\ddagger d \leq_\bot^{\mathsf{R}} g$.

Since $r^{g\ddagger d} = r^g$ and $r^{g\ddagger d} \in N^g_{<d}$, we have $\phi(r^{g\ddagger d}) = r^g$ and, therefore, the root condition. Note that all nodes in $N^g_{=d}$ are labelled with $\bot$ in $g\ddagger d$, i.e. all non-$\bot$-nodes are in $N^g_{<d}$. Thus, the labelling condition is trivially satisfied as for all $n \in N^g_{<d}$ we have

$$\mathsf{lab}^{g\ddagger d}(n) = \mathsf{lab}^g(n) = \mathsf{lab}^g(\phi(n)).$$

For the successor condition, let $n \in N^g_{<d}$. If $n^i \in N^g_{=d}$, then $\mathsf{suc}^{g\ddagger d}_i(n) = n^i$. Hence, we have

$$\phi(\mathsf{suc}^{g\ddagger d}_i(n)) = \phi(n^i) = \mathsf{suc}^g_i(n) = \mathsf{suc}^g_i(\phi(n)).$$

If, on the other hand, $n^i \notin N^g_{=d}$, then $\mathsf{suc}^{g\ddagger d}_i(n) = \mathsf{suc}^g_i(n) \in N^g_{<d}$. Hence, we have

$$\phi(\mathsf{suc}^{g\ddagger d}_i(n)) = \phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^g_i(n) = \mathsf{suc}^g_i(\phi(n)).$$

This shows that ϕ is a $\bot$-homomorphism. In order to prove that ϕ is rigid, we will show that $\mathcal{P}^a_g(\phi(n)) \subseteq \mathcal{P}_{g\ddagger d}(n)$ for all $n \in N^g_{<d}$, which is sufficient according to Lemma 5.7. Note that we can replace $\phi(n)$ by n since $n \in N^g_{<d}$. Therefore, we can show this statement by proving

$$\forall \pi \in \mathbb{N}^* \forall n \in N^g_{<d}.\ (\pi \in \mathcal{P}^a_g(n) \implies \pi \in \mathcal{P}_{g\ddagger d}(n))$$

by induction on the length of π. If $\pi = \langle\rangle$, then $n = r^g$ and, therefore, $\pi \in \mathcal{P}_{g\ddagger d}(n)$. If $\pi = \pi' \cdot \langle i \rangle$, let $m = \mathsf{node}_g(\pi')$. Then we have $m \in \mathsf{Pre}^a_g(n)$ and, therefore, $m \in N^g_{<d}$ by the closure property (T2). And since $\pi' \in \mathcal{P}^a_g(m)$, we can apply the induction hypothesis to obtain that $\pi' \in \mathcal{P}_{g\ddagger d}(m)$. Moreover, because $\mathsf{suc}^g_i(m) = n$, this implies that $m^i \notin N^g_{=d}$. Thus, $\mathsf{suc}^{g\ddagger d}_i(m) = n$ and, therefore, $\pi' \cdot \langle i \rangle \in \mathcal{P}_{g\ddagger d}(n)$. □

Also note that the rigid truncation on term graphs generalises Arnold and Nivat's [4] truncation on terms.

Proposition 6.3. *For each $t \in \mathcal{T}^\infty(\Sigma_\perp)$ and $d \leq \omega$, we have that $t\ddagger d \cong t|d$.*

Proof. For the case that $d \in \{0, \omega\}$, the equation $t\ddagger d = t|d$ holds trivially. For the other cases, we can easily see that $t|d$ is obtained from t by replacing all subterms at depth d by $\perp$. On the other hand, since in a term tree each node has at most one (acyclic) predecessor, which has a strictly smaller depth, we know that the set of retained nodes $N^t_{<d}$ is the set of nodes of depth smaller than d and the set of fringe nodes $N^t_{=d}$ is the set $\{n^i \mid n \in N^t, \mathsf{depth}_t(\mathsf{suc}^t_i(n)) = d\}$. Hence, $t\ddagger d$ is obtained from t by replacing each node at depth d with a fresh node labelled $\perp$. We can thus conclude that $t\ddagger d \cong t|d$. □

Consequently, if we use the rigid truncation to define a metric on term graphs analogously to Arnold and Nivat, we obtain a metric on term graphs that generalises the metric $\mathbf{d}$ on terms.

6.2 The Effect of Truncation

In order to characterise the effect of a truncation to a term graph, we need to associate an appropriate notion of depth to a whole term graph:

Definition 6.4 (symbol/graph depth)**.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $\Delta \subseteq \Sigma$.

(i) The *depth* of g, denoted $\mathsf{depth}(g)$, is the least upper bound of the depth of nodes in g, i.e.

$$\mathsf{depth}(g) = \bigsqcup \left\{ \mathsf{depth}_g(n) \,\middle|\, n \in N^g \right\}.$$

(ii) The Δ-*depth* of g, denoted $\Delta\text{-}\mathsf{depth}(g)$, is the minimum depth of nodes in g labelled in Δ, i.e.

$$\Delta\text{-}\mathsf{depth}(g) = \min \left\{ \mathsf{depth}_g(n) \,\middle|\, n \in N^g, \mathsf{lab}^g(n) \in \Delta \right\} \cup \{\omega\}.$$

If Δ is a singleton set $\{\sigma\}$, we also write $\sigma\text{-}\mathsf{depth}(g)$ instead of $\{\sigma\}\text{-}\mathsf{depth}(g)$.

Notice the difference between depth and Δ-depth. The former is the least upper bound of the depth of nodes in a term graph whereas the latter is the *minimum* depth of nodes labelled by a symbol in Δ. Thus, we have that $\mathsf{depth}(g) = \omega$ iff g is infinite; and $\Delta\text{-}\mathsf{depth}(g) = \omega$ iff g does not contain a Δ-node.

In the following, we will prove a number of lemmas that show how Δ-homomorphisms preserve the depth of nodes in term graphs. Understanding how Δ-homomorphisms affect the depth of nodes will become important for relating the rigid truncation to the rigid partial order $\leq^{\mathsf{R}}_\perp$.

Lemma 6.5 (reverse depth preservation of Δ-homomorphisms)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon g \to_\Delta h$. Then, for all $n \in N^h$ with $\mathsf{depth}_h(n) \leq \Delta\text{-}\mathsf{depth}(g)$, there is a node $m \in \phi^{-1}(n)$ with $\mathsf{depth}_g(m) \leq \mathsf{depth}_h(n)$.*

Proof. We prove the statement by induction on $\mathsf{depth}_h(n)$. If $\mathsf{depth}_h(n) = 0$, then $n = r^h$. With $m = r^g$, we have $\phi(m) = n$ and $\mathsf{depth}_g(m) = 0$. If $\mathsf{depth}_h(n) > 0$, then there is some $n' \in N^h$ with $\mathsf{suc}_i^h(n') = n$ and $\mathsf{depth}_h(n') < \mathsf{depth}_h(n)$. Hence, we can employ the induction hypothesis to obtain some $m' \in N^g$ with $\mathsf{depth}_g(m') \leq \mathsf{depth}_h(n')$ and $\phi(m') = n'$. Since $\mathsf{depth}_g(m') \leq \mathsf{depth}_h(n') < \mathsf{depth}_h(n) \leq \Delta\text{-}\mathsf{depth}(g)$, we have $\mathsf{lab}^g(m') \notin \Delta$. Hence, ϕ is homomorphic in m'. For $m = \mathsf{suc}_i^g(m')$, we can then reason as follows:

$$\phi(m) = \phi(\mathsf{suc}_i^g(m')) = \mathsf{suc}_i^h(\phi(m')) = \mathsf{suc}_i^h(n') = n, \quad \text{and}$$
$$\mathsf{depth}_g(m) \leq \mathsf{depth}_g(m') + 1 \leq \mathsf{depth}_h(n).$$

□

Lemma 6.6 (Δ-depth preservation of Δ-homomorphisms)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon g \to_\Delta h$, then $\Delta\text{-}\mathsf{depth}(g) \leq \Delta\text{-}\mathsf{depth}(h)$.*

Proof. Let $n \in N^h$ with $\mathsf{depth}_h(n) < \Delta\text{-}\mathsf{depth}(g)$. To prove the lemma, we have to show that $\mathsf{lab}^h(n) \notin \Delta$. According to Lemma 6.5, we find a node $m \in N^g$ with $\mathsf{depth}_g(m) \leq \mathsf{depth}_h(n) < \Delta\text{-}\mathsf{depth}(g)$ and $\phi(m) = n$. Since then $\mathsf{lab}^g(m) \notin \Delta$, we also have $\mathsf{lab}^h(n) \notin \Delta$ by the labelling condition for ϕ. □

For rigid Δ-homomorphisms, we even have a stronger form of depth preservation.

Lemma 6.7 (depth preservation of rigid Δ-homomorphisms)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon g \to_\Delta h$ a rigid Δ-homomorphism. Then $\mathsf{depth}_g(n) = \mathsf{depth}_h(\phi(n))$ for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$.*

Proof. If $\mathsf{lab}^g(n) \notin \Delta$, then $\mathcal{P}_g^a(n) = \mathcal{P}_h^a(\phi(n))$. Hence, $\mathsf{depth}_g(n) = \mathsf{depth}_h(\phi(n))$ follows since a shortest position of a node must be acyclic. □

The gaps that are caused by a truncation due to the removal of nodes are filled by fresh ⊥-nodes. The following lemma provides a lower bound for the depth of the introduced ⊥-nodes.

Lemma 6.8 (⊥-depth in rigid truncations)**.** *For all $g \in \mathcal{G}^\infty(\Sigma)$ and $d < \omega$, we have that*

(i) $\bot\text{-}\mathsf{depth}(g‡d) \geq d$, *and*

(ii) if $d > \mathsf{depth}(g) + 1$, *then* $g‡d = g$, *i.e.* $\bot\text{-}\mathsf{depth}(g‡d) = \omega$.

Proof. (i) From the proof of Lemma 6.2, we obtain a rigid ⊥-homomorphism $\phi\colon g‡d \to_\bot g$. Note that the only ⊥-nodes in $g‡d$ are those in $N^g_{=d}$. Each of these nodes has only a single predecessor, a node $n \in N^g_{<d}$ with $\mathsf{depth}_g(n) \geq d - 1$. By Lemma 6.7, we also have $\mathsf{depth}_{g‡d}(n) \geq d - 1$ for these nodes since ϕ is rigid, n is not labelled with ⊥ and $\phi(n) = n$. Hence, we have $\mathsf{depth}_{g‡d}(m) \geq d$ for each node $m \in N^g_{=d}$. Consequently, it holds that $\bot\text{-}\mathsf{depth}(g‡d) \geq d$.

(ii) Note that if $d > \mathsf{depth}(g) + 1$, then $N^g_{<d} = N^g$ and $N^g_{=d} = \emptyset$. Hence, $g‡d = g$. □

Remark 6.9. Note that the precondition for the statement of clause (ii) in the lemma above reads $d > \mathsf{depth}(g)+1$ rather than $d > \mathsf{depth}(g)$ as one might expect. The reason for this is that a truncation might cut off an edge that emanates from a node at depth $d-1$ and closes a cycle. For an example of this phenomenon, take a look at Figure 7a. It shows a term graph g of depth 1 and its rigid truncation at depth 2. Even though there is no node at depth 2 the truncation introduces a $\bot$-node.

On the other hand, although a term graph has depth greater than d, the truncation at depth d might still preserve the whole term graph. An example for this behaviour is the family of term graphs $(g_n)_{n<\omega}$ depicted in Figure 7a. Each of the term graphs g_n has depth $n+1$. Yet, the truncation at depth 2 preserves the whole term graph g_n for each $n > 0$. Even though there might be h-nodes which are at depth ≥ 2 these nodes are directly or indirectly acyclic predecessors of the a-node and are, thus, included in $N^{g_n}_{<2}$.

Intuitively, the following lemma states that a rigid $\bot$-homomorphism has the properties of an isomorphism up to the depth of the shallowest $\bot$-node:

Lemma 6.10 ($\leq^{\mathsf{R}}_{\bot}$ and rigid truncation)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d < \omega$ with $g \leq^{R}_{\bot} h$ and $\bot$-$\mathsf{depth}(g) \geq d$, we have that $g‡d \cong h‡d$.*

The proof of the above lemma is based on a generalisation of Lemma 6.7, which states that rigid $\bot$-homomorphisms map non-$\bot$-nodes to nodes of the same depth. However, since the rigid truncation of a term graph does not only depend on the depth of nodes but also the acyclic sharing in the term graph, we cannot rely on this statement on the depth of nodes alone. The two key components of the proof of Lemma 6.10 are 1. the property of rigid $\bot$-homomorphisms to map retained nodes of the source term graph exactly to the retained nodes of the target term graph and 2. that in the same way fringe nodes are exactly mapped to fringe nodes. Showing the isomorphism between $g‡d$ and $h‡d$ can thus be reduced to the injectivity on retained nodes in $g‡d$ which is obtained from the rigid $\bot$-homomorphism from g to h by applying Lemma 5.6. The full proof of Lemma 6.10 is given in Appendix B.

We can use the above findings in order to obtain the following properties of truncations that one would intuitively expect from a truncation operation:

Lemma 6.11 (smaller truncations)**.** *For all $g, h \in \mathcal{G}^\infty(\Sigma)$ and $e \leq d \leq \omega$, the following holds:*

$$\textit{(i) } g‡e \cong (g‡d)‡e, \textit{ and} \qquad \textit{(ii) } g‡d \cong h‡d \implies g‡e \cong h‡e.$$

Proof. (i) For $d = \omega$, this is trivial. Suppose $d < \omega$. From Lemma 6.2, we obtain $g‡d \leq^{\mathsf{R}}_{\bot} g$. Moreover, by Lemma 6.8, we have $\bot$-$\mathsf{depth}(g‡d) \geq d$ and, a fortiori, $\bot$-$\mathsf{depth}(g‡d) \geq e$. Hence, we can employ Lemma 6.10 to get $(g‡d)‡e \cong g‡e$.

(ii) Since $g‡d \cong h‡d$, we also have $(g‡d)‡e \cong (h‡d)‡e$, as the construction of the truncation only depends on the structure of the term graphs. Hence, using (i) we can conclude

$$g‡e \cong (g‡d)‡e \cong (h‡d)‡e \cong h‡e.$$

□

6.3 Deriving a Metric on Term Graphs

We may now define a rigid distance measure on canonical term graphs in the style of Arnold and Nivat:

Definition 6.12 (rigid distance)**.** The *rigid similarity* of two term graphs $g, h \in \mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma)$, written $\mathsf{sim}_{\ddagger}(g, h)$, is the maximum depth at which the rigid truncation of both term graphs coincide:

$$\mathsf{sim}_{\ddagger}(g, h) = \max\{d \leq \omega \mid g \ddagger d \cong h \ddagger d\}.$$

The *rigid distance* between two term graphs $g, h \in \mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma)$, written $\mathbf{d}_{\ddagger}(g, h)$ is defined as

$$\mathbf{d}_{\ddagger}(g, h) = 2^{-\mathsf{sim}_{\ddagger}(g,h)}, \text{ where we interpret } 2^{-\omega} \text{ as } 0.$$

Indeed, the resulting distance forms an ultrametric on the set of canonical term graphs:

Proposition 6.13 (rigid ultrametric)**.** *The pair* $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma), \mathbf{d}_{\ddagger})$ *forms an ultrametric space.*

Proof. The identity condition is derived as follows:

$$\mathbf{d}_{\ddagger}(g, h) = 0 \iff \mathsf{sim}_{\ddagger}(g, h) = \omega \iff g \cong h \overset{\text{Prop. 4.16}}{\iff} g = h$$

The symmetry condition is satisfied by the following equational reasoning:

$$\mathbf{d}_{\ddagger}(g, h) = 2^{-\mathsf{sim}_{\ddagger}(g,h)} = 2^{-\mathsf{sim}_{\ddagger}(h,g)} = \mathbf{d}_{\ddagger}(h, g)$$

For the strong triangle condition, we have to show that

$$\mathbf{d}_{\ddagger}(g_1, g_3) \leq \max\{\mathbf{d}_{\ddagger}(g_1, g_2), \mathbf{d}_{\ddagger}(g_2, g_3)\},$$

which is equivalent to

$$\mathsf{sim}_{\ddagger}(g_1, g_3) \geq \min\{\mathsf{sim}_{\ddagger}(g_1, g_2), \mathsf{sim}_{\ddagger}(g_2, g_3)\}.$$

Let $d = \mathsf{sim}_{\ddagger}(g_1, g_2)$ and $e = \mathsf{sim}_{\ddagger}(g_2, g_3)$. By symmetry, we can assume w.l.o.g. that $d \leq e$, i.e. $d = \min\{\mathsf{sim}_{\ddagger}(g_1, g_2), \mathsf{sim}_{\ddagger}(g_2, g_3)\}$. By definition of rigid similarity, we have both $g_1 \ddagger d \cong g_2 \ddagger d$ and $g_2 \ddagger e \cong g_3 \ddagger e$. From the latter we obtain, by Lemma 6.11, that $g_2 \ddagger d \cong g_3 \ddagger d$. That is, $g_1 \ddagger d \cong g_2 \ddagger d \cong g_3 \ddagger d$ which means that $\mathsf{sim}_{\ddagger}(g_1, g_3) \geq d$. □

Example 6.14. Figures 8c and 9d on pages 345 and 347, respectively, show two sequences of term graphs that are converging in the metric space $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma), \mathbf{d}_{\ddagger})$. In the sequence $(h_i)_{i<\omega}$ from Figure 8c, we have that the rigid truncation at 0 is trivially $\bot$ for all term graphs in the sequence. From h_1 onwards, the rigid truncation at 1 is the term tree $\bot :: \bot$; from h_2 onwards, the rigid truncation at 2 is the term tree $b :: \bot :: \bot$; etc. Hence, for each $n < \omega$, the metric distance $\mathbf{d}_{\ddagger}(h_i, h_j)$ between two term graphs from h_n onwards, i.e. with $n \leq i, j < \omega$, is at most 2^{-n}. That is, the sequence $(h_i)_{i<\omega}$ is Cauchy. Even more, for the term tree $h_\omega = b :: b :: b :: \ldots$ depicted in Figure 8c we also have that $h_\omega \ddagger 0 = \bot$,

$h_\omega \ddagger 1 = \bot :: \bot$, $h_\omega \ddagger 2 = b :: \bot :: \bot$, etc. Hence, for each $n < \omega$, the metric distance $\mathbf{d}_\ddagger(h_n, h_\omega)$ is at most 2^{-n}. That is, the sequence $(h_i)_{i<\omega}$ converges to h_ω. In a similar fashion, the sequence depicted in Figure 9d converges as well.

Figure 9e shows a sequence $(g_i)_{i<\omega}$ of term graphs that does not converge. In fact, it is not even Cauchy. To see this, notice that the c-node is at depth 1 in g_0 and at depth 2 from g_1 onwards. As in each term graph g_i the c-node is reachable from any node in g_i without forming a cycle, we have that each node is an acyclic ancestor of the c-node. That is, whenever the c-node is retained by a rigid truncation, so is any other node. Consequently, we have that $g_i \ddagger d = g_i$ for each $i < \omega$ and $d > 2$. Hence, the metric distance $\mathbf{d}_\ddagger(g_i, g_j)$ between each pair of term graphs with $i \neq j$ is at least 2^{-2}. That is, $(g_i)_{i<\omega}$ is not Cauchy.

Since we defined the metric on term graphs in the same style as Arnold and Nivat [4] defined the partial order $\mathbf{d}$ on terms, we can use the correspondence between the rigid truncation and the truncation on terms in order to derive that the metric $\mathbf{d}_\ddagger$ generalises the metric $\mathbf{d}$ on terms.

Corollary 6.15. *For all $s, t \in \mathcal{T}^\infty(\Sigma)$, we have that $\mathbf{d}_\ddagger(s,t) = \mathbf{d}(s,t)$.*

Proof. Follows from Proposition 6.3. □

From the above observation, we obtain that convergence in the metric space of term graphs $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$ is a conservative extension of convergence in the metric space of terms $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$:

Proposition 6.16. *Every non-empty sequence over $\mathcal{T}^\infty(\Sigma)$ converges to t in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$ iff it converges to t in $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$.*

Proof. The "if" direction follows immediately from Corollary 6.15.

For the "iff" direction we assume a sequence S over $\mathcal{T}^\infty(\Sigma)$ that converges to t in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$. Consequently, S is also Cauchy in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$. Due to Corollary 6.15, S is then also Cauchy in $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$. Since $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$ is complete, S converges to some term t' in $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$. Using the "if" direction of this proposition, we then obtain that S converges to t' in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$. Since limits are unique in metric spaces, we can conclude that $t = t'$. □

7 Metric vs. Partial Order Convergence

In this section we study both the partially ordered set $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$ and the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$. In particular, we are interested in the notion of convergence that each of the two structures provides. We shall show that on total term graphs – i.e. in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ – both structures yield the same notion of convergence. That is, we obtain the same correspondence that we already know from infinitary term rewriting as stated in Theorem 3.3. Moreover, as a side product, this finding will also show the completeness of the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$.

The cornerstone of this comparison of the rigid metric $\mathbf{d}_\ddagger$ and the rigid partial order $\leq^\mathsf{R}_\bot$ is the following characterisation of the rigid similarity $\mathsf{sim}_\ddagger(\cdot,\cdot)$ in terms of greatest lower bounds in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$:

Proposition 7.1 (characterisation of rigid similarity)**.** *Let* $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ *and* $g \sqcap h$ *the greatest lower bound of* g *and* h *in* $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^R_\bot)$*. Then* $\mathsf{sim}_\ddagger(g,h) = \bot\text{-}\mathsf{depth}(g \sqcap h)$*.*

Proof. At first assume that $g = h$. Hence, $g \sqcap h = g$ and, consequently $\bot\text{-}\mathsf{depth}(g \sqcap h) = \omega$ as g does not contain any node labelled $\bot$. On the other hand, $g = h$ implies $g\ddagger\omega \cong h\ddagger\omega$, and, therefore, $\mathsf{sim}_\ddagger(g,h) = \omega$. If $g \neq h$, then $g \ncong h$ by Proposition 4.16. Hence, $\mathsf{sim}_\ddagger(g,h) < \omega$. Moreover, since $g \ncong h$, we know that $g \sqcap h$ has to be strictly smaller than g or h w.r.t. $\leq^\mathsf{R}_\bot$. Hence, according to Proposition 5.20, $g \sqcap h$ has to contain some node labelled $\bot$, i.e. $\bot\text{-}\mathsf{depth}(g \sqcap h) < \omega$ as well. We prove that $\bot\text{-}\mathsf{depth}(g \sqcap h) = \mathsf{sim}_\ddagger(g,h)$ by showing that both $\bot\text{-}\mathsf{depth}(g \sqcap h) \leq \mathsf{sim}_\ddagger(g,h)$ and $\bot\text{-}\mathsf{depth}(g \sqcap h) \geq \mathsf{sim}_\ddagger(g,h)$ hold.

In order to show the former, let $d = \bot\text{-}\mathsf{depth}(g \sqcap h)$. Since $g \sqcap h \leq^\mathsf{R}_\bot g, h$, we can apply Lemma 6.10 twice in order to obtain $g\ddagger d \cong (g \sqcap h)\ddagger d \cong h\ddagger d$. Hence, $\mathsf{sim}_\ddagger(g,h) \geq d$.

To show the converse direction, let $d = \mathsf{sim}_\ddagger(g,h)$, i.e. $g\ddagger d \cong h\ddagger d$. According to Lemma 6.2, we have both $g\ddagger d \leq^\mathsf{R}_\bot g$ and $h\ddagger d \leq^\mathsf{R}_\bot h$. Note that, for the canonical representation, we then have $\mathcal{C}(g\ddagger d) = \mathcal{C}(h\ddagger d)$, $\mathcal{C}(g\ddagger d) \leq^\mathsf{R}_\bot g$ and $\mathcal{C}(h\ddagger d) \leq^\mathsf{R}_\bot h$ (cf. Proposition 4.16 respectively Remark 5.13). That is, $\mathcal{C}(g\ddagger d)$ is a lower bound of g and h. Thus, $\mathcal{C}(g\ddagger d) \leq^\mathsf{R}_\bot g \sqcap h$ and we can reason as follows:

$$\begin{aligned} d &\leq \bot\text{-}\mathsf{depth}(g\ddagger d) && \text{(Lem. 6.8)} \\ &= \bot\text{-}\mathsf{depth}(\mathcal{C}(g\ddagger d)) && \text{(Lem. 6.7, Cor. 5.8)} \\ &\leq \bot\text{-}\mathsf{depth}(g \sqcap h) && (\mathcal{C}(g\ddagger d) \leq^\mathsf{R}_\bot g \sqcap h\text{, Lem. 6.6)} \end{aligned}$$

□

Remark 7.2. From now on, we are not dealing with the concrete construction of rigid truncations $g\ddagger d$ anymore. Therefore, we will rather use the canonical representation $\mathcal{C}(g\ddagger d)$ of $g\ddagger d$. In order to avoid the notational overhead, we write $g\ddagger d$ instead of $\mathcal{C}(g\ddagger d)$.

In the next step we show that the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$ is indeed complete. The following proposition states even more: the limit of Cauchy sequences in the metric space equals its limit inferior in the partially ordered set $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$:

Proposition 7.3 (metric limit = limit inferior)**.** *Let* $(g_\iota)_{\iota<\alpha}$ *be a non-empty Cauchy sequence in the metric space* $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\ddagger)$ *and* $\liminf_{\iota\to\alpha} g_\iota$ *its limit inferior in* $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^R_\bot)$*. Then* $\lim_{\iota\to\alpha} g_\iota = \liminf_{\iota\to\alpha} g_\iota$*.*

Proof. If α is a successor ordinal, this is trivial, as the limit and the limit inferior are then $g_{\alpha-1}$. Assume that α is a limit ordinal and let $\overline{g}$ be the limit inferior of $(g_\iota)_{\iota<\alpha}$. Since, according to Theorem 5.15, $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$ is a complete semilattice, $\overline{g}$ is well-defined. Since $(g_\iota)_{\iota<\alpha}$ is Cauchy, we obtain that, for each $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that we have $\mathbf{d}_\ddagger(g_\iota, g_{\iota'}) < \varepsilon$ for all $\beta \leq \iota, \iota' < \alpha$. A fortiori, we get that, for each $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that we have $\mathbf{d}_\ddagger(g_\beta, g_\iota) < \varepsilon$ for all $\beta \leq \iota < \alpha$. By definition of $\mathbf{d}_\ddagger$, this is equivalent to $2^{-\mathsf{sim}_\ddagger(g_\beta, g_\iota)} < \varepsilon$. Consequently, we have, for each $d < \omega$, a $\beta < \alpha$ such that $\mathsf{sim}_\ddagger(g_\beta, g_\iota) > d$ for all $\beta \leq \iota < \alpha$. Due to Lemma 6.11, $\mathsf{sim}_\ddagger(g_\beta, g_\iota) > d$ implies $g_\beta\ddagger d = g_\iota\ddagger d$ which in

turn implies $g_\beta \ddagger d \leq^{\mathsf{R}}_{\perp} g_\iota$, according to Lemma 6.2. Hence, $g_\beta \ddagger d$ is a lower bound for $G_\beta = \{g_\iota \,|\, \beta \leq \iota < \alpha\}$, i.e. $g_\beta \ddagger d \leq^{\mathsf{R}}_{\perp} \bigsqcap G_\beta$. Moreover, by the definition of the limit inferior, it holds that $\bigsqcap G_\beta \leq^{\mathsf{R}}_{\perp} \overline{g}$. Consequently, $g_\beta \ddagger d \leq^{\mathsf{R}}_{\perp} \overline{g}$, i.e. we have

$$\forall d < \omega \exists \beta < \alpha \colon \quad g_\beta \ddagger d \leq^{\mathsf{R}}_{\perp} \overline{g} \tag{1}$$

Since, by Lemma 6.8, we have $\perp\text{-}\mathsf{depth}(g_\beta \ddagger d) \geq d$, we can apply Lemma 6.10 to obtain $(g_\beta \ddagger d) \ddagger d \cong \overline{g} \ddagger d$. Hence, by Lemma 6.11, we have $g_\beta \ddagger d \cong \overline{g} \ddagger d$ and therefore $\mathsf{sim}_\ddagger(\overline{g}, g_\beta) \geq d$. That is, we have shown that

$$\forall d < \omega \exists \beta < \alpha \colon \quad \mathsf{sim}_\ddagger(\overline{g}, g_\beta) \geq d$$

Since, for each $\varepsilon \in \mathbb{R}^+$, we find a $d < \omega$ with $2^{-d} < \varepsilon$, this implies

$$\forall \varepsilon \in \mathbb{R}^+ \exists \beta < \alpha \colon \quad \mathbf{d}_\ddagger(\overline{g}, g_\beta) < \varepsilon$$

This shows that $(g_\iota)_{i<\alpha}$ converges to $\overline{g}$. Now it remains to be shown that $\overline{g}$ is indeed in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, i.e. it does not contain any node labelled $\perp$. Suppose that $\overline{g}$ does contain a node labelled with $\perp$. Then $\perp\text{-}\mathsf{depth}(\overline{g}) < \omega$. Let $d = \perp\text{-}\mathsf{depth}(\overline{g}) + 1$. By (1), there is a β with $g_\beta \ddagger d \leq^{\mathsf{R}}_{\perp} \overline{g}$. By applying Lemma 6.8 and Lemma 6.6, we then get

$$\perp\text{-}\mathsf{depth}(\overline{g}) + 1 = d \leq \perp\text{-}\mathsf{depth}(g_\beta \ddagger d) \leq \perp\text{-}\mathsf{depth}(\overline{g}).$$

This is a contradiction. Hence, $\overline{g}$ is indeed in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$. □

This result has two obvious but important consequences: firstly, the limit of a converging sequence in the rigid metric space is equal to the limit inferior in the rigid complete semilattice. Secondly, the rigid metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ is complete:

Theorem 7.4 (completeness of rigid metric)**.** *The metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ is complete.*

Proof. Immediate consequence of Proposition 7.3. □

In the following proposition, we show the converse direction of the relation between the limits of the rigid metric and the limit inferiors of the rigid partial order:

Proposition 7.5 (total limit inferior = limit)**.** *Let $(g_\iota)_{\iota<\alpha}$ be a non-empty sequence in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ and $\liminf_{\iota\to\alpha} g_\iota$ its limit inferior in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\perp), \leq^{\mathsf{R}}_{\perp})$. If $\liminf_{\iota\to\alpha} g_\iota \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, then $\liminf_{\iota\to\alpha} g_\iota = \lim_{\iota\to\alpha} g_\iota$.*

Proof. If α is a successor ordinal, then both the limit and the limit inferior are equal to $g_{\alpha-1}$. Let α be a limit ordinal. According to Proposition 7.3, in order to show that $(g_\iota)_{\iota<\alpha}$ converges and that its limit coincides with its limit inferior, it suffices to prove that $(g_\iota)_{\iota<\alpha}$ is Cauchy.

Let $\overline{g} = \liminf_{\iota\to\alpha} g_\iota$, and define $G_\beta = \{g_\iota \,|\, \beta \leq \iota < \alpha\}$ and $h_\beta = \bigsqcap G_\beta$ for each $\beta < \alpha$. Note that $\overline{g} = \bigsqcup_{\beta<\alpha} h_\beta$. Since $\overline{g}$ is total, i.e. no node in $\overline{g}$ is labelled with $\perp$, we have, according to Theorem 5.12, that for each $\pi \in \mathcal{P}(\overline{g})$ there is some $\beta_\pi < \alpha$ with $h_{\beta_\pi}(\pi) = \overline{g}(\pi)$.

Note that $(h_\iota)_{\iota<\alpha}$ is monotonic w.r.t. $\leq^{\mathsf{R}}_{\perp}$, i.e. $\iota \leq \iota'$ implies $h_\iota \leq^{\mathsf{R}}_{\perp} h_{\iota'}$. Since $h_\iota \leq^{\mathsf{R}}_{\perp} h_{\iota'}$ together with $h_\iota(\pi) \in \Sigma$ implies $h_{\iota'}(\pi) = h_\iota(\pi)$ by Corollary 5.10, we have $h_\gamma(\pi) = \overline{g}(\pi)$ for all $\pi \in \mathcal{P}(\overline{g})$ and $\beta_\pi \leq \gamma < \alpha$.

Let $d < \omega$. Since there are only finitely many positions in $\mathcal{P}(\overline{g})$ of length smaller than d, we know that $\beta_d = \max\{\beta_\pi \,|\, \pi \in \mathcal{P}(\overline{g}), |\pi| < d\}$ is a well-defined ordinal smaller than α. Hence, for all $\pi \in \mathcal{P}(\overline{g})$ with $|\pi| < d$ we have $h_{\beta_d}(\pi) = \overline{g}(\pi)$. Since $\overline{g}$ is total, we thus have that $\perp\text{-}\mathsf{depth}(h_{\beta_d}) \geq d$.

Since $g_\iota, g_{\iota'} \in G_{\beta_d}$ for each $\beta_d \leq \iota, \iota' < \alpha$, we have $h_{\beta_d} \leq^{\mathsf{R}}_{\perp} g_\iota, g_{\iota'}$ and thus $h_{\beta_d} \leq^{\mathsf{R}}_{\perp} g_\iota \sqcap g_{\iota'}$. Consequently, by Lemma 6.6, we have that $\perp\text{-}\mathsf{depth}(g_\iota \sqcap g_{\iota'}) \geq \perp\text{-}\mathsf{depth}(h_{\beta_d})$. That is, for all ι, ι' with $\beta_d < \iota, \iota' < \alpha$, we have the following:

$$\mathsf{sim}_{\ddagger}(g_\iota, g_{\iota'}) \overset{Prop.\ 7.1}{=} \perp\text{-}\mathsf{depth}(g_\iota \sqcap g_{\iota'}) \geq \perp\text{-}\mathsf{depth}(h_{\beta_d}) \geq d$$

Now, let $\varepsilon \in \mathbb{R}^+$. We then find some $d < \omega$ with $\varepsilon > 2^{-d}$. Consequently, we have

$$\mathbf{d}_{\ddagger}(g_\iota, g_{\iota'}) = 2^{-\mathsf{sim}_{\ddagger}(g_\iota, g_{\iota'})} \leq 2^{-d} < \varepsilon \qquad \text{for all } \beta_d \leq \iota, \iota' < \alpha.$$

Hence, $(g_\iota)_{\iota<\alpha}$ is Cauchy. □

Note that Proposition 7.5 depends on the finiteness of the arity of the symbols in the signature. (This is used in the proof above when observing that a term graph has only finitely many positions of a bounded length.) This restriction also applies to terms as the following example shows:

Example 7.6. Let $\Sigma = \{f/\omega, a/0, b/0\}$ and $(t_i)_{i<\omega}$ a sequence with

$$\begin{aligned} t_0 &= f(a, a, a, a, a \dots), \\ t_1 &= f(b, a, a, a, a \dots), \\ t_2 &= f(b, b, a, a, a \dots), \\ t_3 &= f(b, b, b, a, a \dots), \\ &\vdots \end{aligned}$$

$(t_i)_{i<\omega}$ has the limit inferior $f(b, b, b, b, b, \dots)$. On the other hand, the sequence is not even Cauchy since, for each $i \neq j$, we have $\mathsf{sim}_{\ddagger}(t_i, t_j) = 1$ and, therefore, $\mathbf{d}_{\ddagger}(t_i, t_j) = \frac{1}{2}$.

8 Infinitary Term Graph Rewriting

In the previous sections, we have constructed and investigated the necessary metric and partial order structures upon which the infinitary calculus of term graph rewriting that we shall introduce in this section is based. After describing the framework of term graph rewriting that we consider, we will explore two different modes of convergence on term graphs. In the same way that infinitary term rewriting is based on the abstract notions of m- and p-convergence [6], infinitary term graph rewriting is an instantiation of these abstract modes of convergence for term graphs. However, as in the overview of infinitary term rewriting in Section 2, we restrict ourselves to weak notions of convergence.

8.1 Term Graph Rewriting Systems

In this paper, we adopt the term graph rewriting framework of Barendregt et al. [10]. In order to represent placeholders in rewrite rules, this framework uses variables – in a manner much similar to term rewrite rules. To this end, we consider a signature $\Sigma_{\mathcal{V}} = \Sigma \uplus \mathcal{V}$ that extends the signature Σ with a set $\mathcal{V}$ of nullary variable symbols.

Definition 8.1 (term graph rewriting systems)**.**

(i) Given a signature Σ, a *term graph rule* ρ over Σ is a triple (g, l, r) where g is a graph over $\Sigma_{\mathcal{V}}$ and $l, r \in N^g$ such that all nodes in g are reachable from l or r. We write ρ_l respectively ρ_r to denote the left- respectively right-hand side of ρ, i.e. the term graph $g|_l$ respectively $g|_r$. Additionally, we require that, for each variable $v \in \mathcal{V}$, there is at most one node n in g labelled v and that n is different but still reachable from l.

(ii) A *term graph rewriting system (GRS)* $\mathcal{R}$ is a pair (Σ, R) with Σ a signature and R a set of term graph rules over Σ.

The requirement that the root l of the left-hand side is not labelled with a variable symbol is analogous to the requirement that the left-hand side of a term rule is not a variable. Similarly, the restriction that nodes labelled with variable symbols must be reachable from the root of the left-hand side corresponds to the restriction on term rules that every variable occurring on the right-hand side of a rule must also occur on the left-hand side.

Term graphs can be used to compactly represent terms, which is formalised by the unravelling operator $\mathcal{U}(\cdot)$. We extend this operator to term graph rules. Figure 8a illustrates two term graph rules that both represent the term rule $a :: x \to b :: a :: x$ from Example 3.1 to which they unravel.

Definition 8.2 (unravelling of term graph rules)**.** Let ρ be a term graph rule with ρ_l and ρ_r its left- respectively right-hand side term graph. The *unravelling* of ρ, denoted $\mathcal{U}(\rho)$ is the term rule $\mathcal{U}(\rho_l) \to \mathcal{U}(\rho_r)$.

The application of a rewrite rule ρ (with root nodes l and r) to a term graph g is performed in three steps: at first a suitable sub-term graph of g rooted in some node n of g is *matched* against the left-hand side of ρ. This amounts to finding a $\mathcal{V}$-homomorphism $\phi\colon\ \rho_l \to_{\mathcal{V}} g|_n$ from the term graph rooted in l to the sub-term graph rooted in n, the *redex*. The $\mathcal{V}$-homomorphism ϕ allows us to instantiate variables in the rule with sub-term graphs of the redex. In the second step, nodes and edges in ρ that are not reachable from l are copied into g, such that each edge pointing to a node m in the term graph rooted in l is redirected to $\phi(m)$. In the last step, all edges pointing to n are redirected to (the copy of) r and all nodes not reachable from the root of (the now modified version of) g are removed.

The formal definition of this construction is given below:

Definition 8.3 (application of term graph rewrite rules, [10])**.** Let $\rho = (N^\rho, \mathsf{lab}^\rho, \mathsf{suc}^\rho, l^\rho, r^\rho)$ be a term graph rewrite rule in a GRS $\mathcal{R} = (\Sigma, R)$, $g \in \mathcal{G}^\infty(\Sigma)$ with

$N^\rho \cap N^g = \emptyset$ and $n \in N^g$. ρ is called *applicable* to g at n if there is a $\mathcal{V}$-homomorphism $\phi\colon \rho_l \to_{\mathcal{V}} g|_n$. ϕ is called the *matching $\mathcal{V}$-homomorphism* of the rule application, and $g|_n$ is called a *ρ-redex*. Next, we define the *result* of the application of the rule ρ to g at n using the $\mathcal{V}$-homomorphism ϕ. This is done by constructing the intermediate graphs g_1 and g_2, and the final result g_3.

(i) The graph g_1 is obtained from g by adding the part of ρ that is not contained in its left-hand side:

$$N^{g_1} = N^g \uplus (N^\rho \setminus N^{\rho_l})$$

$$\mathsf{lab}^{g_1}(m) = \begin{cases} \mathsf{lab}^g(m) & \text{if } m \in N^g \\ \mathsf{lab}^\rho(m) & \text{if } m \in N^\rho \setminus N^{\rho_l} \end{cases}$$

$$\mathsf{suc}_i^{g_1}(m) = \begin{cases} \mathsf{suc}_i^g(m) & \text{if } m \in N^g \\ \mathsf{suc}_i^\rho(m) & \text{if } m, \mathsf{suc}_i^\rho(m) \in N^\rho \setminus N^{\rho_l} \\ \phi(\mathsf{suc}_i^\rho(m)) & \text{if } m \in N^\rho \setminus N^{\rho_l}, \mathsf{suc}_i^\rho(m) \in N^{\rho_l} \end{cases}$$

(ii) Let $n' = \phi(r^\rho)$ if $r^\rho \in N^{\rho_l}$ and $n' = r^\rho$ otherwise. The graph g_2 is obtained from g_1 by redirecting edges ending in n to n':

$$N^{g_2} = N^{g_1} \qquad \mathsf{lab}^{g_2} = \mathsf{lab}^{g_1} \qquad \mathsf{suc}_i^{g_2}(m) = \begin{cases} \mathsf{suc}_i^{g_1}(m) & \text{if } \mathsf{suc}_i^{g_1}(m) \neq n \\ n' & \text{if } \mathsf{suc}_i^{g_1}(m) = n \end{cases}$$

(iii) The term graph g_3 is obtained by setting the root node r', which is n' if $n = r^g$, and otherwise r^g. That is, $g_3 = g_2|_{r'}$. This also means that all nodes not reachable from r' are removed.

This induces a *pre-reduction step* $\psi = (g, n, \rho, n', g_3)$ from g to g_3, written $\psi\colon g \mapsto_{n,\rho,n'} g_3$. In order to indicate the underlying GRS $\mathcal{R}$, we also write $\psi\colon g \mapsto_{\mathcal{R}} g_3$.

Examples for term graph (pre-)reduction steps are shown in Figure 8. We revisit them in more detail in Example 8.9 below.

The definition of term graph rewriting in the form of pre-reduction steps is very operational in style. The result of applying a rewrite rule to a term graph is constructed in several steps by manipulating nodes and edges explicitly. While this is beneficial for implementing a rewriting system, it is problematic for reasoning on term graphs up to isomorphisms, which is necessary for introducing notions of convergence. In our case, however, this does not cause any harm since the construction in Definition 8.3 is invariant under isomorphism:

Proposition 8.4 (pre-reduction steps)**.** *Let $\phi\colon g \mapsto_{n,\rho,m} h$ be a pre-reduction step in some GRS $\mathcal{R}$ and $\psi_1\colon g' \cong g$. Then there is a pre-reduction step of the form $\phi'\colon g' \mapsto_{n',\rho,m'} h'$ with $\psi_2\colon h' \cong h$ such that $\psi_2(n') = n$ and $\psi_1(m') = m$.*

Proof. Immediate from the construction in Definition 8.3. □

The above finding justifies the following definition of reduction steps:

Definition 8.5 (reduction steps)**.** Let $\mathcal{R} = (\Sigma, R)$ be a GRS, $\rho \in R$ and $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ with $n \in N^g$ and $m \in N^h$. A tuple $\phi = (g, n, \rho, m, h)$ is called a *reduction step*, written $\phi\colon\ g \to_{n,\rho,m} h$, if there is a pre-reduction step $\phi'\colon\ g' \mapsto_{n',\rho,m'} h'$ with $\mathcal{C}(g') = g$, $\mathcal{C}(h') = h$, $n = \mathcal{P}_{g'}(n')$, and $m = \mathcal{P}_{h'}(m')$. Similarly to pre-reduction steps, we also write $\phi\colon\ g \to_\mathcal{R} h$ or simply $\phi\colon\ g \to h$ for short.

In other words, a reduction step is a canonicalised pre-reduction step.

Note that term graph rules do not provide a duplication mechanism. Each variable is allowed to occur at most once. Duplication must always be simulated by sharing. This means for example that variables that should occur on the right-hand side must share the occurrence of that variable on the left-hand side of the rule with its right-hand side. This can be seen in the term graph rules in Figure 8a. The sharing can be direct as in ρ_1 or indirect as in ρ_2. For variables that are supposed to be duplicated on the right-hand side, for example in the term rewrite rule $h(x) \to f(h(x), h(x))$, we have to use sharing in order to represent multiple occurrences of the same variable. This representation can be seen in the corresponding term graph rules in Figure 9a.

8.2 Convergence of Transfinite Reductions

We now employ the partial order $\leq^{\mathsf{R}}_\perp$ and the metric $\mathbf{d}_\ddagger$ for the purpose of defining convergence of transfinite term graph reductions.

The notion of (transfinite) reductions carries over to GRSs straightforwardly:

Definition 8.6 (transfinite reductions)**.** Let $\mathcal{R} = (\Sigma, R)$ be a GRS. A *(transfinite) reduction* in $\mathcal{R}$ is a sequence $(g_\iota \to_\mathcal{R} g_{\iota+1})_{i<\alpha}$ of rewriting steps in $\mathcal{R}$.

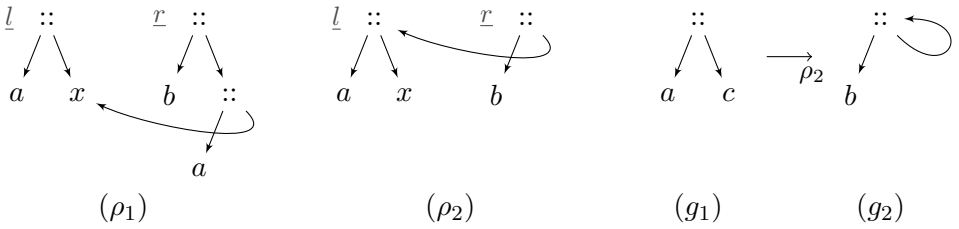
Analogously to reductions in TRSs, we need a notion of convergence in order to define well-behaved reductions. The two modes of convergence that we introduced for this very purpose in Section 5 and Section 6 are only defined on canonical term graphs. It is therefore crucial to work on reduction steps as opposed to pre-reduction steps.

Definition 8.7 (convergence of reductions)**.** Let $\mathcal{R} = (\Sigma, R)$ be a GRS.

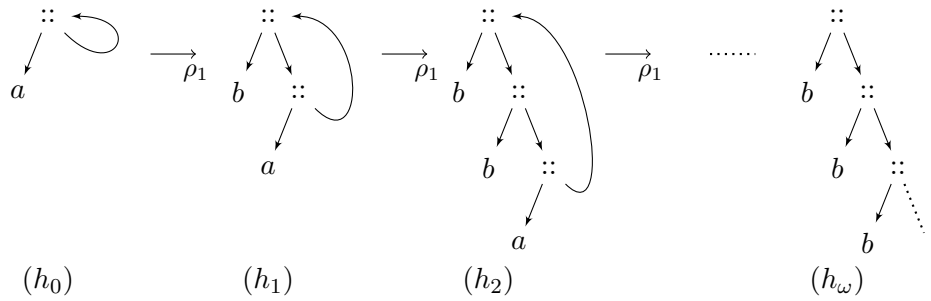
(i) Let $S = (g_\iota \to_\mathcal{R} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}$. S is *m-continuous* in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{m}_\mathcal{R} \ldots$, if the underlying sequence of term graphs $(g_\iota)_{\iota<\widehat{\alpha}}$ is continuous in $\mathcal{R}$, i.e. $\lim_{\iota\to\lambda} g_\iota = g_\lambda$ for each limit ordinal $\lambda < \alpha$. S *m-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{m}_\mathcal{R} g$, if it is m-continuous and $\lim_{\iota\to\widehat{\alpha}} g_\iota = g$.

(ii) Let $\mathcal{R}_\perp$ be the GRS $(\Sigma_\perp, R)$ over the extended signature $\Sigma_\perp$ and $S = (g_\iota \to_{\mathcal{R}_\perp} g_{\iota+1})_{\iota<\alpha}$ a reduction in $\mathcal{R}_\perp$. S is *p-continuous* in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{p}_\mathcal{R} \ldots$, if $\liminf_{\iota\to\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. S *p-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\perp)$ in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{p}_\mathcal{R} g$, if it is p-continuous and $\liminf_{\iota\to\widehat{\alpha}} g_i = g$.

(iii) Let $S = (g_\iota \to_{\mathcal{R}_\perp} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}_\perp$. The reduction S is called *p-continuous in* $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$, if it is p-continuous and $g_\iota \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ for all $\iota < \widehat{\alpha}$. The reduction S is said to *p-converge in* $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ to g, if it is p-continuous in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ and p-converges to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$.



(a) Term graph rules that unravel to $a :: x \to b :: a :: x$. (b) A single ρ_2-step.



(c) An m-convergent term graph reduction over ρ_1.

Figure 8: Term graph rules and their reductions.

Note that, analogously to p-convergence on terms, we extended the signature of $\mathcal{R}$ to $\Sigma_\perp$ for the definition of p-convergence. Like for terms, this approach serves two purposes. First, by considering the extended signature $\Sigma_\perp$, we allow any partial term graph to appear in a reduction as opposed to only total ones. Consequently, we have the whole complete semilattice $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\perp), \leq^{\mathsf{R}}_\perp)$ at our disposal, which means that p-continuity coincides with p-convergence:

Proposition 8.8. *In a GRS, every p-continuous reduction is p-convergent.*

Proof. Follows immediately from Corollary 5.16. □

The second reasons for the extension to $\mathcal{R}_\perp$ is that by not presupposing that the system's signature Σ already contains a designated $\perp$-symbol, we rule out the possibility that this $\perp$ symbol occurs in one of the rules of the system. Consequently, any $\perp$ symbol present in the final term graph of a reduction is either due to the initial term graph or the convergence behaviour. This is crucial for establishing a correspondence result between m- and p-convergence in the vein of Theorem 3.3.

Example 8.9. Consider the term graph rule ρ_1 in Figure 8a, which unravels to the term rule $a :: x \to b :: a :: x$ from Example 3.1. Starting with the term tree $a :: c$, depicted as g_1 in Figure 8b, we obtain the same transfinite reduction as in Example 3.1:

$$S\colon\ a :: c \to_{\rho_1} b :: a :: c \to_{\rho_1} b :: b :: a :: c \to_{\rho_1} \ \ldots$$

Since the modes of convergence of both the partial order $\leq^{\mathsf{R}}_\perp$ and the metric $\mathbf{d}_\ddagger$ coincide with the corresponding modes of convergence on terms (cf. Proposition 5.19 respectively Proposition 6.16), we know that, for reductions consisting

only of term trees, both m- and p-convergence in GRSs coincide with the corresponding notions of convergence in TRSs. This observation applies to the reduction S above. Hence, also in this setting of term graph rewriting, S both m- and p-converges to the term tree h_ω shown in Figure 8c. Similarly, we can reproduce the p-converging but not m-converging reduction T from Example 3.2.

Notice that h_ω is a rational term tree as it can be obtained by unravelling the finite term graph g_2 depicted in Figure 8b. In fact, if we use the rule ρ_2, which unravels to the term rule $a :: x \to b :: a :: x$ as well, we can immediately rewrite g_1 to g_2. In ρ_2, not only the variable x is shared but the whole left-hand side of the rule. This causes each redex of ρ_2 to be *captured* by the right-hand side [14].

Figure 8c indicates a transfinite reduction starting with a cyclic term graph h_0 that unravels to the rational term $t = a :: a :: a :: \ldots$. This reduction both m- and p-converges to the rational term tree h_ω as well. Again, by using ρ_2 instead of ρ_1, we can rewrite h_0 to the cyclic term graph g_2 in one step.

For more detailed explanations of the underlying modes of partial order and metric convergence for the reductions above, revisit Example 5.17 and Example 6.14, respectively.

The following theorem shows that the total fragment of p-converging reductions is in fact equivalent to m-converging reductions:

Theorem 8.10 (p-convergence in $\mathcal{G}^\infty_\mathcal{C}(\Sigma) = m$-convergence)**.** *For every reduction S in a GRS the following equivalences hold:*

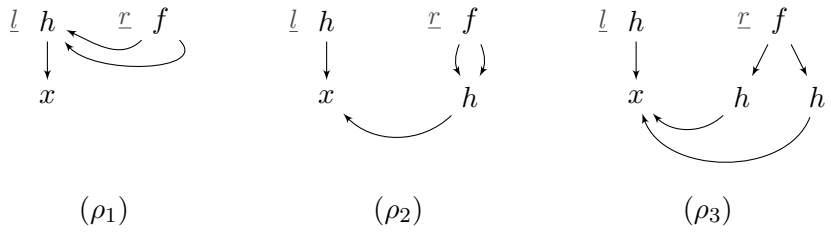
(i)	$S\colon g \xrightarrow{p}_\mathcal{R} h$	*in* $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$	*iff*	$S\colon g \xrightarrow{m}_\mathcal{R} h$
(ii)	$S\colon g \xrightarrow{p}_\mathcal{R} \ldots$	*in* $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$	*iff*	$S\colon g \xrightarrow{m}_\mathcal{R} \ldots$

Proof. We only show (i) since (ii) follows similarly.

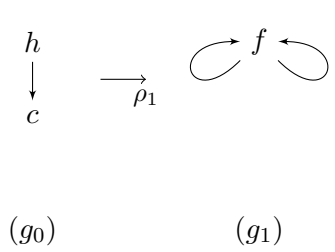
Let $S = (g_\iota \to_{\mathcal{R}_\perp} g_{\iota+1})_{\iota<\alpha}$. For the "only if" direction assume $S\colon g \xrightarrow{p}_\mathcal{R} h$ is p-converging in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$. Since S p-converges in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$, it is a reduction in $\mathcal{R}$. The p-convergence of S implies that $\liminf_{\iota\to\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. Since each g_ι is total, we have, according to Proposition 7.5, that $\lim_{\iota<\lambda} g_i = \liminf_{\iota\to\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. Hence $(g_\iota)_{\iota<\widehat{\alpha}}$ is continuous in the metric space. Likewise, we also have $\lim_{\iota<\widehat{\alpha}} g_i = \liminf_{\iota\to\widehat{\alpha}} g_i = h$. That is, S m-converges to h. For the "if" direction assume $S\colon g \xrightarrow{m}_\mathcal{R} h$. Since $(g_\iota)_{\iota<\widehat{\alpha}}$ is continuous, we have that $\lim_{\iota<\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. According to Proposition 7.3, we then have that $\liminf_{\iota\to\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. Likewise we also have $\liminf_{\iota\to\widehat{\alpha}} g_i = \lim_{\iota<\widehat{\alpha}} g_i = h$. Hence, S is p-converging to h. Since S is m-converging it is by definition also in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$. □

Example 8.11. In order to represent term rewrite rules that are not right-linear, i.e. which have multiple occurrences of the same variable on the right-hand side, we have to use sharing to represent the occurrences of a variable by a single node. Consider the term rewrite rule $h(x) \to f(h(x), h(x))$ that duplicates the variable x on the right-hand side. Note that by repeatedly applying this term rewrite rule starting with term $h(c)$, we obtain a reduction that m-converges to the full binary tree depicted in Figure 9c.

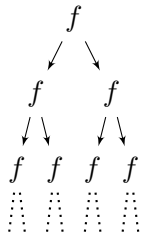
Figure 9a shows three different ways of representing the term rewrite rule $h(x) \to f(h(x), h(x))$ as a term graph rule. The rule ρ_3 has the lowest degree



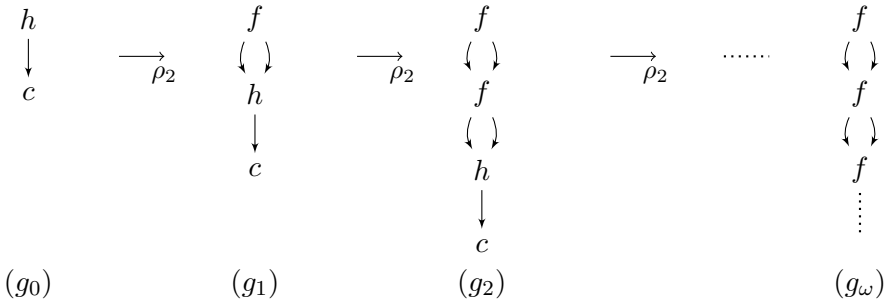
(a) Term graph rules that unravel to $h(x) \to f(h(x), h(x))$.



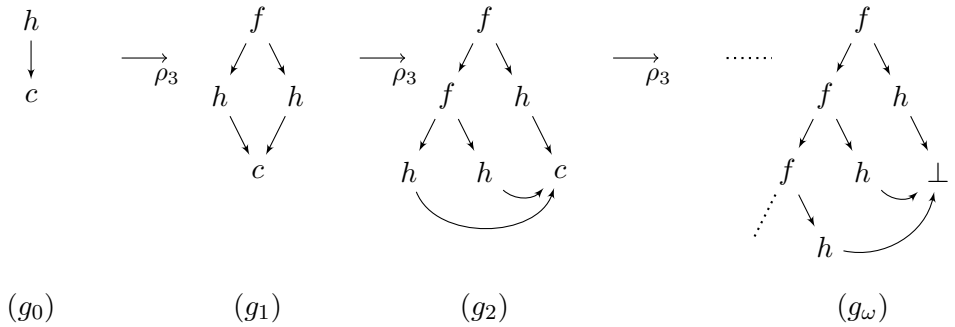
(b) A single ρ_1-step.



(c) The full binary tree of f-nodes.



(d) An m-convergent term graph reduction over ρ_2.



(e) A p-convergent term graph reduction over ρ_3.

Figure 9: Term graph rules for duplicating term rewrite rules.

of sharing since it shares the variable node directly; ρ_1 has the highest degree of sharing as it shares its complete left-hand side with its right-hand side; ρ_2 lies in between the two.

We have observed in Figure 8a before that, by sharing the complete left-hand side with the right-hand side, the redex gets captured by the right-hand side upon applying the rule to a term graph. This can be seen again in Figure 9b. By applying ρ_1 to the term tree $h(c)$ once, we immediately obtain the cyclic term graph g_1, which unravels to the full binary tree from Figure 9c.

With the rule ρ_2, we have to go through an m-convergent reduction of length ω, depicted in Figure 9d, in order to obtain the desired term graph normal form that then unravels to the full binary tree as well.

The same can also be achieved via the rule ρ_3: Starting from $h(c)$ we can construct a reduction that m-converges directly to the full binary tree in Figure 9c. However, we may also form the reduction shown in Figure 9e in which we always contract the leftmost redex. As we can see in the picture, this means that the c-node remains constantly at depth 2 while still reachable from any other node. As we explained in Example 6.14, this means that the reduction does not m-converge. On the other hand, as described in Example 5.17 the reduction p-converges to the partial term graph g_ω. In fact, from this term graph g_ω we can then construct a reduction that p-converges to the full binary tree.

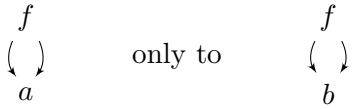
9 Term Graph Rewriting vs. Term Rewriting

In order to assess the value of the modes of convergence on term graphs that we introduced in this paper, we need to compare them to the well-established counterparts on terms. We have already observed that, if restricted to term trees, both the partial order $\leq_\perp^{\mathsf{R}}$ and the metric $\mathbf{d}_\ddagger$ on term graphs coincide with corresponding structures $\leq_\perp$ and $\mathbf{d}$ on terms, cf. Corollary 5.11 and Corollary 6.15, respectively. The same holds for the modes of convergence derived from these structures, cf. Proposition 5.19 and Proposition 6.16.

9.1 Soundness and Completeness Properties

Ideally, we would like to see a strong connection between converging reductions in a GRS $\mathcal{R}$ and converging reductions in the TRS $\mathcal{U}(\mathcal{R})$ that is its unravelling. For example, for m-convergence we want to see that $g \xrightarrow{m}_{\mathcal{R}} h$ implies $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ – i.e. soundness – and vice versa that $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ implies $g \xrightarrow{m}_{\mathcal{R}} h$ – i.e. completeness.

Completeness is already an issue for finitary rewriting [10]: a single term graph redex may corresponds to several term redexes due to sharing. Hence, contracting a term graph redex may correspond to several term rewriting steps. For example, given a rewrite rule $a \to b$, we can rewrite $f(a, a)$ to $f(a, b)$, whereas we can rewrite



which corresponds to a term reduction $f(a,a) \to f(b,b)$. That is, in the term graph we cannot choose which of the two term redexes to contract as they are represented by the same term graph redex.

Note that there are techniques to circumvent this problem by incorporating reduction steps that copy nodes in order to reduce the sharing in a term graph [31]. In this paper, however, we are only concerned with pure term graph rewriting steps derived from rewrite rules.

In the context of weak convergence, also soundness becomes an issue. The underlying reason for this issue is similar to the phenomenon explained above: a single term graph rewrite step may represent several term rewriting steps, i.e. $g \to_{\mathcal{R}} h$ implies $\mathcal{U}(g) \to^{+}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$.[2] When we have a converging term graph reduction $(g_\iota \to g_{\iota+1})_{\iota<\alpha}$, we know that the underlying sequence of term graphs (g_ι) converges. However, the corresponding term reduction does not necessarily produce the sequence $(\mathcal{U}(g_\iota))$ but may intersperse the sequence $(\mathcal{U}(g_\iota))$ with additional intermediate terms, which might change the convergence behaviour.

A similar phenomenon is know in infinitary lambda calculus[26]: while one can simulate certain term rewriting systems with lambda terms, this simulation may fail for infinitary rewriting since a single term rewriting step may require several β-reduction steps. The problem that arises in this setting is that the intermediate terms that are introduced in the lambda reduction may cause the convergence to break.

The same can, in principle, also occur when simulating a term graph reduction by a term reduction. Since a single term graph rewrite step may require several term rewrite steps, we may introduce intermediate terms into the reduction that do not directly correspond to the term graphs in the graph reduction.

9.2 Preservation of Convergence under Unravelling

Due to the abovementioned difficulties, we restrict ourselves in this paper to the soundness of the modes of convergence alone. By soundness in this setting we mean that whenever we have a sequence $(g_\iota)_{\iota<\alpha}$ of term graphs converging to g, then the sequence $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ converges to $\mathcal{U}(g)$. That is, convergence is preserved under unravelling. Since the metric $\mathbf{d}_\ddagger$ on term graphs generalises the metric $\mathbf{d}$ on terms, cf. Corollary 6.15, it does not matter whether we consider the convergence of $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ in the metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ or $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$, according to Proposition 6.16. The same also holds for the limit inferior in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_\bot)$ and $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$, due to Corollary 5.11 and Proposition 5.19.

The cornerstone of the investigation of the unravelling of term graphs is the following simple characterisation of unravelling in terms of labelled quotient trees:

Proposition 9.1. *The unravelling $\mathcal{U}(g)$ of a term graph $g \in \mathcal{G}^\infty(\Sigma)$ is given by the labelled quotient tree $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$.*

Proof. Since $\mathcal{I}_{\mathcal{P}(g)}$ is a subrelation of $\sim_g$, we know that $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$ is a labelled quotient tree and thus uniquely determines a term tree t. By Lemma 4.19, there is a homomorphism from t to g. Hence, $\mathcal{U}(g) = t$. □

[2] If the term graph g is cyclic, the corresponding term reduction may even be infinite.

Employing the above characterisation, we can easily see that the relation $\leq_\bot^{\mathsf{R}}$ is preserved under unravelling:

Proposition 9.2. *Given $g, h \in \mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$, we have that $g \leq_\bot^{R} h$ implies $\mathcal{U}(g) \leq_\bot^{R} \mathcal{U}(h)$.*

Proof. Immediate consequence of Corollary 5.10 and Proposition 9.1. □

Likewise, also least upper bounds of $\leq_\bot^{\mathsf{R}}$ are preserved:

Proposition 9.3 (preservation of lubs under unravelling)**.** *Given a directed set G in $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot), \leq_\bot^{R})$, also $\{\mathcal{U}(g) \mid g \in G\}$ is directed and $\mathcal{U}\Big(\bigsqcup_{g\in G} g\Big) = \bigsqcup_{g\in G} \mathcal{U}(g)$.*

Proof. The fact that $\{\mathcal{U}(g) \mid g \in G\}$ is directed in $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot), \leq_\bot^{\mathsf{R}})$ follows from Proposition 9.2. The equality follows from the characterisation of the lub in Theorem 5.12 and from Proposition 9.1. □

For greatest lower bounds of $\leq_\bot^{\mathsf{R}}$, the situation is more complicated as we have to consider arbitrary non-empty sets of term graphs instead of only directed sets.

We start with the characterisation of glbs in the partially ordered set of terms $(\mathcal{T}^{\infty}(\Sigma_\bot), \leq_\bot)$. Since this partially ordered set forms a complete semi-lattice, we know that it admits glbs of arbitrary non-empty sets. The following lemma characterises these glbs:

Lemma 9.4 (glb on terms)**.** *The glb $\bigsqcap T$ of a non-empty set T in $(\mathcal{T}^{\infty}(\Sigma_\bot), \leq_\bot)$ is given by the labelled quotient tree $(P, l, \mathcal{I}_P)$ where*

$$P = \left\{ \pi \in \bigcap\nolimits_{t\in T} \mathcal{P}(t) \;\middle|\; \forall \pi' < \pi \exists f \in \Sigma_\bot \forall t \in T : t(\pi') = f \right\}$$

$$l(\pi) = \begin{cases} f & \text{if } \forall t \in T : f = t(\pi) \\ \bot & \text{otherwise} \end{cases}$$

Proof. Special case of Proposition 5.9 in [9]. □

By combining the above characterisation with the characterisation of unravelled term graphs, we obtain the following:

Corollary 9.5. *Given a non-empty set G in $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot), \leq_\bot^{R})$, the glb $\bigsqcap_{g\in G} \mathcal{U}(g)$ is given by the labelled quotient tree $(P, l, \mathcal{I}_P)$ where*

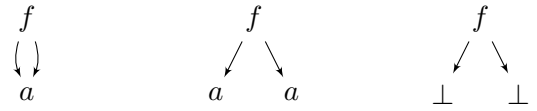
$$P = \left\{ \pi \in \bigcap\nolimits_{g\in G} \mathcal{P}(g) \;\middle|\; \forall \pi' < \pi \exists f \in \Sigma_\bot \forall g \in G : g(\pi') = f \right\}$$

$$l(\pi) = \begin{cases} f & \text{if } \forall g \in G : f = g(\pi) \\ \bot & \text{otherwise} \end{cases}$$

Proof. Follows immediately from Lemma 9.4 and Proposition 9.1. □

Before we deal with the preservation of glbs under unravelling, we need the following property that relates the unravelling of a glb to the original term graphs:

Lemma 9.6 (unravelling of a glb)**.** *For each non-empty set G in $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot), \leq_\bot^{R})$, the term $t = \mathcal{U}(\bigsqcap G)$ satisfies the following for all $g \in G$ and $\pi \in \mathcal{P}(t)$:*



(g) (h) $(g \sqcap h)$

Figure 10: Failure of preservation of glbs under unravelling.

$$\textit{(i)}\ \pi \in \mathcal{P}(g) \qquad \textit{(ii)}\ t(\pi) \in \Sigma \quad \Longrightarrow \quad t(\pi) = g(\pi)$$

Proof. Let $g \in G$, $\pi \in \mathcal{P}(t)$, and $h = \bigsqcap G$. Then $\pi \in \mathcal{P}(h)$ and $h(\pi) = t(\pi)$ according to Proposition 9.1. Since $h \leq^{\mathsf{R}}_{\bot} g$, we may apply Corollary 5.10 to obtain (i) that $\pi \in \mathcal{P}(g)$ and (ii) that $t(\pi) = g(\pi)$ whenever $t(\pi) \in \Sigma$. □

Proposition 9.7 (weak preservation of glbs under unravelling)**.** *If G is a non-empty set in $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot}), \leq^{R}_{\bot})$, then $\mathcal{U}\big(\bigsqcap_{g\in G} g\big) \leq^{R}_{\bot} \bigsqcap_{g\in G} \mathcal{U}(g)$.*

Proof. Let $s = \mathcal{U}\big(\bigsqcap_{g\in G} g\big)$ and $t = \bigsqcap_{g\in G} \mathcal{U}(g)$. Since both s and t are terms, we can use the characterisation of $\leq_{\bot}$ instead of $\leq^{\mathsf{R}}_{\bot}$. That is, we will show that for each $\pi \in \mathcal{P}(s)$, we have that $\pi \in \mathcal{P}(t)$ and that $t(\pi) = s(\pi)$ whenever $s(\pi) \in \Sigma$.

If $\pi \in \mathcal{P}(s)$, then $\pi' \in \mathcal{P}(s)$ for all $\pi' < \pi$. Since $s(\pi')$ cannot be a nullary symbol if $\pi' < \pi$, we know that $s(\pi') \neq \bot$. Hence, we can apply Lemma 9.6 in order to obtain for all $g \in G$ that $\pi \in \mathcal{P}(g)$ and that $s(\pi') = g(\pi')$ for all $\pi' < \pi$. According to Corollary 9.5, this means that $\pi \in \mathcal{P}(t)$. In order to show the second part, assume that $s(\pi) \in \Sigma$. Then, by Lemma 9.6, $g(\pi) = s(\pi)$ for all $g \in G$, which, according to Corollary 9.5, implies that $t(\pi) = s(\pi)$. □

In general, glbs are not fully preserved under unravelling as the following example shows:

Example 9.8. Consider the term graphs g and h in Figure 10. The only difference between the two term graphs is the sharing of the arguments of the root node. Due to this difference in sharing, the glb $g \sqcap h$ of the two term graphs is a proper partial term graph as depicted in Figure 10. On the other hand, since the unravelling of the two term graphs coincides, viz. $\mathcal{U}(g) = \mathcal{U}(h) = h$, we have that $\mathcal{U}(g) \sqcap \mathcal{U}(h) = h$. In particular, we have the strict inequality $\mathcal{U}(g \sqcap h) <^{\mathsf{R}}_{\bot} \mathcal{U}(g) \sqcap \mathcal{U}(h)$.

Unfortunately, this also means that the limit inferior is only weakly preserved under unravelling as well:

Theorem 9.9. *For every sequence $(g_\iota)_{\iota<\alpha}$ in $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\bot}), \leq^{R}_{\bot})$, we have that*

$$\mathcal{U}\Big(\liminf_{\iota\to\alpha} g_\iota\Big) \leq^{R}_{\bot} \liminf_{\iota\to\alpha} \mathcal{U}(g_\iota)\,.$$

Proof. This follows from Proposition 9.7 and Proposition 9.3. □

Again, we can construct a counterexample that shows that the converse inequality does not hold in general:

Example 9.10. Let $(g_\iota)_{\iota<\omega}$ be the sequence alternating between g and h from Figure 10, i.e. $g_{2\iota} = g$ and $g_{2\iota+1} = h$ for all $\iota < \omega$. Then $\bigsqcap_{\alpha\leq\iota<\omega} g_\iota = g \sqcap h$ for each $\alpha < \omega$ and, consequently, $\liminf_{\iota\to\omega} g_\iota = g \sqcap h$. As we have seen in Example 9.8, $g \sqcap h$ is the proper partial term graph depicted in Figure 10. On the other hand, since $\mathcal{U}(g) = \mathcal{U}(h) = h$, we have that $\liminf_{\iota\to\omega} \mathcal{U}(g_\iota) = h$. In particular, we have the strict inequality $\mathcal{U}(\liminf_{\iota\to\omega} g_\iota) <^{\mathsf{R}}_{\bot} \liminf_{\iota\to\omega} \mathcal{U}(g_\iota)$.

Moreover, we cannot expect that any other partial order with properties comparable to those of $\leq^{\mathsf{R}}_{\bot}$ fully preserves the limit inferior under unravelling.

The example above shows that any partial order $\leq$ on partial term graphs whose limit inferior is preserved under unravelling must also satisfy either $g \leq h$ or $h \leq g$ for the term graphs in Figure 10. That is, such a partial order has to give up the property that total term graphs are maximal, cf. Proposition 5.20. This observation is independent of whether this partial order specialises to $\leq_\bot$ on terms.

The sacrifice for full preservation under unravelling goes even further. If a partial order $\leq$ on partial term graphs satisfies preservation of its limit inferior under unravelling, the limit inferior $\liminf_{\iota\to\omega} g_\iota$ of the sequence $(g_\iota)_{\iota<\omega}$ from Example 9.10 has to unravel to h, a total term. That is, $\liminf_{\iota\to\omega} g_\iota$ has to be a total term graph. On the other hand, there is no metric – or any Hausdorff topology for that matter – for which $(g_\iota)_{\iota<\omega}$ converges at all because $(g_\iota)_{\iota<\omega}$ alternates between two distinct term graphs. In other words, the correspondence between m- and p-convergence, which we have for $\leq^{\mathsf{R}}_{\bot}$ as stated in Theorem 8.10, cannot be satisfied for such a partial order $\leq$, regardless of the metric on term graphs.

The simple partial order $\leq^{\mathsf{S}}_{\bot}$, which we briefly discussed in comparison to the rigid partial order $\leq^{\mathsf{R}}_{\bot}$ in Section 5, takes the other side of the trade-off illustrated above: it satisfies $g \leq^{\mathsf{S}}_{\bot} h$ and the preservation of the limit inferior under unravelling but sacrifices the correspondence between total term graphs and maximality as well as the correspondence between m- and p-convergence [9].

Using the correspondence between the limit inferior in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_{\bot})$ and the limit in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$, we can derive full preservation of limits under unravelling:

Theorem 9.11. *For every convergent sequence $(g_\iota)_{\iota<\alpha}$ in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$, also $(\mathcal{U}(g)_\iota)_{\iota<\alpha}$ is convergent and*

$$\mathcal{U}\Big(\lim_{\iota\to\alpha} g_\iota\Big) = \lim_{\iota\to\alpha} \mathcal{U}(g_\iota)\,.$$

Proof. We prove the equality as follows:

$$\mathcal{U}\Big(\lim_{\iota\to\alpha} g_\iota\Big) \overset{(1)}{=} \mathcal{U}\Big(\liminf_{\iota\to\alpha} g_\iota\Big) \overset{(2)}{=} \liminf_{\iota\to\alpha} \mathcal{U}(g_\iota) \overset{(3)}{=} \lim_{\iota\to\alpha} \mathcal{U}(g_\iota)$$

(1) Since $(g_\iota)_{\iota<\alpha}$ is convergent, and thus Cauchy, we can apply Proposition 7.3 to obtain that $\lim_{\iota\to\alpha} g_\iota = \liminf_{\iota\to\alpha} g_\iota$.

(2) Since $\liminf_{\iota\to\alpha} g_\iota$ is total, so is $\mathcal{U}(\liminf_{\iota\to\alpha} g_\iota)$. By Proposition 5.20, this means that $\mathcal{U}(\liminf_{\iota\to\alpha} g_\iota)$ is maximal w.r.t. $\leq^{\mathsf{R}}_{\bot}$. Consequently, the inequality $\mathcal{U}(\liminf_{\iota\to\alpha} g_\iota) \leq^{\mathsf{R}}_{\bot} \liminf_{\iota\to\alpha} \mathcal{U}(g_\iota)$ due to Theorem 9.9 yields (2).

(3) This equality follows from Proposition 7.5 and the fact that $\liminf_{\iota\to\alpha} \mathcal{U}(g_\iota)$ is total.

□

9.3 Finite Representations of Transfinite Term Reductions

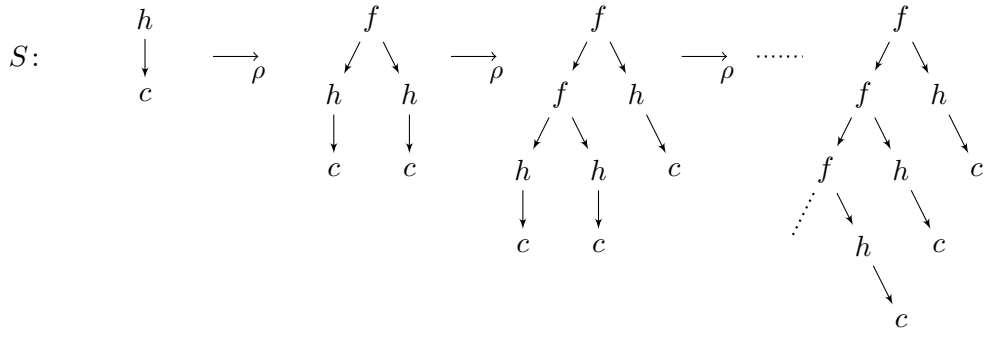
One of the motivations for considering modes of convergence on term graphs in the first place is the study of finite representation of transfinite term reductions as finite term graph reductions. Since both the metric $\mathbf{d}_\ddagger$ and the partial order $\leq^{\mathsf{R}}_\bot$ specialise to the corresponding structures on terms, we can use both the metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ and the partially ordered set $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_\bot)$ to move seamlessly from terms to term graphs and vice versa.

For instance, Example 8.11 illustrates reductions that perform essentially the same computations, however, at different levels of sharing / parallelism. This includes the complete lack of sharing as well, i.e. term rewriting. For each of the cases we can use the partially ordered set $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{R}}_\bot)$ and the metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$ as a unifying framework to determine the convergence behaviour.

In order to use the partial order $\leq^{\mathsf{R}}_\bot$ and the metric $\mathbf{d}_\ddagger$ as a tool to study finite representability of infinite term reductions as finite term graph reductions there still is some work to be done, though.

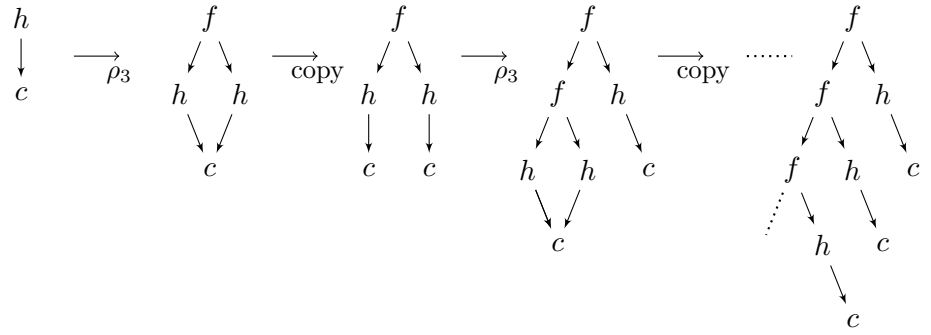
First and foremost, we need a unifying framework for performing both term and term graph rewriting. A straightforward approach to achieve this, is to include copying steps in term graph reductions that allow us to revert the sharing produced by applying term graph rules [31]. For example while the rule ρ_3 from Figure 9a is the term graph rule with the least sharing that unravels to $\rho\colon\ h(x) \to f(h(x), h(x))$, it still has some sharing in order to represent the duplication of the variable x on the right-hand side.

The result of this sharing is seen in Figure 9e, which shows that even if we start with a term tree, the rule ρ_3 turns it into a proper term graph. Consequently it is slightly different from the corresponding term reduction



In fact, while the above term reduction S is m-convergent, the term graph reduction via ρ_3, depicted in Figure 9e, is not.

However, by interspersing the term graph reduction with reduction steps that copy nodes – and in general sub-term graphs – we instead obtain the following term graph reduction:



This reduction simulates the corresponding term reduction S more closely and in fact both reductions m-converge to the same term. Nevertheless, this approach creates the same issue that we have already noted for soundness: the additional term graphs that get interspersed with the original term reduction may affect the convergence behaviour.

The second ingredient that we need in order to study the finite representability of transfinite term reductions is a compression property [25] for transfinite term graph reductions that allows us to compress a transfinite term graph reduction that ends in a finite term graph to a term graph reduction of finite length.

Unfortunately, experience from infinitary term rewriting already shows us that a general compression property – allowing any reduction to be compressed to length at most ω – is not possible for weak convergence [23]. However, the more restrictive version of the compression property that we need, viz. that reductions ending in a finite term graph may be compressed to finite length, does hold for weakly m-converging term reductions [27] and there is hope that this carries over to the term graph rewriting setting.

10 Conclusions & Future Work

With the goal of generalising infinitary term rewriting to term graphs, we have presented two different modes of convergence for an infinitary calculus of term graph rewriting. The success of this generalisation effort was demonstrated by a number of results. Many of the properties of the modes of convergence on terms have been maintained in this transition to term graphs. First and foremost, this includes the intrinsic completeness properties of the underlying structures, i.e. the metric space is still complete and the partially ordered set still forms a complete semilattice. Moreover, we were also able to maintain the correspondence between p- and m-convergence as well as the intuition of the partial order to capture a notion of information preservation.

An important check for the appropriateness of the modes of convergence on term graphs is their relation to the corresponding modes of convergence on terms. For both the partial order and the metric approach, we have that convergence on term graphs is a conservative extension of the convergence on terms. Conversely, convergence on term graphs carries over to convergence on terms via the unravelling mapping. Unfortunately, this preservation of convergence under unravelling is only weak in the case of the partial order setting; cf. Theorem 9.9. However,

as we have explained in Section 9.2, this phenomenon is an unavoidable side effect of the generalisation to term graphs unless other important properties are sacrificed. Fortunately, this phenomenon vanishes in the metric setting and we in fact obtain full preservation of limits under unravelling; cf. Theorem 9.11.

As a result, we have obtained two modes of convergence, which allow us to combine both infinitary term rewriting and term graph rewriting within one theoretical framework. Our motivation for this effort is derived from studying *lazy evaluation* and the correspondence between infinitary term rewriting and finitary term graph rewriting. For both applications, we still require more understanding of the matter, though: for the former, we still lack at least a treatment of higher-order rewriting whereas we are much closer to the latter. We have discussed issues concerning the correspondence between infinitary term rewriting and finitary term graph rewriting in detail in Section 9.3: while the unified modes of convergence are already helpful for studying infinitary rewriting with a varying degree of sharing, we identified two shortcomings that have to be addressed, viz. the lack of a unifying notion of rewriting for terms and term graphs and a compression property for transfinite term graph reductions.

Apart from the abovementioned issues, future work should also be concerned with establishing a stronger correspondence between infinitary term rewriting and infinitary term graph rewriting beyond the preservation of limits under unravellings, which we showed in this paper. Despite the difficulties that we encountered in Section 9.1, we think that obtaining such results is possible. However, a more promising way of approaching this issue is to restrict the notion of convergence to strong convergence as we know it from infinitary term rewriting [25]. Such a stricter notion of convergence takes the location of a reduction step into consideration and, thus, provides a closer correspondence between term graph reductions and their term rewriting counterparts. Indeed, this technique has been applied successfully to convergence on term graphs based on the simple partial order $\leq_{\perp}^{\mathsf{S}}$, which we briefly discussed in comparison to the rigid partial order $\leq_{\perp}^{\mathsf{R}}$ in Section 5, and a corresponding metric [9].

Acknowledgement

The author would like to thank the anonymous referees of RTA 2011 as well as the referees for the special issue of LMCS whose comments greatly helped to improve the presentation of the material.

Bibliography

[1] Z. Ariola and S. Blom. Skew and ω-Skew Confluence and Abstract Böhm Semantics. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838 of *Lecture Notes in Computer Science*, pages 368–403. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30911-6. doi: 10.1007/11601548_19.

[2] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with

letrec. *Annals of Pure and Applied Logic*, 117(1-3):95–168, 2002. ISSN 0168-0072. doi: 10.1016/S0168-0072(01)00104-X.

[3] Z. M. Ariola and J. W. Klop. Lambda Calculus with Explicit Recursion. *Information and Computation*, 139(2):154–233, 1997. ISSN 0890-5401. doi: 10.1006/inco.1997.2651.

[4] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[5] P. Bahr. Infinitary Rewriting - Theory and Applications. Master's thesis, Vienna University of Technology, Vienna, 2009.

[6] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[7] P. Bahr. Partial Order Infinitary Term Rewriting and Böhm Trees. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.67.

[8] P. Bahr. Convergence in Infinitary Term Graph Rewriting Systems is Simple. Submitted to Math. Struct. in Comp. Science, 2012.

[9] P. Bahr. Infinitary Term Graph Rewriting is Simple, Sound and Complete. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69–84, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2012.69.

[10] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In P. C. T. de Bakker A. J. Nijman, editor, *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Berlin / Heidelberg, 1987. doi: 10.1007/3-540-17945-3_8.

[11] E. Barendsen. Term Graph Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 13, pages 712–743. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[12] S. Blom. An Approximation Based Approach to Infinitary Lambda Calculi. In V. van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin / Heidelberg, 2004. doi: 10.1007/b98160.

[13] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90040-9.

[14] W. M. Farmer and R. J. Watro. Redex capturing in term graph rewriting. *International Journal of Foundations of Computer Science*, 1:369–386, 1990. ISSN 0129-0541. doi: 10.1142/S0129054190000266.

[15] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1): 68–95, 1977. ISSN 0004-5411. doi: 10.1145/321992.321997.

[16] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 95–103, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811543.

[17] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

[18] G. Kahn and G. D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2):187–277, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90090-G.

[19] S. Kahrs. Infinitary rewriting: meta-theory and convergence. *Acta Informatica*, 44(2):91–121, 2007. ISSN 0001-5903 (Print) 1432-0525 (Online). doi: 10.1007/s00236-007-0043-2.

[20] J. L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955. ISBN 0387901256.

[21] R. Kennaway. On transfinite abstract reduction systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

[22] R. Kennaway. Infinitary Rewriting and Cyclic Graphs. In *SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 153–166, 1995. doi: 10.1016/S1571-0661(05)80192-6.

[23] R. Kennaway and F.-J. de Vries. Infinitary Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 12, pages 668–711. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[24] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. ISSN 0164-0925. doi: 10.1145/177492.177577.

[25] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[26] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)00171-5.

[27] S. Lucas. Transfinite Rewriting Semantics for Term Rewriting Systems. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 216–230. Springer Berlin / Heidelberg, 2001. doi: 10.1007/3-540-45127-7_17.

[28] S. Marlow. Haskell 2010 Language Report, 2010.

[29] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. ISBN 013453333X.

[30] R. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0201416638.

[31] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 3–61. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. ISBN 981-02-4020-1.

[32] J. G. Simonsen. Weak Convergence and Uniform Normalization in Infinitary Rewriting. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 311–324, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.311.

[33] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

A Proof of Lemma 5.14

Lemma 5.14 (compatible elements have lub)**.** *Each pair g_1, g_2 of compatible term graphs in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$ has a least upper bound.*

Proof of Lemma 5.14. Since $\{g_1, g_2\}$ is not necessarily directed, its lub might have positions that are neither in g_1 nor in g_2. Therefore, it is easier to employ a different construction here: Following Remark 5.13, we will use the structure $(\mathcal{G}^\infty(\Sigma_\bot)/_\cong, \leq^\mathsf{R}_\bot)$ which is isomorphic to $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^\mathsf{R}_\bot)$. To this end, we will construct a term graph $\overline{g}$ such that $[\overline{g}]_\cong$ is the lub of $\{[g_1]_\cong, [g_2]_\cong\}$. Since we assume that $\{[g_1]_\cong, [g_2]_\cong\}$ has an upper bound, say $[\widehat{g}]_\cong$, there are two rigid $\bot$-homomorphisms $\phi_i \colon g_i \to_\bot \widehat{g}$.

Let $g_j = (N^j, \mathsf{suc}^j, \mathsf{lab}^j, r^j)$, $j = 1, 2$. Since we are dealing with isomorphism classes, we can assume w.l.o.g. that the nodes in g_j are of the form n^j for $j = 1, 2$. Let $\overline{M} = N^1 \uplus N^2$ and define the relation $\sim$ on $\overline{M}$ as follows:

$$n^j \sim m^k \quad \text{iff} \quad \mathcal{P}_{g_j}(n^j) \cap \mathcal{P}_{g_k}(m^k) \neq \emptyset$$

$\sim$ is clearly reflexive and symmetric. Hence, its transitive closure $\sim^+$ is an equivalence relation on $\overline{M}$. Now define the term graph $\overline{g} = (\overline{N}, \overline{\mathsf{lab}}, \overline{\mathsf{suc}}, \overline{r})$ as follows:

$$\overline{N} = \overline{M}/_{\sim^+} \qquad \overline{\mathsf{lab}}(N) = \begin{cases} f & \text{if } f \in \Sigma, \exists n^j \in N.\ \mathsf{lab}^j(n^j) = f \\ \bot & \text{otherwise} \end{cases}$$

$$\overline{r} = [r^1]_{\sim^+} \qquad \overline{\mathsf{suc}}_i(N) = N' \quad \text{iff} \quad \exists n^j \in N.\ \mathsf{suc}_i^j(n^j) \in N'$$

Note that since $\langle\rangle \in \mathcal{P}_{g_1}(r^1) \cap \mathcal{P}_{g_2}(r^2)$, we also have $\overline{r} = [r^2]_{\sim^+}$.

Before we argue about the well-definedness of $\overline{g}$, we need to establish some auxiliary claims:

$$n^j \sim^+ m^k \implies \phi_j(n^j) = \phi_k(m^k) \qquad \text{for all } n^j, m^k \in \overline{M} \tag{1}$$

$$\phi_j(n^j) = \phi_k(m^k) \implies n^j \sim m^k \qquad \text{for all } n^j, m^k \in \overline{M} \text{ with } \mathsf{lab}^j(n^j), \mathsf{lab}^k(m^k) \in \Sigma \tag{1'}$$

We show (1) by proving that $n^j \sim^p m^k$ implies $\phi_j(n^j) = \phi_k(m^k)$ by induction on $p > 0$. If $p = 1$, then $n^j \sim m^k$. Hence, $\mathcal{P}_{g_j}(n^j) \cap \mathcal{P}_{g_k}(m^k) \neq \emptyset$. Additionally, from Lemma 4.10 we obtain both $\mathcal{P}_{g_j}(n^j) \subseteq \mathcal{P}_{\widehat{g}}(\phi_j(n^j))$ and $\mathcal{P}_{g_k}(m^k) \subseteq \mathcal{P}_{\widehat{g}}(\phi_k(m^k))$. Consequently, we also have that $\mathcal{P}_{\widehat{g}}(\phi_j(n^j)) \cap \mathcal{P}_{\widehat{g}}(\phi_k(m^k)) \neq \emptyset$, i.e. $\phi_j(n^j) = \phi_k(m^k)$. If $p = q + 1 > 1$, then there is some $o^l \in \overline{M}$ with $n^j \sim o^l$ and $o^l \sim^q m^k$. Applying the induction hypothesis immediately yields $\phi_j(n^j) = \phi_l(o^l) = \phi_k(m^k)$.

For (1'), let $n^j, m^k \in \overline{M}$ with $\mathsf{lab}^j(n^j), \mathsf{lab}^k(m^k) \in \Sigma$ and $\phi_j(n^j) = \phi_k(m^k)$. Since ϕ_j and ϕ_k are rigid $\bot$-homomorphisms, we have the following equations:

$$\mathcal{P}^a_{g_j}(n^j) = \mathcal{P}^a_{\widehat{g}}(\phi_j(n^j)) = \mathcal{P}^a_{\widehat{g}}(\phi_k(m^k)) = \mathcal{P}^a_{g_k}(m^k).$$

Hence, $\mathcal{P}_{g_j}(n^j) \cap \mathcal{P}_{g_k}(m^k) \neq \emptyset$ and, therefore, $n^j \sim m^k$.

Next we show that $\overline{\mathsf{lab}}$ is well-defined. To this end, let $N \in \overline{N}$ and $n^j, m^k \in N$ such that $\mathsf{lab}^j(n^j) = f_1 \in \Sigma$ and $\mathsf{lab}^k(m^k) = f_2 \in \Sigma$. We need to show that $f_1 = f_2$. By (1), we have that $\phi_j(n^j) = \phi_k(m^k)$. Since $f_1, f_2 \in \Sigma$, we can employ the labelling condition for ϕ_j and ϕ_k in order to obtain that

$$f_1 = \mathsf{lab}^j(n^j) = \widehat{\overline{\mathsf{lab}}}(\phi_j(n^j)) = \widehat{\overline{\mathsf{lab}}}(\phi_k(m^k)) = \mathsf{lab}^k(m^k) = f_2.$$

To argue that $\overline{\mathsf{suc}}$ is well-defined, we first have to show for all $N \in \overline{N}$ that $\overline{\mathsf{suc}}_i(N)$ is defined iff $i < \mathsf{ar}(\overline{\mathsf{lab}}(N))$. Suppose that $\overline{\mathsf{suc}}_i(N)$ is defined. Then there is some $n^j \in N$ such that $\mathsf{suc}_i^j(n^j)$ is defined. Hence, $i < \mathsf{ar}(\mathsf{lab}^j(n^j))$. Since then also $\mathsf{lab}^j(n^j) \in \Sigma$, we have $\overline{\mathsf{lab}}(N) = \mathsf{lab}^j(n^j)$. Therefore, $i < \mathsf{ar}(\overline{\mathsf{lab}}(N))$. If, conversely, there is some $i \in \mathbb{N}$ with $i < \mathsf{ar}(\overline{\mathsf{lab}}(N))$, then we know that $\overline{\mathsf{lab}}(N) = f \in \Sigma$. Hence, there is some $n^j \in N$ with $\mathsf{lab}^j(n^j) = f$. Hence, $i < \mathsf{ar}(\mathsf{lab}^j(n^j))$ and, therefore, $\mathsf{suc}_i^j(n^j)$ is defined. Hence, $\overline{\mathsf{suc}}_i(N)$ is defined.

To finish the argument showing that $\overline{\mathsf{suc}}$ is well-defined, we have to show that, for all $N, N_1, N_2 \in \overline{N}$ and $n^j, m^k \in N$ such that $\mathsf{suc}_i^j(n^j) \in N_1$ and $\mathsf{suc}_i^k(m^k) \in N_2$, we indeed have $N_1 = N_2$. As $n^j, m^k \in N$, we have $n^j \sim^+ m^k$ and, therefore,

$\phi_j(n^j) = \phi_k(n^k)$ according to (1). Since both $\mathsf{suc}_i^j(n^j)$ and $\mathsf{suc}_i^k(m^k)$ are defined, we have $\mathsf{lab}^j(n^j), \mathsf{lab}^k(m^k) \in \Sigma$. By (1') we then have $n^j \sim m^k$, i.e. there is some $\pi \in \mathcal{P}_{g_j}(n^j) \cap \mathcal{P}_{g_k}(m^k)$. Consequently, $\pi \cdot \langle i \rangle \in \mathcal{P}_{g_j}(\mathsf{suc}_i^j(n^j)) \cap \mathcal{P}_{g_k}(\mathsf{suc}_i^k(m^k))$. Hence, $\mathsf{suc}_i^j(n^j) \sim \mathsf{suc}_i^k(m^k)$ and, therefore, $N_1 = N_2$.

Before we begin the main argument we need establish the following auxiliary claims:

$$\mathcal{P}_{g_j}(n^j) \subseteq \mathcal{P}_{\overline{g}}([n^j]_{\sim^+}) \quad \text{for all } n^j \in \overline{M} \tag{2}$$

$$\forall \pi \in \mathcal{P}^a_{\overline{g}}(N)\ \exists n^j \in N.\ \mathsf{lab}^j(n^j) \in \Sigma, \pi \in \mathcal{P}^a_{g_j}(n^j) \quad \begin{array}{l}\text{for all } N \in \overline{N} \\ \text{with } \overline{\mathsf{lab}}(N) \in \Sigma\end{array} \tag{3}$$

$$n^j \sim^+ m^k \quad \Longrightarrow \quad \mathcal{P}^a_{g_j}(n^j) = \mathcal{P}^a_{g_k}(m^k) \quad \begin{array}{l}\text{for all } n^j, m^j \in \overline{M} \text{ with} \\ \mathsf{lab}^j(n^j), \mathsf{lab}^k(m^k) \in \Sigma\end{array} \tag{4}$$

For (2), we will show that $\pi \in \mathcal{P}_{g_j}(n^j)$ implies $\pi \in \mathcal{P}_{\overline{g}}([n^j]_{\sim^+})$ by induction on the length of π. The case $\pi = \langle\rangle$ is trivial. If $\pi = \pi' \cdot \langle i \rangle$, then $\pi' \cdot \langle i \rangle \in \mathcal{P}_{g_j}(n^j)$, i.e. for $m^j = \mathsf{node}_{g_j}(\pi')$, we have $\mathsf{suc}_i^j(m^j) = n^j$. Employing the induction hypothesis, we obtain $\pi' \in \mathcal{P}_{\overline{g}}([m^j]_{\sim^+})$. Additionally, according to the construction of $\overline{g}$, we have $\overline{\mathsf{suc}}_i([m^j]_{\sim^+}) = [n^j]_{\sim^+}$. Consequently, $\pi' \cdot \langle i \rangle \in \mathcal{P}_{\overline{g}}([n^j]_{\sim^+})$ holds.

Similarly, we also show (3) by induction on the length of π. If $\pi = \langle\rangle$, then we have $\langle\rangle \in \mathcal{P}^a_{\overline{g}}(N)$, i.e. $N = \overline{r}$. Since, by assumption, $\overline{\mathsf{lab}}(\overline{r}) \in \Sigma$ holds, there is some $j \in \{1, 2\}$ such that $\mathsf{lab}^j(r^j) \in \Sigma$. Moreover, we clearly have $\langle\rangle \in \mathcal{P}^a_{g_j}(r^j)$. If $\pi = \pi' \cdot \langle i \rangle$, then we have $\pi' \cdot \langle i \rangle \in \mathcal{P}^a_{\overline{g}}(N)$. Let $N' = \mathsf{node}_{\overline{g}}(\pi')$. Since $\pi' \cdot \langle i \rangle$ is acyclic in $\overline{g}$, so is π', i.e. $\pi' \in \mathcal{P}^a_{\overline{g}}(N')$. Moreover, we have that $\overline{\mathsf{suc}}_i(N')$ is defined, i.e. $\overline{\mathsf{lab}}(N')$ is not nullary and in particular not $\bot$. Thus, we can apply the induction hypothesis to obtain some $n^j \in N'$ with $\mathsf{lab}^j(n^j) \in \Sigma$ and $\pi' \in \mathcal{P}^a_{g_j}(n^j)$. Hence, according to the construction of $\overline{g}$, we have $\mathsf{lab}^j(n^j) = \overline{\mathsf{lab}}(N')$, i.e. $\mathsf{suc}_i^j(n^j) = m^j$ is defined. Furthermore, we then get $m^j \in N$. Since $\pi' \cdot \langle i \rangle \in \mathcal{P}_{g_j}(m^j)$, it remains to be shown that $\pi' \cdot \langle i \rangle$ is acyclic in g_j. Suppose that $\pi' \cdot \langle i \rangle$ is cyclic in g_j. As π' is acyclic in g_j, this means that there is some position $\pi^* \in \mathcal{P}_{g_j}(m^j)$ with $\pi^* < \pi' \cdot \langle i \rangle$. Using (2), we obtain that $\pi^* \in \mathcal{P}_{\overline{g}}(N)$. This contradicts the assumption of $\pi' \cdot \langle i \rangle$ being acyclic in $\overline{g}$. Hence, $\pi' \cdot \langle i \rangle$ is acyclic.

For (4), suppose that $n^j \sim^+ m^k$ holds with $\mathsf{lab}^j(n^j), \mathsf{lab}^k(m^k) \in \Sigma$. From (1), we obtain $\phi_j(n^j) = \phi_k(n^k)$. Moreover, since both n^j and m^k are not labelled with $\bot$, we know that ϕ_j and ϕ_k are rigid in n^j and m^k, respectively, which yields the equations

$$\mathcal{P}^a_{g_j}(n^j) = \mathcal{P}^a_{\widehat{g}}(\phi_j(n^j)) = \mathcal{P}^a_{\widehat{g}}(\phi_k(m^k)) = \mathcal{P}^a_{g_k}(m^k).$$

Next we show that $[g_1]_{\cong}, [g_1]_{\cong} \leq^{\mathsf{R}}_{\bot} [\overline{g}]_{\cong}$ holds by giving two rigid $\bot$-homomorphisms $\psi_j \colon g_j \to_{\bot} \overline{g}$, $j = 1, 2$. Define $\psi_j \colon N^j \to \overline{N}$ by $n^j \mapsto [n^j]_{\sim^+}$. From (2) and the fact that, according to the construction of $\overline{g}$, $\mathsf{lab}^j(n^j) \in \Sigma$ implies $\mathsf{lab}^j(n^j) = \overline{\mathsf{lab}}([n^j]_{\sim^+})$, we immediately get that ψ_j is a $\bot$-homomorphism by applying Lemma 4.10. In order to argue that ψ_j is rigid, assume that $n^j \in N^j$ with $\mathsf{lab}^j(n^j) \in \Sigma$. According to Lemma 5.7, it suffices to show that $\mathcal{P}^a_{\overline{g}}(\psi_j(n^j)) \subseteq \mathcal{P}_{g_j}(n^j)$. Suppose that $\pi \in \mathcal{P}^a_{\overline{g}}(\psi_j(n^j))$. Note that, by the construction of $\overline{g}$, $\psi_j(n^j)$ is not labelled with $\bot$ either. Hence, we can apply (3) to obtain some

$m^k \in \psi_j(n^j)$ with $\mathsf{lab}^k(m^k) \in \Sigma$ and $\pi \in \mathcal{P}^a_{g_k}(m^k)$. By definition, $m^k \in \psi_j(n^j)$ is equivalent to $n^j \sim^+ m^k$. Therefore, we can employ (4), which yields $\mathcal{P}^a_{g_k}(m^k) = \mathcal{P}^a_{g_j}(n^j)$. Hence, $\pi \in \mathcal{P}^a_{g_j}(n^j)$.

Note that the construction of $\overline{g}$ did not depend on $\widehat{g}$, viz. for any other upper bound $[\widehat{h}]_\cong$ of $[g_1]_\cong, [g_2]_\cong$, we get the same term graph $\overline{g}$. Hence, it is still just an arbitrary upper bound which means that in order to show that $[\overline{g}]_\cong$ is the least upper bound, it suffices to show $[\overline{g}]_\cong \leq^{\mathsf{R}}_{\perp} [\widehat{g}]_\cong$. For this purpose, we will devise a rigid $\perp$-homomorphism $\psi\colon \overline{g} \to_\perp \widehat{g}$. Define $\psi\colon \overline{N} \to \widehat{N}$ by $[n^j]_{\sim^+} \mapsto \phi_j(n^j)$. (1) shows that ψ is well-defined. The root condition for ψ follows from the root condition for ϕ_1:

$$\psi(\overline{r}) = \psi([r^1]_{\sim^+}) = \phi_1(r^1) = \widehat{r}.$$

For the labelling condition, assume that $\overline{\mathsf{lab}}(N) = f \in \Sigma$ for some $N \in \overline{N}$. Then there is some $n^j \in N$ with $\mathsf{lab}^j(n^j) = f$. Therefore, the labelling condition for ϕ_j yields

$$\widehat{\mathsf{lab}}(\psi(N)) = \widehat{\mathsf{lab}}(\phi_j(n^j)) = \mathsf{lab}^j(n^j) = f$$

For the successor condition, let $\overline{\mathsf{suc}}_i(N) = N'$ for some $N, N' \in \overline{N}$. Then there is some $n^j \in N$ with $\mathsf{suc}^j_i(n^j) \in N'$. Therefore, the successor condition for ψ follows from the successor condition for ϕ_j as follows:

$$\begin{aligned}\psi(\overline{\mathsf{suc}}_i(N)) = \psi(N') &= \psi([\mathsf{suc}^j_i(n^j)]_{\sim^+}) = \phi_j(\mathsf{suc}^j_i(n^j)) \\ &= \widehat{\mathsf{suc}}_i(\phi_j(n^j)) = \widehat{\mathsf{suc}}_i(\psi([n^j]_{\sim^+})) = \widehat{\mathsf{suc}}_i(\psi(N))\end{aligned}$$

Finally, we show that ψ is rigid. To this end, let $N \in \overline{N}$ with $\overline{\mathsf{lab}}(N) \in \Sigma$. That is, there is some $n^j \in N$ with $\mathsf{lab}^j(n^j) \in \Sigma$. Recall, that we have shown that $\psi_j\colon g^j \to_\perp \overline{g}$ is rigid. That is, we have

$$\mathcal{P}^a_{g_j}(n^j) = \mathcal{P}^a_{\overline{g}}(\psi_j(n^j)) = \mathcal{P}^a_{\overline{g}}([n^j]_{\sim^+}).$$

Analogously, we have $\mathcal{P}^a_{\widehat{g}}(\phi_j(n^j)) = \mathcal{P}^a_{g_j}(n^j)$ as ϕ_j is rigid, too. Using this, we can obtain the following equations:

$$\mathcal{P}^a_{\widehat{g}}(\psi(N)) = \mathcal{P}^a_{\widehat{g}}(\psi([n^j]_{\sim^+})) = \mathcal{P}^a_{\widehat{g}}(\phi_j(n^j)) = \mathcal{P}^a_{g_j}(n^j) = \mathcal{P}^a_{\overline{g}}([n^j]_{\sim^+}) = \mathcal{P}^a_{\overline{g}}(N)$$

Hence, ψ is a rigid $\perp$-homomorphism from $\overline{g}$ to $\widehat{g}$. □

B Proof of Lemma 6.10

In this appendix, we will give the full proof of Lemma 6.10. Before we can do this we have to establish a number of technical auxiliary lemmas.

The lemma below will serve as a tool for the two lemmas that are to follow afterwards. We know that the set of retained nodes $N^g_{<d}$ contains at least all nodes at depth smaller than d due to the closure condition (T1). However, due to the closure condition (T2) also nodes at a larger depth may be included in $N^g_{<d}$. The following lemma shows that this is not possible for nodes labelled with nullary symbols:

Lemma B.1 (labelling)**.** *Let $g \in \mathcal{G}^\infty(\Sigma)$, $\Delta \subseteq \Sigma^{(0)}$ and $d < \omega$. If Δ-$\mathsf{depth}(g) \geq d$, then $\mathsf{lab}^g(n) \notin \Delta$ for all $n \in N^g_{<d}$.*

Proof. We will show that $N_\nabla = \{n \in N^g \mid \mathsf{lab}^g(n) \notin \Delta\}$ satisfies the properties (T1) and (T2) of Definition 6.1 for the term graph g and depth d. Since $N^g_{<d}$ is the least such set, we then obtain $N^g_{<d} \subseteq N_\nabla$ and, thereby, the claimed statement.

For (T1), let $n \in N^g$ with $\mathsf{depth}_g(n) < d$. Since $\Delta\text{-}\mathsf{depth}(g) \geq d$, we have $\mathsf{lab}^g(n) \notin \Delta$ and, therefore, $n \in N_\nabla$. For (T2), let $n \in N_\nabla$ and $m \in \mathsf{Pre}^a_g(n)$. Then m cannot be labelled with a nullary symbol, a fortiori $\mathsf{lab}^g(m) \notin \Delta$. Hence, we have $m \in N_\nabla$. □

The following two lemmas are rather technical. They state that rigid Δ-homomorphisms preserve retained nodes and in a stricter sense also fringe nodes.

Lemma B.2 (preservation of retained nodes)**.** *Suppose $g, h \in \mathcal{G}^\infty(\Sigma)$, $d < \omega$, $\phi\colon g \to_\Delta h$ is rigid, and $\Delta\text{-}\mathsf{depth}(g) \geq d$. Then $\phi(N^g_{<d}) = N^h_{<d}$.*

Proof. Let $N_\nabla = \{n \in N^g \mid \mathsf{lab}^g(n) \notin \Delta\}$. At first we will show that $\phi(N^g_{<d}) \subseteq N^h_{<d}$. To this end, we will show that $\phi^{-1}(N^h_{<d}) \cap N_\nabla$ satisfies (T1) and (T2) of Definition 6.1 for term graph g and depth d. Since $N^g_{<d}$ is the least such set, we then obtain $N^g_{<d} \subseteq \phi^{-1}(N^h_{<d}) \cap N_\nabla$ and, a fortiori, $N^g_{<d} \subseteq \phi^{-1}(N^h_{<d})$ which is equivalent to $\phi(N^g_{<d}) \subseteq N^h_{<d}$.

For (T1), let $n \in N^g$ with $\mathsf{depth}_g(n) < d$. Because $\Delta\text{-}\mathsf{depth}(g) \geq d$, we know that $\mathsf{lab}^g(n) \notin \Delta$, which means by Lemma 6.7 that we also have $\mathsf{depth}_h(\phi(n)) < d$. Hence, $\phi(n) \in N^h_{<d}$ by (T1). Since $\mathsf{lab}^g(n) \notin \Delta$, we thus have $n \in \phi^{-1}(N^h_{<d}) \cap N_\nabla$.

For (T2), let $n \in \phi^{-1}(N^h_{<d}) \cap N_\nabla$. That is, we have $\phi(n) \in N^h_{<d}$ and $\mathsf{lab}^g(n) \notin \Delta$. Hence, by (T2), it holds that $\mathsf{Pre}^a_h(\phi(n)) \subseteq N^h_{<d}$. We have to show now that $\mathsf{Pre}^a_g(n) \subseteq \phi^{-1}(N^h_{<d}) \cap N_\nabla$. Let $m \in \mathsf{Pre}^a_g(n)$. That is, there is some $\pi \cdot \langle i \rangle \in \mathcal{P}^a_g(n)$ with $\pi \in \mathcal{P}_g(m)$. As $\mathsf{lab}^g(n) \notin \Delta$ and ϕ is rigid, we know that ϕ is rigid in n. Consequently, $\pi \cdot \langle i \rangle \in \mathcal{P}^a_h(\phi(n))$. Moreover, we have $\pi \in \mathcal{P}_h(\phi(m))$ by Lemma 4.10. Hence, $\phi(m) \in \mathsf{Pre}^a_h(\phi(n))$ and, therefore, $\phi(m) \in N^h_{<d}$ by (T2). Additionally, as m has a successor in g, it cannot be labelled with a symbol in Δ. Hence, $m \in \phi^{-1}(N^h_{<d}) \cap N_\nabla$.

In order to prove the converse inclusion $\phi(N^g_{<d}) \supseteq N^h_{<d}$, we will show that $\phi(N^g_{<d})$ satisfies (T1) and (T2) for term graph h and depth d. This will prove the abovementioned inclusion since $N^h_{<d}$ is the least such set.

For (T1), let $n \in N^h$ with $\mathsf{depth}_h(n) < d$. By Lemma 6.5, there is some $m \in N^g$ with $\mathsf{depth}_g(m) < d$ and $\phi(m) = n$. Hence, according to (T1), we have $m \in N^g_{<d}$ and, therefore, $n \in \phi(N^g_{<d})$.

For (T2), let $n \in \phi(N^g_{<d})$. That is, there is some $m \in N^g_{<d}$ with $\phi(m) = n$. By (T2), we have $\mathsf{Pre}^a_g(m) \subseteq N^g_{<d}$. We must show that $\mathsf{Pre}^a_h(n) \subseteq \phi(N^g_{<d})$. Let $n' \in \mathsf{Pre}^a_h(n)$. That is, there is some $\pi \cdot \langle i \rangle \in \mathcal{P}^a_h(n)$ with $\pi \in \mathcal{P}_h(n')$. Since $m \in N^g_{<d}$, we have $\mathsf{lab}^g(m) \notin \Delta$ by Lemma B.1. Consequently, ϕ is rigid in m which yields that $\pi \cdot \langle i \rangle \in \mathcal{P}^a_g(m)$. Note that then also $\pi \in \mathcal{P}(g)$. Let $m' = \mathsf{node}_g(\pi)$. Thus, $m' \in \mathsf{Pre}^a_g(m)$ and, therefore, $m' \in N^g_{<d}$ according to (T2). Moreover, because $\pi \in \mathcal{P}_g(m') \cap \mathcal{P}_h(n')$, we are able to obtain from Lemma 4.10 that $\phi(m') = n'$. Hence, $n' \in \phi(N^g_{<d})$. □

Lemma B.3 (preservation of fringe nodes)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$, $\phi\colon g \to_\Delta h$ rigid, $0 < d < \omega$, $\Delta\text{-}\mathsf{depth}(g) \geq d$, $n \in N^g$, and $0 \leq i < \mathsf{ar}_g(n)$. Then $n^i \in N^g_{=d}$ iff $\phi(n)^i \in N^h_{=d}$.*

Proof. Note that, by Lemma B.1, we have that $\mathsf{lab}^g(n) \notin \Delta$ for all nodes $n \in N^g_{<d}$. Additionally, by Lemma B.2, we obtain $\phi(N^g_{<d}) = N^h_{<d}$ and, therefore, according to the labelling condition for ϕ, we get that $\mathsf{lab}^h(n) \notin \Delta$ for all $n \in N^h_{<d}$.

At first we will show the "only if" direction. To this end, let $n^i \in N^g_{=d}$. By definition, we then have $\mathsf{depth}_g(n) \geq d-1$. Hence, by Lemma 6.7, $\mathsf{depth}_h(\phi(n)) \geq d-1$. Furthermore, we have that $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ or $n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$. We show now that in either case we can conclude $\phi(n)^i \in N^h_{=d}$.

Let $\mathsf{suc}^g_i(n) \notin N^g_{<d}$. If we have $\mathsf{suc}^h_i(\phi(n)) \notin N^h_{<d}$, then $\phi(n)^i \in N^h_{=d}$. So suppose $\mathsf{suc}^h_i(\phi(n)) \in N^h_{<d}$. Since $N^h_{<d} = \phi(N^g_{<d})$, according to Lemma B.2, we find some $m \in N^g_{<d}$ with $\phi(m) = \mathsf{suc}^h_i(\phi(n))$. However, since $\mathsf{suc}^g_i(n) \notin N^g_{<d}$, we know that $m \neq \mathsf{suc}^g_i(n)$. We now show $\phi(n) \notin \mathsf{Pre}^a_h(\mathsf{suc}^h_i(\phi(n)))$ by showing that $\pi \cdot \langle i \rangle \notin \mathcal{P}^a_h(\mathsf{suc}^h_i(\phi(n)))$ whenever $\pi \in \mathcal{P}^a_h(\phi(n))$:

$$\begin{aligned}
\pi \in \mathcal{P}^a_h(\phi(n)) &\iff \pi \in \mathcal{P}^a_g(n) && (\phi \text{ is rigid in } n)\\
&\implies \pi \cdot \langle i \rangle \notin \mathcal{P}^a_g(m) && (m \neq \mathsf{suc}^g_i(n))\\
&\iff \pi \cdot \langle i \rangle \notin \mathcal{P}^a_h(\phi(m)) && (\phi \text{ is rigid in } m)\\
&\iff \pi \cdot \langle i \rangle \notin \mathcal{P}^a_h(\mathsf{suc}^h_i(\phi(n))) && (\phi(m) = \mathsf{suc}^h_i(\phi(n)))
\end{aligned}$$

Together with $\mathsf{depth}_h(\phi(n)) \geq d-1$, this implies that $\phi(n)^i \in N^h_{=d}$.

Let $n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$. If $\phi(n) \notin \mathsf{Pre}^a_h(\mathsf{suc}^h_i(\phi(n)))$, then $\phi(n)^i \in N^h_{=d}$. So suppose that $\phi(n) \in \mathsf{Pre}^a_h(\mathsf{suc}^h_i(\phi(n)))$. Hence, $\phi(n) \in \mathsf{Pre}^a_h(\phi(\mathsf{suc}^g_i(n)))$ as ϕ is homomorphic in n. If $\mathsf{lab}^g(\mathsf{suc}^g_i(n)) \notin \Delta$, then ϕ is rigid in $\mathsf{suc}^g_i(n)$ and we would also get $n \in \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$ which contradicts the assumption. Hence, $\mathsf{lab}^g(\mathsf{suc}^g_i(n)) \in \Delta$ and, therefore, $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ according to Lemma B.1. Thus, we can employ the argument for this case that we have already given above.

We now turn to the converse direction. For this purpose, let $\phi(n)^i \in N^h_{=d}$. Then $\mathsf{depth}_h(\phi(n)) \geq d-1$ and, consequently $\mathsf{depth}_g(n) \geq d-1$ by Lemma 6.7. Additionally, we also have $\mathsf{suc}^h_i(\phi(n)) \notin N^h_{<d}$ or $\phi(n) \notin \mathsf{Pre}^a_h(\mathsf{suc}^h_i(\phi(n)))$. Again we will show that in either case we can conclude $n^i \in N^g_{=d}$.

If $\mathsf{suc}^h_i(\phi(n)) \notin N^h_{<d}$, then $\phi(\mathsf{suc}^g_i(n)) \notin N^h_{<d}$ and, therefore, $\phi(\mathsf{suc}^g_i(n)) \notin \phi(N^g_{<d})$ according to Lemma B.2. Consequently, $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ which implies that $n^i \in N^g_{=d}$.

Let $\phi(n) \notin \mathsf{Pre}^a_h(\mathsf{suc}^h_i(\phi(n)))$. If $n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$, then we get $n^i \in N^g_{=d}$ immediately. So assume that $n \in \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$. If $\mathsf{lab}^g(\mathsf{suc}^g_i(n)) \notin \Delta$, then ϕ would be rigid in $\mathsf{suc}^g_i(n)$. Thereby, we would get $\phi(n) \in \mathsf{Pre}^a_h(\phi(\mathsf{suc}^g_i(n)))$ which contradicts the assumption. Hence, $\mathsf{lab}^g(\mathsf{suc}^g_i(n)) \in \Delta$. According to Lemma B.1, we then have $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ and, therefore, $n^i \in N^g_{=d}$. □

The above lemma depends on the peculiar definition of fringe nodes – in particular those fringe nodes that are due to the condition

$$\mathsf{depth}_g(n) \geq d-1 \text{ and } n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n)).$$

Recall that this condition produces a fringe node for each edge from a retained node that closes a cycle. Let us have a look at the term graph g depicted in Figure 11. The rigid truncation $g‡2$ of g is shown in Figure 7a. If the above-mentioned alternative condition for fringe nodes would not be present, then the

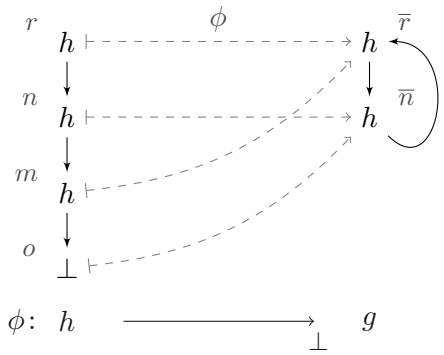


Figure 11: Fringe nodes and rigid $\bot$-homomorphisms.

set $N^g_{=2}$ would be empty (and, thus, $g‡2 = g$). Then, however, the rigid $\bot$-homomorphism ϕ illustrated in Figure 11 would violate Lemma B.3. Since the node m is cut off from h in the truncation $h‡2$, there is a fringe node n^0 in $h‡2$. On the other hand, there would be no fringe node $\overline{n}^0$ in $g‡2$ if not for the alternative condition above.

Lemma 6.10 ($\leq^{\mathsf{R}}_{\bot}$ and rigid truncation)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d < \omega$ with $g \leq^R_\bot h$ and $\bot$-$\mathsf{depth}(g) \geq d$, we have that $g‡d \cong h‡d$.*

Proof of Lemma 6.10. For $d = 0$, this is trivial. So assume $d > 0$. Since $g \leq^{\mathsf{R}}_{\bot} h$, there is a rigid $\bot$-homomorphism $\phi\colon\ g \to_\bot h$. Define the function ψ as follows:

$$\begin{aligned}\psi\colon\ & N^{g‡d} \to N^{h‡d}\\ & N^g_{<d} \ni n \mapsto \phi(n)\\ & N^g_{=d} \ni n^i \mapsto \phi(n)^i\end{aligned}$$

At first we have to argue that ψ is well-defined. For this purpose, we first need that $\phi(N^g_{<d}) \subseteq N^{g‡d}$. Lemma B.2 confirms this. Secondly, we need that $n^i \in N^g_{=d}$ implies $\phi(n)^i \in N^{g‡d}$. This is guaranteed by Lemma B.3.

Next we show that ψ is a homomorphism from $g‡d$ to $h‡d$. The root condition is inherited from ϕ as $r^{g‡d} \in N^g_{<d}$. Note that, according to Lemma B.1, we have $\mathsf{lab}^g(n) \in \Sigma$ for all $n \in N^g_{<d}$. Hence, ϕ is homomorphic in $N^g_{<d}$ which means that the labelling condition for nodes in $N^g_{<d}$ is also inherited from ϕ. For nodes $n^i \in N^g_{=d}$, we have $\mathsf{lab}^{g‡d}(n^i) = \bot$. Since, by definition, $\psi(n^i) \in N^h_{=d}$, we can conclude $\mathsf{lab}^{h‡d}(\psi(n^i)) = \bot$.

The successor condition is trivially satisfied by nodes in $N^g_{=d}$ as they do not have any successors. Let $n \in N^g_{<d}$ and $0 \leq i < \mathsf{ar}_{g‡d}(n)$. We distinguish two cases: At first assume that $n^i \notin N^g_{=d}$. Hence, $\mathsf{suc}^{g‡d}_i(n) = \mathsf{suc}^g_i(n) \in N^g_{<d}$. Since, by Lemma B.3, also $\phi(n)^i \notin N^h_{=d}$, we additionally have $\mathsf{suc}^{h‡d}_i(\phi(n)) = \mathsf{suc}^h_i(\phi(n))$. Hence, using the successor condition for ϕ, we can reason as follows:

$$\begin{aligned}\psi(\mathsf{suc}^{g‡d}_i(n)) &= \psi(\mathsf{suc}^g_i(n)) = \phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^h_i(\phi(n))\\ &= \mathsf{suc}^{h‡d}_i(\phi(n)) = \mathsf{suc}^{h‡d}_i(\psi(n))\end{aligned}$$

If, on the other hand, $n^i \in N^g_{=d}$, then $\mathsf{suc}_i^{g\ddagger d}(n) = n^i$. Moreover, since then $\phi(n)^i \in N^h_{=d}$ by Lemma B.3, we have $\mathsf{suc}_i^{h\ddagger d}(\phi(n)) = \phi(n)^i$, too. Hence, we can reason as follows:

$$\psi(\mathsf{suc}_i^{g\ddagger d}(n)) = \psi(n^i) = \phi(n)^i = \mathsf{suc}_i^{h\ddagger d}(\phi(n)) = \mathsf{suc}_i^{h\ddagger d}(\psi(n))$$

This shows that ψ is a homomorphism. Note that, according to Lemma 5.6, ϕ is injective in $N^g_{<d}$. Then also ψ is injective in $N^g_{<d}$. For the same reason, ψ is also injective in $N^g_{=d}$. Moreover, we have $\psi(N^g_{<d}) \subseteq N^h_{<d}$ and $\psi(N^g_{=d}) \subseteq N^h_{=d}$, i.e. $\psi(N^g_{<d}) \cap \psi(N^g_{=d}) = \emptyset$. Hence, ψ is injective which implies, by Lemma 4.12, that ψ is an isomorphism from $g\ddagger d$ to $h\ddagger d$. □

Convergence in Infinitary Term Graph Rewriting Systems is Simple

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

Term graph rewriting provides a formalism for implementing term rewriting in an efficient manner by avoiding duplication. Infinitary term rewriting has been introduced to study infinite term reduction sequences. Such infinite reductions can be used to model *non-strict evaluation*. In this paper, we unify term graph rewriting and infinitary term rewriting thereby addressing both components of *lazy evaluation*: non-strictness and sharing.

In contrast to previous attempts to formalise infinitary term graph rewriting, our approach is based on a simple and natural generalisation of the modes of convergence of infinitary term rewriting. We show that this new approach is better suited for infinitary term graph rewriting as it is simpler and more general. The latter is demonstrated by the fact that our notions of convergence give rise to two independent canonical and exhaustive constructions of infinite term graphs from finite term graphs via metric and ideal completion. In addition, we show that our notions of convergence on term graphs are sound w.r.t. the ones employed in infinitary term rewriting in the sense that convergence is preserved by unravelling term graphs to terms. Moreover, the resulting infinitary term graph calculi provide a unified framework for both infinitary term rewriting and term graph rewriting, which makes it possible to study the correspondences between these two worlds more closely.

Contents

1 Introduction

Term graphs are a generalisation of terms, which allow us to avoid duplication of subterms and instead use pointers in order to refer to the same subterm several times. In this paper, we aim to extend the theory of infinitary term rewriting to the setting of term graphs.

As the basis for our infinitary calculi we use the well-established term graph rewriting formalism of Barendregt et al. [10] as it will allows us to draw on the work investigating the relation between (infinitary) term rewriting on the one hand and term graph rewriting on the other hand [21].

In order to devise an infinitary calculus, we have to conceive a notion of convergence that constrains reductions of transfinite length in a meaningful way. To this end, we generalise the metric on terms that is used to define convergence for infinitary term rewriting [13] to term graphs. In a similar way, we generalise

the partial order on terms that has been recently used to define a closely related notion of convergence for infinitary term rewriting [7]. The use of two different – but on terms closely related – approaches to convergence will allow us both to assess the appropriateness of the resulting infinitary calculi and to compare them against the corresponding infinitary calculi of term rewriting.

The focus of the present work is primarily on the foundational aspects of infinitary term graph rewriting. That is, our major concerns are the underlying notions of convergence and their appropriateness. That is why we only consider weak forms of convergence, i.e. notions of convergence that are purely based on the convergence of the terms respectively term graphs along a reduction, as opposed to strong convergence [22] that also considers the positions of contracted redexes.

1.1 Motivation

1.1.1 Lazy Evaluation

Term rewriting is a useful formalism for studying declarative programs, in particular, functional programs. A functional program essentially consists of functions defined by a set of equations and an expression that is supposed to be evaluated according to these equations. The conceptual process of evaluating an expression is nothing else than term rewriting.

A particularly interesting feature of modern functional programming languages, such as Haskell [23], is the ability to deal with conceptually infinite data structures. For example, the following function `from` constructs for each number n the infinite list of consecutive numbers starting from n:

```
from(n) = n :: from(s(n))
```

Here, we use the binary infix symbol `::` to denote the list constructor cons and `s` for the successor function. While we cannot use the infinite list generated by `from` directly – the evaluation of an expression of the form `from n` does not terminate – we can use it in a setting in which we only read a finite prefix of the infinite list conceptually defined by `from`. Functional languages such as Haskell allow this use of semantically infinite data structures through a *non-strict evaluation* strategy, which delays the evaluation of a subexpression until its result is actually required for further evaluation of the expression. This non-strict semantics is not only a conceptual neatness but in fact one of the major features that make functional programs highly modular [16]!

The above definition of the function `from` can be represented as a term rewriting system with the following rule:

$$from(x) \to x :: from(s(x))$$

Starting with the term $from(0)$, we then obtain the following infinite reduction:

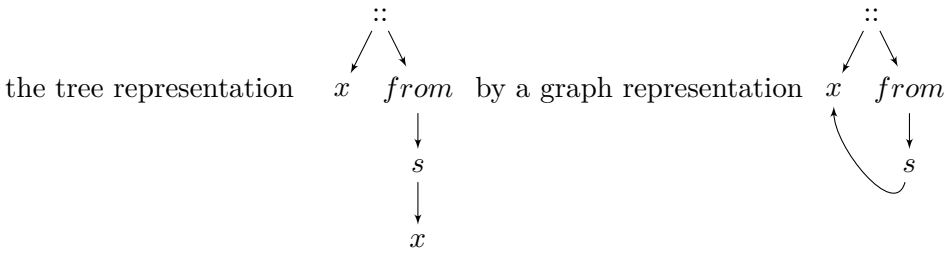
$$from(0) \to 0 :: from(s(0)) \to 0 :: s(0) :: from(s(s(0))) \to \ldots$$

The theory of infinitary term rewriting [20] provides a notion of convergence that may assign a meaningful limit term to such an infinite reduction provided

there exists one. In this sense, the above reduction converges to the infinite term $0 :: s(0) :: s(s(0)) :: \ldots$, which represents the infinite list of numbers $0, 1, 2, \ldots$. Due to this extension of term rewriting with explicit limit constructions for non-terminating reductions, infinitary term rewriting allows us to directly reason about non-terminating functions and infinite data structures.

Non-strict evaluation is rarely found unescorted, though. Usually, it is implemented as lazy evaluation [15], which complements a non-strict evaluation strategy with *sharing*. The latter avoids duplication of subexpressions by using pointers instead of copying. For example, the function `from` above duplicates its argument `n` – it occurs twice on the right-hand side of the defining equation. A lazy evaluator simulates this duplication by inserting two pointers pointing to the actual argument. Sharing is a natural companion for non-strict evaluation as it avoids re-evaluation of expressions that are duplicated before they are evaluated.

The underlying formalism that is typically used to obtain sharing for functional programming languages is term graph rewriting [24, 25]. Term graph rewriting [10, 26] uses graphs to represent terms thus allowing multiple arcs to point to the same node. Term graphs allows us, e.g. for the right-hand side $x :: from(s(x))$ of the term rewrite rule defining the function $from$, to replace

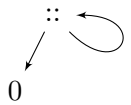


which shares the variable x by having two arcs pointing to it.

While infinitary term rewriting is used to model the non-strictness of lazy evaluation, term graph rewriting models the sharing part of it. By endowing term graph rewriting with a notion of convergence, we aim to unify the two formalisms into one calculus, thus allowing us to model both aspects withing the same calculus.

1.1.2 Rational Terms

Term graphs can do more than only share common subexpressions. Through cycles term graphs may also provide a finite representation of certain infinite terms – so-called rational terms. For example, the infinite term $0 :: 0 :: 0 :: \ldots$ can be represented as the finite term graph



Since a single node on a cycle in a term graph represents infinitely many corresponding subterms, the contraction of a single term graph redex may correspond to a transfinite term reduction that contracts infinitely many term redexes. For example, if we apply the rewrite rule $0 \to s(0)$ to the above term graph, we obtain

a term graph that represents the term $s(0) :: s(0) :: s(0) :: \ldots$, which can only be obtained from the term $0 :: 0 :: 0 :: \ldots$ via a *transfinite* term reduction with the rule $0 \to s(0)$. Kennaway et al. [21] investigated this correspondence between cyclic term graph rewriting and infinitary term rewriting. Among other results they characterise a subset of transfinite term reductions – called rational reductions – that can be simulated by a corresponding finite term graph reduction. The above reduction from the term $0 :: 0 :: 0 :: \ldots$ is an example of such a rational reduction.

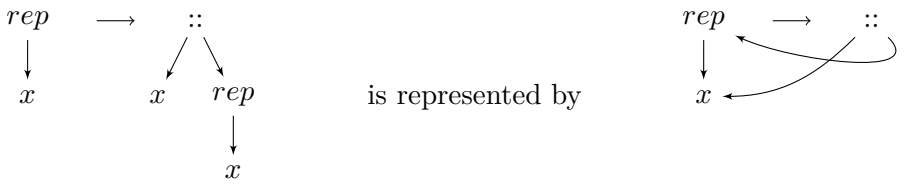
With the help of a unified formalism for infinitary and term graph rewriting, it should be easier to study the correspondence between infinitary term rewriting and finitary term graph rewriting further. The move from an infinitary term rewriting system to a term graph rewriting system is then only a change in the degree of sharing if we use infinitary term graph rewriting as a common framework.

For example, consider the term rewrite rule $rep(x) \to x :: rep(x)$, which defines a function rep that repeats its argument infinitely often:

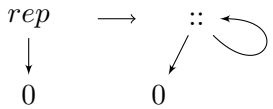
$$rep(0) \to 0 :: rep(0) \to 0 :: 0 :: rep(0) \to 0 :: 0 :: 0 :: rep(0) \to \quad \ldots \quad 0 :: 0 :: 0 :: \ldots$$

This reduction happens to be not a rational reduction in the sense of Kennaway et al. [21].

The move from the term rule $rep(x) \to x :: rep(x)$ to a term graph rule is a simple matter of introducing sharing of common subexpressions:



Instead of creating a fresh copy of the redex on the right-hand side, the redex is reused by placing an edge from the right-hand side of the rule to its left-hand side. This allows us to represent the infinite reduction approximating the infinite term $0 :: 0 :: 0 :: \ldots$ with the following single step term graph reduction induced by the above term graph rule:



Via its cyclic structure the resulting term graph represents the infinite term $0 :: 0 :: 0 :: \ldots$.

Since both transfinite term reductions and the corresponding finite term graph reductions can be treated within the same formalism, we hope to provide a tool for studying the ability of cyclic term graph rewriting to finitely represent transfinite term reductions.

1.2 Contributions & Related Work

1.2.1 Contributions

The main contributions of this paper are:

1. We devise a simple partial order on term graphs based on graph homomorphisms. We show that this partial order forms a complete semilattice and thus is technically suitable for defining a notion of convergence.

2. We devise a simple metric on term graphs and show that it forms a complete ultrametric space on term graphs.

3. Based on the partial order respectively the metric we define a notion of weak convergence for infinitary term graph rewriting. We show that the partial order convergence subsumes the metric convergence.

4. We confirm that the partial order and the metric on term graphs generalise the partial order respectively the metric that is used for infinitary term rewriting. Moreover, we show that the corresponding notions of convergence are preserved by unravelling term graphs to terms thus establishing the soundness of our notions of convergence on term graphs w.r.t. the convergence on terms.

5. Finally, we show that both the partial order and the metric provide completion constructions – ideal completion and metric completion, respectively – that construct the set of finite and infinite term graphs from the set of finite term graphs.

In this paper we study the foundations of infinitary term graph rewriting and therefore focus purely on weak notions of convergence, i.e. notions that are based on the sequence of term graphs produced along a term graph reduction. Similar to infinitary term rewriting, weak notions of convergence for infinitary term graph rewriting are difficult to study and often manifest some unexpected behaviour. In particular, soundness and completeness properties w.r.t. infinitary term rewriting are hard to come by. Yet, we gathered much evidence that support the appropriateness of our infinitary calculi. More evidence can be found when moving to strong convergence, which does exhibit solid soundness and completeness properties w.r.t. infinitary term rewriting [9].

1.2.2 Related Work

Calculi with explicit sharing and/or recursion, e.g. via *letrec*, can also be considered as a form of term graph rewriting. Ariola and Klop [3] recognised that adding such an explicit recursion mechanism to the lambda calculus may break confluence. In order to reconcile this, Ariola and Blom [1, 2] developed a notion of skew confluence that allows them to define an infinite normal form in the vein of Böhm trees.

In previous work, we have investigated notions of convergence for term graph rewriting [8]. The approach that we have taken in that work is very similar to the approach adopted in this paper: by generalising the metric and the partial

order on terms to term graphs, we devised a weak notion of convergence for infinitary term graph rewriting. However, both the metric and the partial order on term graphs are very carefully crafted in order to make them very similar to the corresponding structures on terms. While the thus obtained two notions of convergence manifest the same correspondence that is known from infinitary term rewriting [7], they are too restrictive as we will illustrate in this paper. Due to the close resemblance to the convergence on terms, these notions of convergence are not able to capture all forms of sharing appropriately.

In this paper, we follow a different approach by taking the arguably simplest generalisation of the metric and the partial order to term graphs. We will show that this approach is better suited for infinitary term graph rewriting as it lifts the restrictions that we observe in our previous formalisation [8].

1.3 Overview

The structure of this paper is as follows: in Section 2, we give an overview of infinitary term rewriting including the necessary background for metric spaces and partially ordered sets. Section 3 provides the necessary theory for graphs and term graphs. Sections 4 and 5 form the core of this paper. In these sections we study the partial order and the metric on term graphs that are the basis for the notions of convergence we consider in this paper. In Section 6, we use these two notions of convergence to study two corresponding infinitary term graph rewriting calculi. Sections 7 and 8 are concerned with forms of soundness and completeness properties of our notions of convergence. In the former, we show that both notions of convergence generalise the corresponding notions of convergence on terms and that they are preserved under unravelling term graphs to terms. In the latter, we show that the set of (finite and infinite) term graphs arises both as the metric completion and the ideal completion of the set of finite term graphs.

2 Infinitary Term Rewriting

For devising an infinitary calculus, we have to devise a notion of convergence that constrains transfinite reductions in a meaningful way. Before pondering over the right approach to an infinitary calculus of term graph rewriting, we want to provide a brief overview of infinitary term rewriting [7, 20]. In this paper, we will only consider weak notions of convergence, i.e. convergence is solely determined by the sequence of terms respectively term graphs that are produced along a reduction [13].

We assume the reader to be familiar with the basic theory of ordinal numbers, orders and topological spaces [18], as well as term rewriting [27]. In the following, we briefly recall the most important notions.

2.1 Sequences

We use the von Neumann definition of ordinal numbers. That is, an *ordinal number* (or simply *ordinal*) α is the set of all ordinal numbers strictly smaller than α. In particular, each natural number $n \in \mathbb{N}$ is an ordinal number with $n = \{0, 1, \ldots, n-1\}$. The least infinite ordinal number is denoted by ω and is

the set of all natural numbers. Ordinal numbers will be denoted by lower case Greek letters $\alpha, \beta, \gamma, \delta, \lambda, \iota$.

A *sequence* S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*. For a finite sequence $(a_i)_{i<n}$, we also use the notation $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. In particular, $\langle\rangle$ denotes the empty sequence. We write A^* for the set of all finite sequences in A.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences $(a_\iota)_{\iota<\alpha}$ and $(b_\iota)_{\iota<\beta}$ is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a (proper) *prefix* of a sequence T, denoted $S \leq T$ (respectively $S < T$), if there is a (non-empty) sequence S' with $S \cdot S' = T$. The prefix of T of length $\beta \leq |T|$ is denoted $T|_\beta$. The thus defined binary prefix relation $\leq$ forms a complete semilattice (cf. Section 2.3). Similarly, a sequence S is a (proper) *suffix* of a sequence T if there is a (non-empty) sequence S' with $S' \cdot S = T$.

2.2 Metric Spaces

Given a set M, a pair $(M, \mathbf{d})$ is called a *metric space* if $\mathbf{d}\colon M \times M \to \mathbb{R}_0^+$ is a function satisfying $\mathbf{d}(x, y) = 0$ iff $x = y$ (identity), $\mathbf{d}(x, y) = \mathbf{d}(y, x)$ (symmetry), and $\mathbf{d}(x, z) \leq \mathbf{d}(x, y) + \mathbf{d}(y, z)$ (triangle inequality), for all $x, y, z \in M$. If $\mathbf{d}$, instead of the triangle inequality, satisfies the stronger property $\mathbf{d}(x, z) \leq \max\{\mathbf{d}(x, y), \mathbf{d}(y, z)\}$ (strong triangle), then $(M, \mathbf{d})$ is called an *ultrametric space*. Let $(a_\iota)_{\iota<\alpha}$ be a sequence in a metric space $(M, \mathbf{d})$. The sequence $(a_\iota)_{\iota<\alpha}$ *converges* to an element $a \in M$, written $\lim_{\iota\to\alpha} a_\iota$, if, for each $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that $\mathbf{d}(a, a_\iota) < \varepsilon$ for every $\beta < \iota < \alpha$; $(a_\iota)_{\iota<\alpha}$ is *continuous* if $\lim_{\iota\to\lambda} a_\iota = a_\lambda$ for each limit ordinal $\lambda < \alpha$. The sequence $(a_\iota)_{\iota<\alpha}$ is called *Cauchy* if, for any $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that, for all $\beta < \iota < \gamma < \alpha$, we have that $\mathbf{d}(m_\iota, m_\gamma) < \varepsilon$. A metric space is called *complete* if each of its non-empty Cauchy sequences converges.

Given two metric spaces $(M_1, \mathbf{d}_1)$ and $(M_2, \mathbf{d}_2)$, a function $\phi\colon M_1 \to M_2$ is called an *isometric embedding* of $(M_1, \mathbf{d}_1)$ into $(M_2, \mathbf{d}_2)$ if it preserves distances, i.e.

$$\mathbf{d}_2(\phi(x), \phi(y)) = \mathbf{d}_1(x, y) \quad \text{for all } x, y \in M_1.$$

If, additionally, ϕ is bijective, then it is called an *isometry* and the metric spaces $(M_1, \mathbf{d}_1)$ and $(M_2, \mathbf{d}_2)$ are said to be *isometric*.

2.3 Partial Orders

A *partial order* $\leq$ on a set A is a binary relation on A that is *transitive*, *reflexive*, and *antisymmetric*. The pair $(A, \leq)$ is then called a *partially ordered set*. A subset D of the underlying set A is called *directed* if it is non-empty and each pair of elements in D has an upper bound in D. A partially ordered set $(A, \leq)$ is called a *complete partial order* (*cpo*) if it has a least element and each directed set D has a *least upper bound* (*lub*) $\bigsqcup D$. A cpo $(A, \leq)$ is called a *complete semilattice* if every *non-empty* set B has *greatest lower bound* (*glb*) $\bigsqcap B$. In particular, this means that for any sequence $(a_\iota)_{\iota<\alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha} \left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists.

There is also a different characterisation of complete semilattices in terms of bounded complete cpos: a partially ordered set $(A, \leq)$ is called *bounded complete* if each set $B \subseteq A$ that has an upper bound in A also has a least upper bound in A.

Proposition 2.1 (complete semilattice, Kahn and Plotkin [17])**.** *Given a cpo $(A, \leq)$, the following are equivalent:*

(i) $(A, \leq)$ is a complete semilattice.

(ii) $(A, \leq)$ is bounded complete.

Given two partially ordered sets $(A, \leq_A)$ and $(B, \leq_B)$, a function $\phi\colon A \to B$ is called *monotonic* if $a_1 \leq_A a_2$ implies $\phi(a_1) \leq_B \phi(a_2)$. In particular, a sequence $(b_\iota)_{\iota<\alpha}$ in $(B, \leq_B)$ is called monotonic if $\iota \leq \gamma < \alpha$ implies $b_\iota \leq_B b_\gamma$. An *order isomorphism* from $(A, \leq_A)$ to $(B, \leq_B)$ is a monotonic function $\phi\colon A \to B$ such that there is a monotonic function $\psi\colon B \to A$ which is the inverse of ϕ, i.e. $\psi \circ \phi$ and $\phi \circ \psi$ are identity functions on A respectively B. If there is an order isomorphism from $(A, \leq_A)$ to $(B, \leq_B)$, then $(A, \leq_A)$ and $(B, \leq_B)$ are called *order isomorphic*.

With the prefix order $\leq$ on sequences we can generalise concatenation to arbitrary sequences of sequences: let $(S_\iota)_{\iota<\alpha}$ be a sequence of sequences in some set A. The concatenation of $(S_\iota)_{\iota<\alpha}$, written $\prod_{\iota<\alpha} S_\iota$, is recursively defined as the empty sequence $\langle\rangle$ if $\alpha = 0$, $(\prod_{\iota<\alpha'} S_\iota) \cdot S_{\alpha'}$ if $\alpha = \alpha' + 1$, and $\bigsqcup_{\gamma<\alpha} \prod_{\iota<\gamma} S_\iota$ if α is a limit ordinal.

2.4 Terms

Since we are interested in the infinitary calculus of term rewriting, we consider the set $\mathcal{T}^\infty(\Sigma)$ of (potentially infinite) *terms* over some *signature* Σ. A *signature* Σ is a countable set of symbols. Each symbol f has an associated arity $\mathsf{ar}(f) \in \mathbb{N}$, and we write $\Sigma^{(n)}$ for the set of symbols in Σ which have arity n. The set $\mathcal{T}^\infty(\Sigma)$ is defined as the *greatest* set such that $t \in \mathcal{T}^\infty(\Sigma)$ implies $t = f(t_0, \ldots, t_{k-1})$ for some $f \in \Sigma^{(k)}$ and $t_0, \ldots, t_{k-1} \in \mathcal{T}^\infty(\Sigma)$. For each nullary symbol $c \in \Sigma^{(0)}$, we write c for the term $c()$. For a term $t \in \mathcal{T}^\infty(\Sigma)$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. $\mathcal{P}(t)$ is the least subset of $\mathbb{N}^*$ such that $\langle\rangle \in \mathcal{P}(t)$ and $\langle i\rangle \cdot \pi \in \mathcal{P}(t)$ if $t = f(t_0, \ldots, t_{k-1})$ with $0 \leq i < k$ and $\pi \in \mathcal{P}(t_i)$. For terms $s, t \in \mathcal{T}^\infty(\Sigma)$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, $t(\pi)$ for the function symbol in t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s. The set $\mathcal{T}(\Sigma)$ of *finite terms* is the set of terms $t \in \mathcal{T}^\infty(\Sigma)$ for which $\mathcal{P}(t)$ is a finite set.

On $\mathcal{T}^\infty(\Sigma)$ a similarity measure $\mathsf{sim}\colon \mathcal{T}^\infty(\Sigma) \times \mathcal{T}^\infty(\Sigma) \to \omega+1$ can be defined by setting

$$\mathsf{sim}(s,t) = \min\{|\pi| \mid \pi \in \mathcal{P}(s) \cap \mathcal{P}(t), s(\pi) \neq t(\pi)\} \cup \{\omega\} \qquad \text{for } s, t \in \mathcal{T}^\infty(\Sigma)$$

That is, $\mathsf{sim}(s,t)$ is the minimal depth at which s and t differ, respectively ω if $s = t$. Based on this, a distance function $\mathbf{d}$ can be defined by $\mathbf{d}(s,t) = 2^{-\mathsf{sim}(s,t)}$, where we interpret $2^{-\omega}$ as 0. The pair $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$ is known to form a complete ultrametric space [4]. *Partial terms*, i.e. terms over signature $\Sigma_\perp = \Sigma \uplus \{\perp\}$ with

$\bot$ a fresh nullary symbol, can be endowed with a binary relation $\leq_\bot$ by defining $s \leq_\bot t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\bot$. Interpreting the term $\bot$ as denoting "undefined", $\leq_\bot$ can be read as "is less defined than". The pair $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$ is known to form a complete semilattice [14]. When dealing with terms in $\mathcal{T}^\infty(\Sigma_\bot)$, we call terms that do not contain the symbol $\bot$, i.e. terms that are contained in $\mathcal{T}^\infty(\Sigma)$, *total*.

2.5 Term Rewriting Systems

For term rewriting systems, we have to consider terms with variables. To this end, we assume a countably infinite set $\mathcal{V}$ of variable symbols and extend a signature Σ to a signature $\Sigma_\mathcal{V} = \Sigma \uplus \mathcal{V}$ with variable symbols in $\mathcal{V}$ as nullary symbols. Instead of $\mathcal{T}^\infty(\Sigma_\mathcal{V})$ we also write $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}^\infty(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ such that all variables occurring in r also occur in l. Note that both the left- and the right-hand side may be infinite. We usually use x, y, z and primed respectively indexed variants thereof to denote variables in $\mathcal{V}$.

Similar to the setting of finitary term rewriting, every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$ on terms in $\mathcal{T}^\infty(\Sigma)$ as follows:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \text{substitution } \sigma\colon\ s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

Instead of $s \to_\mathcal{R} t$, we sometimes write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π, or simply $s \to t$. The subterm $s|_\pi$ is called a *ρ-redex* or simply *redex*, $r\sigma$ its *contractum*, and $s|_\pi$ is said to be *contracted* to $r\sigma$.

2.6 Convergence of Transfinite Term Reductions

At first, we look at the metric based approach of infinitary term rewriting [13, 20]. The convergence of an infinite reduction is determined by the convergence of the underlying sequence of terms in the metric space $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$.

A *reduction* in a term rewriting system $\mathcal{R}$, is a sequence $S = (t_\iota \to_\mathcal{R} t_{\iota+1})_{\iota<\alpha}$ of rewriting steps in $\mathcal{R}$. The sequence $(t_\iota)_{\iota<\widehat{\alpha}}$ is the underlying sequence of terms, where $\widehat{\alpha} = \alpha$ if α is a limit ordinal, and $\widehat{\alpha} = \alpha + 1$ otherwise. The reduction S is called *weakly m-continuous*, written $S\colon\ t_0 \overset{m}{\to} \ldots$, if the underlying sequence of terms $(t_\iota)_{\iota<\widehat{\alpha}}$, is continuous, i.e. $\lim_{\iota\to\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$. The reduction S is said to *weakly m-converge* to a term t, written $S\colon\ t_0 \overset{m}{\to} t$, if it is weakly m-continuous and the underlying sequence of terms converges to t, i.e. $\lim_{\iota\to\widehat{\alpha}} t_\iota = t$.

Example 2.2. Consider the rewrite rule $\rho\colon\ x :: y :: z \to y :: x :: y :: z$, where $::$ is a binary symbol that we write infix and assume to associate to the right. That is, in its explicitly parenthesised form ρ reads $x :: (y :: z) \to y :: (x :: (y :: z))$. Think of the $::$ symbol as the list constructor *cons*. Using the rule ρ, we have the following reduction S of length ω:

$$S\colon\ a :: a :: c \to a :: a :: \underline{a} :: c \to a :: a :: a :: \underline{a} :: c \to a :: a :: a :: a :: \underline{a} :: c \to \ldots$$

The position at which two consecutive terms differ – indicated by the underlining – moves deeper and deeper into the term structure during the reduction S. Hence, the underlying sequence of terms converges to the infinite term s satisfying the equation $s = a :: s$, i.e. $s = a :: a :: a :: \ldots$. This means that S weakly m-converges to s.

Now consider the starting term $a :: b :: c$. By repeatedly applying ρ at the root we obtain the following reduction:

$$T\colon\ a :: b :: c \to \underline{b} :: a :: b :: c \to \underline{a} :: b :: a :: b :: c \to \underline{b} :: a :: b :: a :: b :: c \to \ldots$$

The difference between consecutive terms remains right at the root position. Hence, the underlying sequence of terms is not Cauchy and, therefore, does not converge. Consequently, T does not weakly m-converge.

However, we can form a weakly m-converging reduction starting from the term $a :: b :: c$ by applying the rule ρ at increasingly deep positions:

$$T'\colon\ a :: b :: c \to \underline{b} :: a :: b :: c \to b :: \underline{b} :: a :: b :: c \to b :: b :: \underline{b} :: a :: b :: c \to \ldots$$

The reduction T' weakly m-converges to the infinite term $t' = b :: b :: b :: \ldots$.

In the partial order approach of infinitary rewriting [6, 7], convergence is defined in terms of the limit inferior in the partially ordered set $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$: a reduction $S = (t_\iota \to_\mathcal{R} t_{\iota+1})_{\iota<\alpha}$ of *partial terms* is called *weakly p-continuous*, written $S\colon\ t_0 \overset{p}{\to} \ldots$, if $\liminf_{\iota<\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$. The reduction S is said to *weakly p-converge* to a term t, written $S\colon\ t_0 \overset{p}{\to} t$, if it is weakly p-continuous and $\liminf_{\iota<\widehat{\alpha}} t_\iota = t$.

The distinguishing feature of the partial order approach is that, due to the complete semilattice structure of $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$, each continuous reduction also converges. Intuitively, weak p-convergence on terms describes an approximation process. To this end, the partial order $\leq_\bot$ captures a notion of *information preservation*: $s \leq_\bot t$ iff t contains at least the same information as s does but potentially more. A monotonic sequence of terms $t_0 \leq_\bot t_1 \leq_\bot \ldots$ thus approximates the information contained in $t = \bigsqcup_{i<\omega} t_i$: any finite part of t is contained is some t_i and subsequently remains stable in $t_{i+1}, t_{i+1}, t_{i+2}, \ldots$. Given this reading of $\leq_\bot$, the glb $\bigsqcap T$ of a set of terms T captures the common (non-contradicting) information of the terms in T. Leveraging this property of the partial order $\leq_\bot$, a sequence of terms $(s_i)_{i<\omega}$ that is not necessarily monotonic can be turned into a monotonic sequence $(t_j)_{j<\omega}$ by setting $t_j = \bigsqcap_{j\leq i<\omega} s_i$. That is, each t_j contains exactly the information that remains stable in $(s_i)_{i<\omega}$ from j onwards. Hence, the limit inferior $\liminf_{i\to\omega} s_i = \bigsqcup_{j<\omega} \bigsqcap_{j\leq i<\omega} s_i$ is the term that contains the accumulated information that eventually remains stable in $(s_i)_{i<\omega}$. This is expressed as an approximation of the monotonically increasing information that remains stable from some point on.

Example 2.3. Reconsider the rule ρ and its induced reduction S from Example 2.2. The reduction S also weakly p-converges to s, i.e. $\liminf_{i\to\omega} s_i$ for $(s_i)_{i<\omega}$ the underlying sequence of terms in S. To see this, consider the sequence $(t_j)_{j<\omega}$ of terms $t_j = \bigsqcap_{j\leq i<\omega} s_i$ each of which intuitively encodes the information that remains stable from j onwards:

$$a :: a :: \bot, \quad a :: a :: a :: \bot, \quad a :: a :: a :: a :: \bot, \quad \ldots$$

This sequence of terms approximates $s = a :: a :: a :: \ldots$ in the sense that $s = \bigsqcup_{j<\omega} t_j$. Likewise, also the reduction T' from Example 2.2 weakly p-converges to the term $t' = b :: b :: b :: \ldots$. The sequence of stable information of T' is

$$\bot :: \bot :: \bot, \quad b :: \bot :: \bot :: \bot, \quad b :: b :: \bot :: \bot :: \bot, \quad \ldots$$

As we have seen, the reduction T from Example 2.2 does not weakly m-converge. However, since T it is trivially weakly p-continuous, it is weakly p-converging. The corresponding sequence of stable information is

$$\bot :: \bot :: \bot, \quad \bot :: \bot :: \bot :: \bot, \quad \bot :: \bot :: \bot :: \bot :: \bot, \quad \ldots$$

This sequence approximates the term $t = \bot :: \bot :: \bot :: \ldots$ and we thus have that T weakly p-converges to t.

The relation between weak m- and p-convergence illustrated in the examples above is characteristic: weak p-convergence is a conservative extension of weak m-convergence. In order to qualify this, we say that a reduction $S = (t_\iota \to t_{\iota+1})_{\iota<\alpha}$ weakly p-converges to t in $\mathcal{T}^\infty(\Sigma)$ if S weakly p-converges to t and t as well as each t_ι with $\iota < \widehat{\alpha}$ is in $\mathcal{T}^\infty(\Sigma)$. Analogously, we say that S is weakly p-continuous in $\mathcal{T}^\infty(\Sigma)$ if S is weakly p-continuous and each t_ι with $\iota < \widehat{\alpha}$ is in $\mathcal{T}^\infty(\Sigma)$. We then have the following correspondence between m- and p-convergence:

Theorem 2.4 (p-convergence in $\mathcal{T}^\infty(\Sigma)$ = m-convergence, Bahr [5])**.** *For every reduction S in a TRS the following equivalences hold:*

(i) $S\colon s \xrightarrow{p} \ldots$ *in* $\mathcal{T}^\infty(\Sigma)$ *iff* $S\colon s \xrightarrow{m} \ldots$.

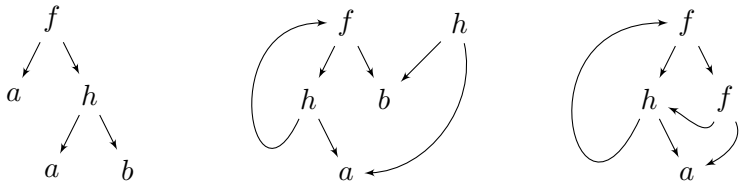
(ii) $S\colon s \xrightarrow{p} t$ *in* $\mathcal{T}^\infty(\Sigma)$ *iff* $S\colon s \xrightarrow{m} t$.

Kennaway [19] and Bahr [6] investigated abstract models of infinitary rewriting based on metric spaces respectively partially ordered sets. We will take these abstract models as a basis to formulate a theory of infinitary term graph reductions. The key question that we have to address is what an appropriate metric space respectively partial order on term graphs looks like.

3 Graphs and Term Graphs

This section provides the basic notions for term graphs and more generally for graphs. Terms over a signature, say Σ, can be thought of as rooted trees whose nodes are labelled with symbols from Σ. Moreover, in these trees a node labelled with a k-ary symbol is restricted to have out-degree k and the outgoing edges are ordered. In this way the i-th successor of a node labelled with a symbol f is interpreted as the root node of the subtree that represents the i-th argument of f. For example, consider the term $f(a, h(a, b))$. The corresponding representation as a tree is shown in Figure 1a.

In term graphs, the restriction to a tree structure is abolished. The corresponding notion of term graphs we are using is taken from Barendregt et al. [10].



(a) $f(a, h(a, b))$. (b) A graph. (c) A term graph.

Figure 1: Tree representation of a term and generalisation to (term) graphs.

Definition 3.1 (graphs)**.** Let Σ be a signature. A *graph* over Σ is a triple $g = (N, \mathsf{lab}, \mathsf{suc})$ consisting of a set N (of *nodes*), a *labelling function* $\mathsf{lab}\colon N \to \Sigma$, and a *successor function* $\mathsf{suc}\colon N \to N^*$ such that $|\mathsf{suc}(n)| = \mathsf{ar}(\mathsf{lab}(n))$ for each node $n \in N$, i.e. a node labelled with a k-ary symbol has precisely k successors. The graph g is called *finite* whenever the underlying set N of nodes is finite. If $\mathsf{suc}(n) = \langle n_0, \ldots, n_{k-1}\rangle$, then we write $\mathsf{suc}_i(n)$ for n_i. Moreover, we use the abbreviation $\mathsf{ar}_g(n)$ for the arity $\mathsf{ar}(\mathsf{lab}(n))$ of n.

Example 3.2. Let $\Sigma = \{f/2, h/2, a/0, b/0\}$ be a signature. The graph over Σ, depicted in Figure 1b, is given by the triple $(N, \mathsf{lab}, \mathsf{suc})$ with $N = \{n_0, n_1, n_2, n_3, n_4\}$, $\mathsf{lab}(n_0) = f, \mathsf{lab}(n_1) = \mathsf{lab}(n_4) = h, \mathsf{lab}(n_2) = b, \mathsf{lab}(n_3) = a$ and $\mathsf{suc}(n_0) = \langle n_1, n_2\rangle, \mathsf{suc}(n_1) = \langle n_0, n_3\rangle, \mathsf{suc}(n_2) = \mathsf{suc}(n_3) = \langle\rangle, \mathsf{suc}(n_4) = \langle n_2, n_3\rangle$.

Definition 3.3 (paths, reachability)**.** Let $g = (N, \mathsf{lab}, \mathsf{suc})$ be a graph and n, m nodes in g.

(i) A *path* in g from n to m is a finite sequence $\pi \in \mathbb{N}^*$ such that either

  (a) π is empty and $n = m$, or

  (b) $\pi = \langle i\rangle \cdot \pi'$ with $0 \leq i < \mathsf{ar}_g(n)$ and the suffix π' a path in g from $\mathsf{suc}_i(n)$ to m.

(ii) If there exists a path in g from n to m, we say that m is *reachable* from n in g.

Definition 3.4 (term graphs)**.** Given a signature Σ, a *term graph* g over Σ is a tuple $(N, \mathsf{lab}, \mathsf{suc}, r)$ consisting of an *underlying* graph $(N, \mathsf{lab}, \mathsf{suc})$ over Σ whose nodes are all reachable from the *root node* $r \in N$. The term graph g is called *finite* if the underlying graph is finite, i.e. the set N of nodes is finite. The class of all term graphs over Σ is denoted $\mathcal{G}^\infty(\Sigma)$; the class of all finite term graphs over Σ is denoted $\mathcal{G}(\Sigma)$. We use the notation N^g, lab^g, suc^g and r^g to refer to the respective components N,lab, suc and r of g. Given a graph or a term graph h and a node n in h, we write $h|_n$ to denote the sub-term graph of h rooted in n.

Example 3.5. Let $\Sigma = \{f/2, h/2, c/0\}$ be a signature. The term graph over Σ, depicted in Figure 1c, is given by the quadruple $(N, \mathsf{lab}, \mathsf{suc}, r)$, where $N = \{r, n_1, n_2, n_3\}$, $\mathsf{suc}(r) = \langle n_1, n_2\rangle$, $\mathsf{suc}(n_1) = \langle r, n_3\rangle$, $\mathsf{suc}(n_2) = \langle n_1, n_3\rangle$, $\mathsf{suc}(n_3) = \langle\rangle$ and $\mathsf{lab}(r) = \mathsf{lab}(n_2) = f$, $\mathsf{lab}(n_1) = h$, $\mathsf{lab}(n_3) = c$.

Paths in a graph are not absolute but relative to a starting node. In term graphs, however, we have a distinguished root node from which each node is reachable. Paths relative to the root node correspond to positions in terms and are central for dealing with term graphs:

Definition 3.6 (positions, depth, cyclicity, trees)**.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $n \in N^g$.

(i) A *position* of n is a path in the underlying graph of g from r^g to n. The set of all positions in g is denoted $\mathcal{P}(g)$; the set of all positions of n in g is denoted $\mathcal{P}_g(n)$.[1]

(ii) The *depth* of n in g, denoted $\mathsf{depth}_g(n)$, is the minimum of the lengths of the positions of n in g, i.e. $\mathsf{depth}_g(n) = \min\{|\pi| \mid \pi \in \mathcal{P}_g(n)\}$.

(iii) For a position $\pi \in \mathcal{P}(g)$, we write $\mathsf{node}_g(\pi)$ for the unique node $n \in N^g$ with $\pi \in \mathcal{P}_g(n)$ and $g(\pi)$ for its symbol $\mathsf{lab}^g(n)$.

(iv) A position $\pi \in \mathcal{P}(g)$ is called *cyclic* if there are paths $\pi_1 < \pi_2 \leq \pi$ with $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. The non-empty path π' with $\pi_1 \cdot \pi' = \pi_2$ is then called a *cycle* of $\mathsf{node}_g(\pi_1)$. A position that is not cyclic is called *acyclic*. If g has a cyclic position, g is called cyclic; otherwise g is called acyclic.

(v) The term graph g is called a *term tree* if each node in g has exactly one position.

Note that the labelling function of graphs – and thus term graphs – is *total*. In contrast, Barendregt et al. [10] considered *open* (term) graphs with a *partial* labelling function such that unlabelled nodes denote holes or variables. This is reflected in their notion of homomorphisms in which the homomorphism condition is suspended for unlabelled nodes.

3.1 Homomorphisms

Instead of a partial node labelling function, we chose a *syntactic* approach that is closer to the representation in terms: variables, holes and "bottoms" are represented as distinguished syntactic entities. We achieve this on term graphs by making the notion of homomorphisms dependent on a distinguished set of constant symbols Δ for which the homomorphism condition is suspended:

Definition 3.7 (Δ-homomorphisms)**.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$, and $g, h \in \mathcal{G}^\infty(\Sigma)$.

(i) A function $\phi\colon N^g \to N^h$ is called *homomorphic* in $n \in N^g$ if the following holds:

$$\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n)) \qquad \text{(labelling)}$$

$$\phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^h_i(\phi(n)) \quad \text{for all } 0 \leq i < \mathsf{ar}_g(n) \qquad \text{(successor)}$$

[1] The notion/notation of positions is borrowed from terms: every position π of a node n corresponds to the subterm represented by n occurring at position π in the unravelling of the term graph to a term.

(ii) A Δ-*homomorphism* ϕ from g to h, denoted $\phi\colon g \to_\Delta h$, is a function $\phi\colon N^g \to N^h$ that is homomorphic in n for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$ and satisfies

$$\phi(r^g) = r^h \qquad \text{(root)}$$

It should be obvious that we get the usual notion of homomorphisms on term graphs if $\Delta = \emptyset$. The Δ-nodes can be thought of as holes in the term graphs which can be filled with other term graphs. For example, if we have a distinguished set of variable symbols $\mathcal{V} \subseteq \Sigma^{(0)}$, we can use $\mathcal{V}$-homomorphisms to formalise the matching of a term graph against a term graph rule, which requires the instantiation of variables.

Proposition 3.8 (Δ-homomorphism preorder)**.** *Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a category which is a preorder. That is, there is at most one Δ-homomorphism from one term graph to another.*

Proof. The identity Δ-homomorphism is obviously the identity mapping on the set of nodes. Moreover, an easy equational reasoning reveals that the composition of two Δ-homomorphisms is again a Δ-homomorphism. Associativity of this composition is obvious as Δ-homomorphisms are functions.

To show that the category is a preorder, assume that there are two Δ-homomorphisms $\phi_1, \phi_2\colon g \to_\Delta h$. We prove that $\phi_1 = \phi_2$ by showing that $\phi_1(n) = \phi_2(n)$ for all $n \in N^g$ by induction on the depth of n in g.

Let $\mathsf{depth}_g(n) = 0$, i.e. $n = r^g$. By the root condition for ϕ, we have that $\phi_1(r^g) = r^h = \phi_2(r^g)$. Let $\mathsf{depth}_g(n) = d > 0$. Then n has a position $\pi \cdot \langle i \rangle$ in g such that $\mathsf{depth}_g(n') < d$ for $n' = \mathsf{node}_g(\pi)$. Hence, we can employ the induction hypothesis for n' to obtain the following:

$$\begin{aligned}
\phi_1(n) &= \mathsf{suc}_i^h(\phi_1(n')) && \text{(successor condition for } \phi_1\text{)}\\
&= \mathsf{suc}_i^h(\phi_2(n')) && \text{(induction hypothesis)}\\
&= \phi_2(n) && \text{(successor condition for } \phi_2\text{)}
\end{aligned}$$

□

As a consequence, each Δ-homomorphism is both monic and epic, and whenever there are two Δ-homomorphisms $\phi\colon g \to_\Delta h$ and $\psi\colon h \to_\Delta g$, they are inverses of each other, i.e. Δ-*isomorphisms*. If two term graphs are Δ-*isomorphic*, we write $g \cong_\Delta h$.

For the two special cases $\Delta = \emptyset$ and $\Delta = \{\sigma\}$, we write $\phi\colon g \to h$ respectively $\phi\colon g \to_\sigma h$ instead of $\phi\colon g \to_\Delta h$ and call ϕ a homomorphism respectively a σ-homomorphism. The same convention applies to Δ-isomorphisms.

Lemma 3.9 (homomorphisms are surjective)**.** *Every homomorphism $\phi\colon g \to h$, with $g, h \in \mathcal{G}^\infty(\Sigma)$, is surjective.*

Proof. Follows from an easy induction on the depth of the nodes in h. □

Note that injectivity of Δ-homomorphisms is in general different from both being monic and the existence of left-inverses. The same holds for surjectivity and being epic respectively having right-inverses. Likewise, a bijective Δ-homomorphism is not necessarily a Δ-isomorphism. To realise this, consider two term graphs g, h, each with one node only. Let the node in g be labelled with a and the node in h with b then the only possible a-homomorphism from g to h is clearly a bijection but not an a-isomorphism. On the other hand, bijective homomorphisms are isomorphisms.

Lemma 3.10 (bijective homomorphisms are isomorphisms)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\phi\colon\ g \to h$. Then the following are equivalent*

(a) ϕ is an isomorphism.

(b) ϕ is bijective.

(c) ϕ is injective.

Proof. The implication (a) $\Rightarrow$ (b) is trivial. The equivalence (b) $\Leftrightarrow$ (c) follows from Lemma 3.9. For the implication (b) $\Rightarrow$ (a), consider the inverse ϕ^{-1} of ϕ. We need to show that ϕ^{-1} is a homomorphism from h to g. The root condition follows immediately from the root condition for ϕ. Similarly, an easy equational reasoning reveals that the fact that ϕ is homomorphic in N^g implies that ϕ^{-1} is homomorphic in N^h □

3.2 Canonical Term Graphs

In this section, we introduce a canonical representation of isomorphism classes of term graphs. We use a well-known trick to achieve this [26]. As we shall see at the end of this section, this will also enable us to construct term graphs modulo isomorphism very easily.

Definition 3.11 (canonical term graphs)**.** A term graph g is called *canonical* if $n = \mathcal{P}_g(n)$ holds for each $n \in N^g$. That is, each node is the set of its positions in the term graph. The set of all (finite) canonical term graphs over Σ is denoted $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ (respectively $\mathcal{G}_{\mathcal{C}}(\Sigma)$).

By associating nodes with their respective set of positions we obtain a convenient characterisation of Δ-homomorphisms:

Lemma 3.12 (characterisation of Δ-homomorphisms)**.** *For $g, h \in \mathcal{G}^\infty(\Sigma)$, a function $\phi\colon\ N^g \to N^h$ is a Δ-homomorphism $\phi\colon\ g \to_\Delta h$ iff the following holds for all $n \in N^g$:*

(a) $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(\phi(n))$, and

(b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ whenever $\mathsf{lab}^g(n) \notin \Delta$.

Proof. For the "only if" direction, assume that $\phi\colon\ g \to_\Delta h$. (b) is the labelling condition and is therefore satisfied by ϕ. To establish (a), we show the equivalent statement

$$\forall \pi \in \mathcal{P}(g).\ \forall n \in N^g.\ \pi \in \mathcal{P}_g(n) \implies \pi \in \mathcal{P}_h(\phi(n))$$

We do so by induction on the length of π. If $\pi = \langle\rangle$, then $\pi \in \mathcal{P}_g(n)$ implies $n = r^g$. By the root condition, we have $\phi(r^g) = r^h$ and, therefore, $\pi = \langle\rangle \in \phi(r^g)$. If $\pi = \pi' \cdot \langle i \rangle$, then let $n' = \mathsf{node}_g(\pi')$. Consequently, $\pi' \in \mathcal{P}_g(n')$ and, by induction hypothesis, $\pi' \in \mathcal{P}_h(\phi(n'))$. Since $\pi = \pi' \cdot \langle i \rangle$, we have $\mathsf{suc}_i^g(n') = n$. By the successor condition we can conclude $\phi(n) = \mathsf{suc}_i^h(\phi(n'))$. This and $\pi' \in \mathcal{P}_h(\phi(n'))$ yields that $\pi' \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n))$.

For the "if" direction, we assume (a) and (b). The labelling condition follows immediately from (b). For the root condition, observe that since $\langle\rangle \in \mathcal{P}_g(r^g)$, we also have $\langle\rangle \in \mathcal{P}_h(\phi(r^g))$. Hence, $\phi(r^g) = r^h$. In order to show the successor condition, let $n, n' \in N^g$ and $0 \leq i < \mathsf{ar}_g(n)$ such that $\mathsf{suc}_i^g(n) = n'$. Then there is a position $\pi \in \mathcal{P}_g(n)$ with $\pi \cdot \langle i \rangle \in \mathcal{P}_g(n')$. By (a), we can conclude that $\pi \in \mathcal{P}_h(\phi(n))$ and $\pi \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n'))$ which implies that $\mathsf{suc}_i^h(\phi(n)) = \phi(n')$. □

By Proposition 3.8, there is at most one Δ-homomorphism between two term graphs. The lemma above uniquely defines this Δ-homomorphism: if there is a Δ-homomorphism from g to h, it is defined by $\phi(n) = n'$, where n' is the unique node $n' \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(n')$.

By associating with each position π in a term graph g the node $\mathsf{node}_g(\pi)$, we obtain an equivalence relation $\sim_g$ on the set $\mathcal{P}(g)$ of positions in g as follows: $\pi_1 \sim_g \pi_2$ iff $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. Using this equivalence relation, the above characterisation of Δ-homomorphisms can be recast to obtain the following lemma that characterises the *existence* of Δ-homomorphisms:

Lemma 3.13 (characterisation of Δ-homomorphisms)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, there is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff, for all $\pi, \pi' \in \mathcal{P}(g)$, we have*

$$\text{(a) } \pi \sim_g \pi' \quad \Longrightarrow \quad \pi \sim_h \pi', \text{ and} \quad \text{(b) } g(\pi) = h(\pi) \quad \text{whenever} \quad g(\pi) \notin \Delta.$$

Proof. For the "only if" direction, assume that ϕ is a Δ-homomorphism from g to h. Then we can use the properties (a) and (b) of Lemma 3.12, which we will refer to as (a') and (b') to avoid confusion. In order to show (a), assume $\pi \sim_g \pi'$. Then there is some node $n \in N^g$ with $\pi, \pi' \in \mathcal{P}_g(n)$. (a') yields $\pi, \pi' \in \phi(n)$ and, therefore, $\pi \sim_g \pi'$. To show (b), we assume some $\pi \in \mathcal{P}(g)$ with $g(\pi) \notin \Delta$. Then we can reason as follows:

$$g(\pi) = \mathsf{lab}^g(\mathsf{node}_g(\pi)) \overset{\text{(b')}}{=} \mathsf{lab}^h(\phi(\mathsf{node}_g(\pi))) \overset{\text{(a')}}{=} \mathsf{lab}^h(\mathsf{node}_h(\pi)) = h(\pi)$$

For the converse direction, assume that both (a) and (b) hold. Define the function $\phi\colon N^g \to N^h$ by $\phi(n) = m$ iff $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$ for all $n \in N^g$ and $m \in N^h$. To see that this is well-defined, we show at first that, for each $n \in N^g$, there is at most one $m \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$. Suppose there is another node $m' \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m')$. Since $\mathcal{P}_g(n) \neq \emptyset$, this implies $\mathcal{P}_h(m) \cap \mathcal{P}_h(m') \neq \emptyset$. Hence, $m = m'$. Secondly, we show that there is at least one such node m. Choose some $\pi^* \in \mathcal{P}_g(n)$. Since then $\pi^* \sim_g \pi^*$ and, by (a), also $\pi^* \sim_h \pi^*$ holds, there is some $m \in N^h$ with $\pi^* \in \mathcal{P}_h(m)$. For each $\pi \in \mathcal{P}_g(n)$, we have $\pi^* \sim_g \pi$ and, therefore, $\pi^* \sim_h \pi$ by (a). Hence, $\pi \in \mathcal{P}_h(m)$. So we know that ϕ is well-defined. By construction, ϕ satisfies (a'). Moreover, because of (b), it is also easily seen to satisfy (b'). Hence, ϕ is a homomorphism from g to h. □

Intuitively, (a) means that h has at least as much sharing of nodes as g has, whereas (b) means that h has at least the same non-Δ-symbols as g.

From the two characterisations of Δ-homomorphisms that we have developed above, we can easily derive the following characterisation of Δ-isomorphisms using the uniqueness of Δ-homomorphisms between two term graphs:

Corollary 3.14 (characterisation of Δ-isomorphisms)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, the following holds:*

(i) $\phi\colon N^g \to N^h$ is a Δ-isomorphism iff for all $n \in N^g$

(a) $\mathcal{P}_h(\phi(n)) = \mathcal{P}_g(n)$, and
(b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ or $\mathsf{lab}^g(n), \mathsf{lab}^h(\phi(n)) \in \Delta$.

(ii) $g \cong_\Delta h$ iff (a) $\sim_g = \sim_h$, and (b) $g(\pi) = h(\pi)$ or $g(\pi), h(\pi) \in \Delta$.

Proof. This follows from Lemma 3.12 and Lemma 3.13 using Proposition 3.8. □

From clause (ii) we immediately obtain the following equivalence between isomorphisms and σ-isomorphisms:

Corollary 3.15 (σ-isomorphism = isomorphism)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\sigma \in \Sigma^{(0)}$, we have $g \cong h$ iff $g \cong_\sigma h$.*

Now we can revisit the notion of canonical term graphs using the above characterisation of Δ-isomorphisms. We will define a function $\mathcal{C}(\cdot)\colon \mathcal{G}^\infty(\Sigma) \to \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ that maps a term graph to its canonical representation. To this end, let $g = (N, \mathsf{lab}, \mathsf{suc}, r)$ be a term graph and define $\mathcal{C}(g) = (N', \mathsf{lab}', \mathsf{suc}', r')$ as follows:

$$N' = \{\mathcal{P}_g(n) \mid n \in N\} \quad r' = \mathcal{P}_g(r)$$
$$\mathsf{lab}'(\mathcal{P}_g(n)) = \mathsf{lab}(n) \qquad \mathsf{suc}'_i(\mathcal{P}_g(n)) = \mathcal{P}_g(\mathsf{suc}_i(n)) \quad \text{for all } n \in N, 0 \leq i < \mathsf{ar}_g(n)$$

$\mathcal{C}(g)$ is obviously a well-defined canonical term graph. With this definition we indeed obtain canonical representatives isomorphism classes:

Proposition 3.16 (canonical term graphs are a canonical representation)**.** *Given $g \in \mathcal{G}^\infty(\Sigma)$, the term graph $\mathcal{C}(g)$ canonically represents the equivalence class $[g]_\cong$. More precisely, it holds that*

$$\textit{(i) } [g]_\cong = [\mathcal{C}(g)]_\cong, \textit{ and} \qquad \textit{(ii) } [g]_\cong = [h]_\cong \quad \textit{iff} \quad \mathcal{C}(g) = \mathcal{C}(h).$$

In particular, we have, for all canonical term graphs g, h, that $g = h$ iff $g \cong h$.

Proof. Straightforward consequence of Corollary 3.14. □

Corollary 3.14 has shown that term graphs can be characterised up to isomorphism by only giving the equivalence $\sim_g$ and the labelling $g(\cdot)\colon \pi \mapsto g(\pi)$. This observation gives rise to the following definition:

Definition 3.17 (labelled quotient trees)**.** A *labelled quotient tree* over signature Σ is a triple $(P, l, \sim)$ consisting of a non-empty set $P \subseteq \mathbb{N}^*$, a function $l\colon P \to \Sigma$, and an equivalence relation $\sim$ on P that satisfies the following conditions for all $\pi, \pi' \in \mathbb{N}^*$ and $i \in \mathbb{N}$:

$$\pi \cdot \langle i \rangle \in P \implies \pi \in P \quad \text{and} \quad i < \mathsf{ar}(l(\pi)) \qquad \text{(reachability)}$$

$$\pi \sim \pi' \implies \begin{cases} l(\pi) = l(\pi') & \text{and} \\ \pi \cdot \langle i \rangle \sim \pi' \cdot \langle i \rangle & \text{for all } \ i < \mathsf{ar}(l(\pi)) \end{cases} \qquad \text{(congruence)}$$

In other words, a labelled quotient tree $(P, l, \sim)$ is a ranked tree domain P together with a congruence $\sim$ on it and a labelling function $l\colon P/_\sim \to \Sigma$ that honours the rank.

The following lemma confirms that labelled quotient trees uniquely characterise any term graph up to isomorphism:

Lemma 3.18 (labelled quotient trees are canonical)**.** *Each term graph $g \in \mathcal{G}^\infty(\Sigma)$ induces a labelled quotient tree $(\mathcal{P}(g), g(\cdot), \sim_g)$ over Σ. Vice versa, for each labelled quotient tree $(P, l, \sim)$ over Σ there is a unique canonical term graph $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ whose labelled quotient tree is $(P, l, \sim)$, i.e. $\mathcal{P}(g) = P$, $g(\pi) = l(\pi)$ for all $\pi \in P$, and $\sim_g \,= \sim$.*

Proof. The first part is trivial: $(\mathcal{P}(g), g(\cdot), \sim_g)$ satisfies the conditions from Definition 3.17.

For the second part, let $(P, l, \sim)$ be a labelled quotient tree. Define the term graph $g = (N, \mathsf{lab}, \mathsf{suc}, r)$ by

$$\begin{aligned} N &= P/_\sim & \mathsf{lab}(n) &= f \quad \text{iff} \quad \exists \pi \in n.\ l(\pi) = f \\ r &= [\langle\rangle]_\sim & \mathsf{suc}_i(n) &= n' \quad \text{iff} \quad \exists \pi \in n.\ \pi \cdot \langle i \rangle \in n' \end{aligned}$$

The functions lab and suc are well-defined due to the congruence condition satisfied by $(P, l, \sim)$. Since P is non-empty and closed under prefixes, it contains $\langle\rangle$. Hence, r is well-defined. Moreover, by the reachability condition, each node in N is reachable from the root node. An easy induction proof shows that $\mathcal{P}_g(n) = n$ for each node $n \in N$. Thus, g is a well-defined canonical term graph. The labelled quotient tree of g is obviously $(P, l, \sim)$. Whenever there are two canonical term graphs with the same labelled quotient tree $(P, l, \sim)$, they are isomorphic due to Corollary 3.14 and, therefore, have to be identical by Proposition 3.16. □

Example 3.19. The term graph g_1 depicted in Figure 2 on page 390 is given by the labelled quotient tree $(P, l, \sim)$ with $P = \{\langle\rangle, \langle 0\rangle, \langle 1\rangle\}$, $l(\langle\rangle) = f$, $l(\langle 0\rangle) = l(\langle 1\rangle) = c$ and $\sim$ the least equivalence relation on P with $\langle 0\rangle \sim \langle 1\rangle$.

Labelled quotient trees provide a valuable tool for constructing canonical term graphs. Nevertheless, the original graph representation remains convenient for practical purposes as it allows a straightforward formalisation of term graph rewriting and provides a finite representation of finite cyclic term graphs, which induce an infinite labelled quotient tree.

Before we continue, it is instructive to make the correspondence between terms and term graphs clear. First note that, for each term tree t, the equivalence $\sim_t$

is the identity relation $\mathcal{I}_{\mathcal{P}(t)}$ on $\mathcal{P}(t)$, i.e. $\pi_1 \sim_t \pi_2$ iff $\pi_1 = \pi_2$. Consequently, we have the following one-to-one correspondence between canonical term *trees* and terms: each term $t \in \mathcal{T}^\infty(\Sigma)$ induces the canonical term tree given by the labelled quotient tree $(\mathcal{P}(t), t(\cdot), \mathcal{I}_{\mathcal{P}(t)})$. For example, the term tree depicted in Figure 1a corresponds to the term $f(a, h(a, b))$. We thus consider the set of terms $\mathcal{T}^\infty(\Sigma)$ as the subset of canonical term trees of $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$.

With this correspondence in mind, we define the *unravelling* of a term graph g, denoted $\mathcal{U}(g)$, as the unique term t such that there is a homomorphism $\phi\colon\ t \to g$.

For example, the term $f(c, c)$ is the unravelling of the term graph g_1 in Figure 2 and the infinite term $b :: b :: b :: \ldots$ representing an infinite list of 'b's is the unravelling of both the term graphs h and h' in Figure 5b.

4 A Simple Partial Order on Term Graphs

In this section, we want to establish a partial order suitable for formalising convergence of sequences of canonical term graphs similarly to weak p-convergence on terms.

Recall that weak p-convergence on term rewriting systems is based on a partial order $\leq_\bot$ on the set $\mathcal{T}^\infty(\Sigma_\bot)$ of *partial* terms. The partial order $\leq_\bot$ instantiates occurrences of $\bot$ from left to right, i.e. $s \leq_\bot t$ iff t is obtained by replacing occurrences of $\bot$ in s by arbitrary terms in $\mathcal{T}^\infty(\Sigma_\bot)$.

Analogously, we will consider the class of *partial term graphs* simply as term graphs over the signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$. In order to generalise the partial order $\leq_\bot$ to term graphs, we need to formalise the instantiation of occurrences of $\bot$ in term graphs. To this end, we will look more closely at Δ-homomorphisms with $\Delta = \{\bot\}$, or $\bot$-homomorphisms for short. A $\bot$-homomorphism $\phi\colon g \to_\bot h$ maps each node in g to a node in h while "preserving its structure". Except for nodes labelled $\bot$ this also includes preserving the labelling. This exception to the homomorphism condition allows the $\bot$-homomorphism ϕ to instantiate each $\bot$-node in g with an arbitrary node in h.

Therefore, we shall use $\bot$-homomorphisms as the basis for generalising $\leq_\bot$ to canonical partial term graphs. This approach is based on the observation that $\bot$-homomorphisms characterise the partial order $\leq_\bot$ on terms. Considering terms as canonical term trees, we obtain the following characterisation of $\leq_\bot$ on terms $s, t \in \mathcal{T}^\infty(\Sigma_\bot)$:

$$s \leq_\bot t \iff \text{there is a } \bot\text{-homomorphism } \phi\colon\ s \to_\bot t.$$

Embodying a natural concept on term graphs, $\bot$-homomorphisms thus constitute the ideal tool to define a partial order on canonical partial term graphs that generalises $\leq_\bot$.

In this paper, we focus on the simplest among these partial orders on term graphs:

Definition 4.1 (simple partial order $\leq^{\mathsf{S}}_\bot$)**.** The relation $\leq^{\mathsf{S}}_\bot$ on $\mathcal{G}^\infty(\Sigma_\bot)$ is defined as follows: $g \leq^{\mathsf{S}}_\bot h$ iff there is a $\bot$-homomorphism $\phi\colon\ g \to_\bot h$.

One of our objective is to argue that the simple partial order $\leq^{\mathsf{S}}_\bot$ is indeed a suitable structure for deriving a notion of convergence on term graphs in general and for infinitary term graph rewriting in particular.

Due to the preorder structure of $\bot$-homomorphisms on term graphs and the characterisation of isomorphisms as given by Corollary 3.15, the relation $\leq_\bot^{\mathsf{S}}$ forms a partial order if restricted to canonical term graphs.

Proposition 4.2 (simple partial order $\leq_\bot^{\mathsf{S}}$)**.** *The relation $\leq_\bot^{\mathsf{S}}$ is a partial order on $\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$.*

Proof. Transitivity and reflexivity of $\leq_\bot^{\mathsf{S}}$ follows from Proposition 3.8. For antisymmetry, consider $g, h \in \mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$ with $g \leq_\bot^{\mathsf{S}} h$ and $h \leq_\bot^{\mathsf{S}} g$. Then, by Proposition 3.8, $g \cong_\bot h$. This is equivalent to $g \cong h$ by Corollary 3.15 from which we can conclude $g = h$ using Proposition 3.16. □

Before we study the properties of the partial order $\leq_\bot^{\mathsf{S}}$, it is helpful to make its characterisation in terms of labelled quotient trees explicit:

Corollary 4.3 (characterisation of $\leq_\bot^{\mathsf{S}}$)**.** *Let $g, h \in \mathcal{G}^{\infty}(\Sigma_\bot)$. Then $g \leq_\bot^{\mathsf{S}} h$ iff the following conditions are met:*

(a) $\pi \sim_g \pi' \implies \pi \sim_h \pi'$ *for all* $\pi, \pi' \in \mathcal{P}(g)$

(b) $g(\pi) = h(\pi)$ *for all* $\pi \in \mathcal{P}(g)$ *with* $g(\pi) \in \Sigma$.

Proof. This follows immediately from Lemma 3.13. □

Note that the partial order $\leq_\bot$ on terms is entirely characterised by (b). In other words, the partial order $\leq_\bot^{\mathsf{S}}$ is a combination of the partial order $\leq_\bot$ imposed on the underlying tree structure of term graphs (i.e. their unravelling) and the preservation of sharing as stipulated by (a).

In order to reflect on the merit of the partial order $\leq_\bot^{\mathsf{S}}$ as a suitable basis for a notion of convergence on term graphs, recall the characteristics of the partial order-based notion of convergence for terms: weak p-convergence on terms is based on the ability of the partial order $\leq_\bot$ to capture *information preservation* between terms – $s \leq_\bot t$ means that t contains at least the same information as s does. The limit inferior – and thus weak p-convergence – comprises the accumulated information that eventually remains stable along a sequence. Following the approach on terms, a partial order suitable as a basis for convergence for term graph rewriting, has to capture an appropriate notion of information preservation as well.

One has to keep in mind, however, that term graphs encode an additional dimension of information through *sharing* of nodes, i.e. nodes with multiple positions. Since $\leq_\bot^{\mathsf{S}}$ specialises to $\leq_\bot$ on terms, it does preserve the information on the tree structure in the same way as $\leq_\bot$ does. The difficult part is to determine the right approach to the role of sharing.

Indeed, $\bot$-homomorphisms instantiate occurrences of $\bot$ and are thereby able to introduce new information. But they also introduce sharing by mapping different nodes to the same target node: for the term graphs g_0 and g_1 in Figure 2, we have an obvious $\bot$-homomorphism – in fact a homomorphism – $\phi\colon\ g_0 \to_\bot g_1$ and thus $g_0 \leq_\bot^{\mathsf{S}} g_1$. However, this homomorphism ϕ maps both c-nodes in g_0 to the single c-node in g_1.

There are at least two different ways to interpret the differences in g_0 and g_1. The first one dismisses $\leq_\bot^{\mathsf{S}}$ as a partial order suitable for our purposes: the term

graphs g_0 and g_1 contain contradicting information. While in g_0 the two children of the f-node are distinct, they are identical in g_1. We adopted this view in our previous work on convergence for term graphs [8], where we studied a more rigid partial order $\leq_\bot^{\mathsf{R}}$ for which g_0 and g_1 are indeed incomparable. The second view, which we will adopt in this paper, does not see g_0 and g_1 in contradiction. Both show the f-nodes with two successors, both of which are labelled with c. The term graph g_1 merely contains the additional piece of information that the two successor nodes of the f-node are identical. Hence, $g_0 \leq_\bot^{\mathsf{S}} g_1$.

The rest of this section is concerned with showing that the partial order $\leq_\bot^{\mathsf{S}}$ has indeed the properties that make it a suitable basis for weak p-convergence, i.e. that it forms a complete semilattice. At first we show its cpo structure:

Theorem 4.4. *The partially ordered set $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{S}})$ is a cpo. In particular, it has the least element $\bot$, and the least upper bound of a directed set G is given by the following labelled quotient tree $(P, l, \sim)$:*

$$P = \bigcup_{g \in G} \mathcal{P}(g) \qquad \sim = \bigcup_{g \in G} \sim_g \qquad l(\pi) = \begin{cases} f & \text{if } f \in \Sigma \text{ and } \exists g \in G.\ g(\pi) = f \\ \bot & \text{otherwise} \end{cases}$$

Proof. The least element of $\leq_\bot^{\mathsf{S}}$ is obviously $\bot$. Hence, it remains to be shown that each each directed subset G of $\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$ has a least upper bound $\overline{g}$ given by the labelled quotient tree $(P, l, \sim)$ defined above. To this end, we will make extensive use of Corollary 4.3 using (a) and (b) to refer to its corresponding conditions.

At first we need to show that l is indeed well-defined. For this purpose, let $g_1, g_2 \in G$ and $\pi \in \mathcal{P}(g_1) \cap \mathcal{P}(g_2)$ with $g_1(\pi), g_2(\pi) \in \Sigma$. Since G is directed, there is some $g \in G$ such that $g_1, g_2 \leq_\bot^{\mathsf{S}} g$. By (b), we can conclude $g_1(\pi) = g(\pi) = g_2(\pi)$.

Next we show that $(P, l, \sim)$ is indeed a labelled quotient tree. Recall that $\sim$ needs to be an equivalence relation. For the reflexivity, assume that $\pi \in P$. Then there is some $g \in G$ with $\pi \in \mathcal{P}(g)$. Since $\sim_g$ is an equivalence relation, $\pi \sim_g \pi$ must hold and, therefore, $\pi \sim \pi$. For the symmetry, assume that $\pi_1 \sim \pi_2$. Then there is some $g \in G$ such that $\pi_1 \sim_g \pi_2$. Hence, we get $\pi_2 \sim_g \pi_1$ and, consequently, $\pi_2 \sim \pi_1$. In order to show transitivity, assume that $\pi_1 \sim \pi_2, \pi_2 \sim \pi_3$. That is, there are $g_1, g_2 \in G$ with $\pi_1 \sim_{g_1} \pi_2$ and $\pi_2 \sim_{g_2} \pi_3$. Since G is directed, we find some $g \in G$ such that $g_1, g_2 \leq_\bot^{\mathsf{S}} g$. By (a), this implies that also $\pi_1 \sim_g \pi_2$ and $\pi_2 \sim_g \pi_3$. Hence, $\pi_1 \sim_g \pi_3$ and, therefore, $\pi_1 \sim \pi_3$.

For the reachability condition, let $\pi \cdot \langle i \rangle \in P$. That is, there is a $g \in G$ with $\pi \cdot \langle i \rangle \in \mathcal{P}(g)$. Hence, $\pi \in \mathcal{P}(g)$, which in turn implies $\pi \in P$. Moreover, $\pi \cdot \langle i \rangle \in \mathcal{P}(g)$ implies that $i < \mathsf{ar}(g(\pi))$. Since $g(\pi)$ cannot be a nullary symbol and in particular not $\bot$, we obtain that $l(\pi) = g(\pi)$. Hence, $i < \mathsf{ar}(l(\pi))$.

For the congruence condition, assume that $\pi_1 \sim \pi_2$ and that $l(\pi_1) = f$. If $f \in \Sigma$, then there are $g_1, g_2 \in G$ with $\pi_1 \sim_{g_1} \pi_2$ and $g_2(\pi_1) = f$. Since G is directed, there is some $g \in G$ such that $g_1, g_2 \leq_\bot^{\mathsf{S}} g$. Hence, by (a) respectively (b), we have $\pi_1 \sim_g \pi_2$ and $g(\pi_1) = f$. Using Lemma 3.18 we can conclude that $g(\pi_2) = g(\pi_1) = f$ and that $\pi_1 \cdot \langle i \rangle \sim_g \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(g(\pi_1))$. Because $g \in G$, it holds that $l(\pi_2) = f$ and that $\pi_1 \cdot \langle i \rangle \sim \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(l(\pi_1))$. If $f = \bot$, then also $l(\pi_2) = \bot$, for if $l(\pi_2) = f'$ for some $f' \in \Sigma$, then, by the symmetry of

$\sim$ and the above argument (for the case $f \in \Sigma$), we would obtain $f = f'$ and, therefore, a contradiction. Since $\bot$ is a nullary symbol, the remainder of the condition is vacuously satisfied.

This shows that $(P, l, \sim)$ is a labelled quotient tree which, by Lemma 3.18, uniquely defines a canonical term graph. In order to show that the thus obtained term graph $\overline{g}$ is an upper bound for G, we have to show that $g \leq^{\mathsf{S}}_{\bot} \overline{g}$ for all $g \in G$ by establishing (a) and (b). This is an immediate consequence of the construction of $\overline{g}$.

In the final part of this proof, we will show that $\overline{g}$ is the least upper bound of G. For this purpose, let $\hat{g}$ be an upper bound of G, i.e. $g \leq^{\mathsf{S}}_{\bot} \hat{g}$ for all $g \in G$. We will show that $\overline{g} \leq^{\mathsf{S}}_{\bot} \hat{g}$ by establishing (a) and (b). For (a), assume that $\pi_1 \sim \pi_2$. Hence, there is some $g \in G$ with $\pi_1 \sim_g \pi_2$. Since, by assumption, $g \leq^{\mathsf{S}}_{\bot} \widehat{g}$, we can conclude $\pi_1 \sim_{\widehat{g}} \pi_2$ using (a). For (b), assume $\pi \in P$ and $l(\pi) = f \in \Sigma$. Then there is some $g \in G$ with $g(\pi) = f$. Applying (b) then yields $\widehat{g}(\pi) = f$ since $g \leq^{\mathsf{S}}_{\bot} \widehat{g}$. □

The following proposition shows that the partial order $\leq^{\mathsf{S}}_{\bot}$ also admits glbs of arbitrary non-empty sets:

Proposition 4.5. *In the partially ordered set $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{S}}_{\bot})$ every non-empty set has a glb. In particular, the glb of a non-empty set G is given by the following labelled quotient tree $(P, l, \sim)$:*

$$P = \left\{ \pi \in \bigcap_{g \in G} \mathcal{P}(g) \;\middle|\; \forall \pi' < \pi \exists f \in \Sigma_\bot \forall g \in G : g(\pi') = f \right\}$$

$$l(\pi) = \begin{cases} f & \text{if } \forall g \in G : f = g(\pi) \\ \bot & \text{otherwise} \end{cases} \qquad \sim \; = \bigcap_{g \in G} \sim_g \cap\, P \times P$$

Proof. At first we need to prove that $(P, l, \sim)$ is in fact a well-defined labelled quotient tree. That $\sim$ is an equivalence relation follows straightforwardly from the fact that each $\sim_g$ is an equivalence relation.

Next, we show the reachability and congruence properties of Definition 3.17. In order to show the reachability property, assume some $\pi \cdot \langle i \rangle \in P$. Then, for each $\pi' \leq \pi$ there is some $f_{\pi'} \in \Sigma_\bot$ such that $g(\pi') = f_{\pi'}$ for all $g \in G$. Hence, $\pi \in P$. Moreover, we have in particular that $i < \mathsf{ar}(f_\pi) = \mathsf{ar}(l(\pi))$.

For the congruence condition, assume that $\pi_1 \sim \pi_2$. Hence, $\pi_1 \sim_g \pi_2$ for all $g \in G$. Consequently, we have for each $g \in G$ that $g(\pi_1) = g(\pi_2)$ and that $\pi_1 \cdot \langle i \rangle \sim_g \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(g(\pi_1))$. We distinguish two cases: at first assume that there are some $g_1, g_2 \in G$ with $g_1(\pi_1) \neq g_2(\pi_1)$. Hence, $l(\pi_2) = \bot$. Since we also have that $g_1(\pi_2) = g_1(\pi_1) \neq g_2(\pi_1) = g_2(\pi_2)$, we can conclude that $l(\pi_2) = \bot = l(\pi_1)$. Since $\mathsf{ar}(\bot) = 0$, we are done for this case. Next, consider the alternative case that there is some $f \in \Sigma_\bot$ such that $g(\pi_1) = f$ for all $g \in G$. Consequently, $l(\pi_1) = f$ and since also $g(\pi_2) = g(\pi_1) = f$ for all $g \in G$, we can conclude that $l(\pi_2) = f = l(\pi_1)$. Moreover, we obtain from the initial assumption for this case, that $\pi_1 \cdot \langle i \rangle, \pi_2 \cdot \langle i \rangle \in P$ for all $i < \mathsf{ar}(f)$ which implies that $\pi_1 \cdot \langle i \rangle \sim \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(f) = \mathsf{ar}(l(\pi_1))$.

Next, we show that the term graph $\overline{g}$ defined by $(P, l, \sim)$ is a lower bound of G, i.e. that $\overline{g} \leq^{\mathsf{S}}_{\bot} g$ for all $g \in G$. By Corollary 4.3, it suffices to show $\sim \cap\, P \times P \subseteq \sim_g$

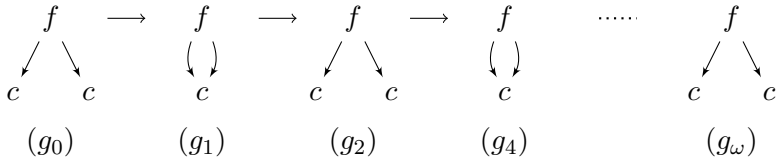


Figure 2: Limit inferior in the presence of acyclic sharing.

and $l(\pi) = g(\pi)$ for all $\pi \in P$ with $l(\pi) \in \Sigma$. Both conditions follow immediately from the construction of $\overline{g}$.

Finally, we show that $\overline{g}$ is the greatest lower bound of G. To this end, let $\widehat{g} \in \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\perp})$ with $\widehat{g} \leq^{\mathsf{S}}_{\perp} g$ for each $g \in G$. We will show that then $\widehat{g} \leq^{\mathsf{S}}_{\perp} \overline{g}$ using Corollary 4.3. At first we show that $\mathcal{P}(\widehat{g}) \subseteq P$. Let $\pi \in \mathcal{P}(\widehat{g})$. We know that $\widehat{g}(\pi') \in \Sigma$ for all $\pi' < \pi$. According to Corollary 4.3, using the assumption that $\widehat{g} \leq^{\mathsf{S}}_{\perp} g$ for all $g \in G$, we obtain that $g(\pi') = \widehat{g}(\pi')$ for all $\pi' < \pi$. Consequently, $\pi \in P$. Next, we show part (a) of Corollary 4.3. Let $\pi_1, \pi_2 \in \mathcal{P}(\widehat{g}) \subseteq P$ with $\pi_1 \sim_{\widehat{g}} \pi_2$. Hence, using the assumption that $\widehat{g}$ is a lower bound of G, we have $\pi_1 \sim_g \pi_2$ for all $g \in G$ according to Corollary 4.3. Consequently, $\pi_1 \sim \pi_2$. For part (b) of Corollary 4.3 let $\pi \in \mathcal{P}(\widehat{g}) \subseteq P$ with $\widehat{g}(\pi) = f \in \Sigma$. Using Corollary 4.3, we obtain that $g(\pi) = f$ for all $g \in G$. Hence, $l(\pi) = f$. □

From this we can immediately derive the complete semilattice structure of $\leq^{\mathsf{S}}_{\perp}$:

Theorem 4.6. *The partially ordered set $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\perp}), \leq^{\mathsf{S}}_{\perp})$ forms a complete semilattice.*

Proof. Follows from Theorem 4.4 and Proposition 4.5. □

In particular, this means that the limit inferior is defined for every sequence of term graphs. Moreover, from the constructions given in Theorem 4.4 and Proposition 4.5, we can derive the following direct construction of the limit inferior:

Corollary 4.7. *The limit inferior of a sequence $(g_\iota)_{\iota<\alpha}$ in $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\perp}), \leq^{\mathsf{S}}_{\perp})$ is given by the following labelled quotient tree $(P, \sim, l)$:*

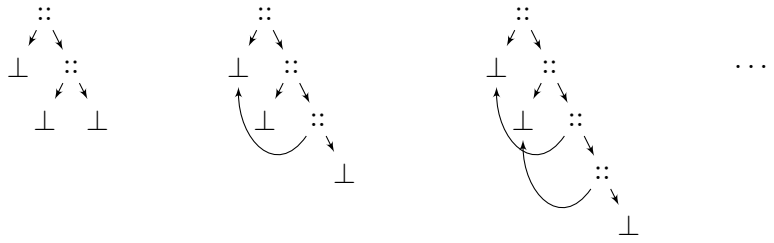
$$P = \bigcup_{\beta<\alpha} \{\pi \in \mathcal{P}(g_\beta) \mid \forall \pi' < \pi \forall \beta \leq \iota < \alpha \colon g_\iota(\pi') = g_\beta(\pi')\}$$

$$\sim \; = \left(\bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota} \right) \cap P \times P$$

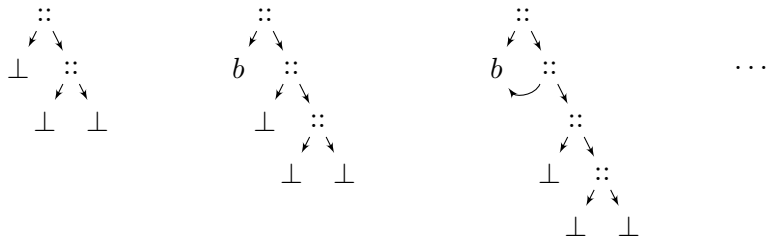
$$l(\pi) = \begin{cases} g_\beta(\pi) & \text{if } \exists \beta < \alpha \forall \beta \leq \iota < \alpha \colon g_\iota(\pi) = g_\beta(\pi) \\ \perp & \text{otherwise} \end{cases} \qquad \text{for all } \pi \in P$$

In particular, given $\beta < \alpha$ and $\pi \in \mathcal{P}(g_\beta)$, we have that $g(\pi) = g_\beta(\pi)$ if $g_\iota(\pi') = g_\beta(\pi')$ for all $\pi' \leq \pi$ and $\beta \leq \iota < \alpha$.

Example 4.8. Figure 5c and 5d on page 407 illustrate two sequences of term graphs $(g_\iota)_{\iota<\omega}$ and $(h_\iota)_{\iota<\omega}$ together with their limit inferiors g_ω respectively h_ω. To see how these limits come about, consider first the sequence of glbs $(\bigsqcap_{\alpha\leq\iota<\omega} g_\iota)_{\alpha<\omega}$ of $(g_\iota)_{\iota<\omega}$:



The lub of this sequence of term graphs is the term graph g_ω. The corresponding sequence $(\bigsqcap_{\alpha \leq \iota < \omega} h_\iota)_{\alpha < \omega}$ of glbs for $(h_\iota)_{\iota < \omega}$ looks as follows:



With each step the number of edges into the b-node increases by one and the $\bot$-nodes move further down the graph structure. The lub of this sequence is the term graph h_ω.

Changing acyclic sharing may, however, expose an oddity of the partial order $\leq_\bot^{\mathsf{S}}$. Let $(g_\iota)_{\iota < \omega}$ be the sequence of term graphs illustrated in Figure 2. The sequence alternates between g_0 and g_1 which differ only in the sharing of the two arguments of the f function symbol. Hence, there is an obvious homomorphism from g_0 to g_1 and we thus have $g_0 \leq_\bot^{\mathsf{S}} g_1$. Therefore, g_0 is the greatest lower bound of every suffix of $(g_\iota)_{\iota < \omega}$, which means that $\liminf_{\iota \to \omega} g_\iota = g_0$.

In our previous work [8], we have used a partial order $\leq_\bot^{\mathsf{R}}$ that is more rigid than $\leq_\bot^{\mathsf{S}}$. In the context of this partial order $\leq_\bot^{\mathsf{R}}$, limit inferior of the sequence illustrated in Figure 2 changes to the term tree $f(\bot, \bot)$ instead of $f(c, c)$.

The difference in the convergence behaviour of $\leq_\bot^{\mathsf{S}}$ and $\leq_\bot^{\mathsf{R}}$ stems from their difference in dealing with sharing, which we have discussed in the beginning of this section: the partial order $\leq_\bot^{\mathsf{S}}$ sees the term graph g_1 as the term graph g_0 with the additional information that the two arguments of f coincide. Since this additional piece of information is not stable throughout the sequence $(g_i)_{i < \omega}$, the limit inferior is only the term graph g_0.

The partial order $\leq_\bot^{\mathsf{R}}$, on the other hand, sees the two term graphs g_0 and g_1 in conflict due to the difference in the arguments of f. Thus, the sequence $(g_i)_{i < \omega}$ is only stable in the root nodes of the term graphs and the limit inferior is consequently the term tree $f(\bot, \bot)$.

In our previous work [8], we chose the rigid partial order as there is a metric space that is "compatible" with it. However, this property of the partial order $\leq_\bot^{\mathsf{R}}$ comes at a price: $\leq_\bot^{\mathsf{R}}$ is quite restrictive in its ability to represent acyclic sharing. For example, the sequence $(h_\iota)_{\iota < \omega}$ of term graphs depicted in Figure 5d does not have the anticipated limit inferior h_ω but instead the term graph obtained from h_ω by relabelling the b-node with $\bot$.

For the partial order $\leq_\perp^{\mathsf{S}}$, we will not be able to find a metric space that is "compatible" with it in the same way and as a consequence we will not obtain the same correspondence that Theorem 2.4 exposed for infinitary term rewriting. In the following section, we will, however, devise a simple metric space that comes close enough to being "compatible" with $\leq_\perp^{\mathsf{S}}$ such that it is possible to regain the correspondence between p-convergence and m-convergence in the setting of strong convergence [9].

5 A Simple Metric on Term Graphs

In this section, we pursue the metric approach to convergence in rewriting systems. To this end, we shall define a metric space on canonical term graphs. We base our approach to defining a metric distance on the definition of the metric distance $\mathbf{d}$ on terms.

Originally, Arnold and Nivat [4] used a notion of truncation of terms to define the metric on terms. The truncation of a term t at depth d, denoted $t|d$, replaces all subterms at depth d by $\perp$:

$$t|0 = \perp, \quad f(t_1, \ldots, t_k)|d+1 = f(t_1|d, \ldots, t_k|d), \quad t|\omega = t$$

For technical reasons, we also define the truncation at depth ω, which does not affect the term at all.

Recall that the metric distance $\mathbf{d}$ on terms is defined by $\mathbf{d}(s,t) = 2^{-\mathsf{sim}(s,t)}$. The underlying notion of similarity $\mathsf{sim}\colon \mathcal{T}^\infty(\Sigma) \times \mathcal{T}^\infty(\Sigma) \to \omega + 1$ can be characterised via truncations:

$$\mathsf{sim}(s,t) = \max\{d \leq \omega \mid s|d = t|d\}$$

We will adopt this approach for term graphs as well. To this end, we will first define abstractly what a truncation on term graphs is and how a metric distance can be derived from it. Then we devise a concrete truncation and show that the induced metric space is in fact complete. We will conclude the section by showing that the metric space we considered is robust in the sense that it is invariant under small changes to the definition of truncation. Lastly, we contrast this finding with the properties of the complete metric that we have previously studied as a candidate for describing convergence on term graphs [8].

5.1 Truncation Functions

As we have seen above, the truncation on terms is a function that, depending on a depth value d, transforms a term t to a term $t|d$. We shall generalise this to term graphs and stipulate some axioms that ensure that we can derive a metric distance in the style of Arnold and Nivat [4]:

Definition 5.1 (truncation function)**.** A family of functions on term graphs $\tau = (\tau_d\colon \mathcal{G}^\infty(\Sigma_\perp) \to \mathcal{G}^\infty(\Sigma_\perp))_{d \leq \omega}$ is called a *truncation function* if it satisfies the following properties for all $g, h \in \mathcal{G}^\infty(\Sigma_\perp)$ and $d \leq \omega$:

(a) $\tau_0(g) \cong \perp$, (b) $\tau_\omega(g) \cong g$, and (c) $\tau_d(g) \cong \tau_d(h) \implies \tau_e(g) \cong \tau_e(h)$ for all $e < d$.

Note that from axioms (b) and (c) it follows that truncation functions must be defined modulo isomorphism, i.e. $g \cong h$ implies $\tau_d(g) \cong \tau_d(h)$ for all $d \leq \omega$.

Given a truncation function, we can define a distance measure in the style of Arnold and Nivat:

Definition 5.2 (truncation-based similarity/distance)**.** Let τ be a truncation function. The *τ-similarity* is the function $\mathsf{sim}_\tau \colon \mathcal{G}^\infty(\Sigma_\bot) \times \mathcal{G}^\infty(\Sigma_\bot) \to \omega + 1$ defined by

$$\mathsf{sim}_\tau(g,h) = \max\{d \leq \omega \mid \tau_d(g) \cong \tau_d(h)\}$$

The *τ-distance* is the function $\mathbf{d}_\tau \colon \mathcal{G}^\infty(\Sigma_\bot) \times \mathcal{G}^\infty(\Sigma_\bot) \to \mathbb{R}_0^+$ that is defined by $\mathbf{d}_\tau(g,h) = 2^{-\mathsf{sim}_\tau(g,h)}$, where $2^{-\omega}$ is interpreted as 0.

The similarity $\mathsf{sim}_\tau(g,h)$ induced by a truncation function τ is well-defined since the axiom (a) of Definition 5.1 insures that the set $\{d \leq \omega \mid \tau_d(g) \cong \tau_d(h)\}$ is not empty. The following proposition confirms that the τ-distance restricted to $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ is indeed an ultrametric:

Proposition 5.3 (truncation-based ultrametric)**.** *For each truncation function τ, the τ-distance $\mathbf{d}_\tau$ constitutes an ultrametric on $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$.*

Proof. The identity respectively the symmetry condition follow by

$$\mathbf{d}_\tau(g,h) = 0 \iff \mathsf{sim}_\tau(g,h) = \omega \iff \tau_\omega(g) \cong \tau_\omega(h) \overset{(*)}{\iff} g \cong h \overset{(**)}{\iff} g = h,$$
$$\text{and} \qquad \mathbf{d}_\tau(g,h) = 2^{-\mathsf{sim}_\tau(g,h)} = 2^{-\mathsf{sim}_\tau(h,g)} = \mathbf{d}_\tau(h,g).$$

The equivalences $(*)$ and $(**)$ are valid by axiom (b) of Definition 5.1 and Proposition 3.16, respectively. For the strong triangle condition, we have to show that

$$\mathsf{sim}_\tau(g_1,g_3) \geq \min\{\mathsf{sim}_\tau(g_1,g_2), \mathsf{sim}_\tau(g_2,g_3)\}.$$

With $d = \min\{\mathsf{sim}_\tau(g_1,g_2), \mathsf{sim}_\tau(g_2,g_3)\}$ we have, by axiom (c) of Definition 5.1, that $\tau_d(g_1) \cong \tau_d(g_2)$ and $\tau_d(g_2) \cong \tau_d(g_3)$. Since we have that $\tau_d(g_1) \cong \tau_d(g_3)$ then, we can conclude that $\mathsf{sim}_\tau(g_1,g_3) \geq d$. □

Given their particular structure, we can reformulate the characterisation of Cauchy sequences and convergence in metric spaces induced by truncation functions in terms of the truncation function itself:

Lemma 5.4. *For each truncation function τ, term graph $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$, and sequence $(g_\iota)_{\iota<\alpha}$ in $\mathcal{G}^\infty_\mathcal{C}(\Sigma))$ the following holds:*

(i) $(g_\iota)_{\iota<\alpha}$ is Cauchy in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\tau)$ iff for each $d < \omega$ there is some $\beta < \alpha$ such that $\tau_d(g_\gamma) \cong \tau_d(g_\iota)$ for all $\beta \leq \gamma, \iota < \alpha$.

(ii) $(g_\iota)_{\iota<\alpha}$ converges to g in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\tau)$ iff for each $d < \omega$ there is some $\beta < \alpha$ such that $\tau_d(g) \cong \tau_d(g_\iota)$ for all $\beta \leq \iota < \alpha$.

Proof. We only show (i) as (ii) follows analogously. For "only if" direction assume that $(g_\iota)_{\iota<\alpha}$ is Cauchy and that $d < \omega$. We then find some $\beta < \alpha$ such that $\mathbf{d}_\tau(g_\gamma, g_\iota) < 2^{-d}$ for all $\beta \leq \gamma, \iota < \alpha$. Hence, we obtain that $\mathsf{sim}_\tau(g_\gamma, g_\iota) > d$ for

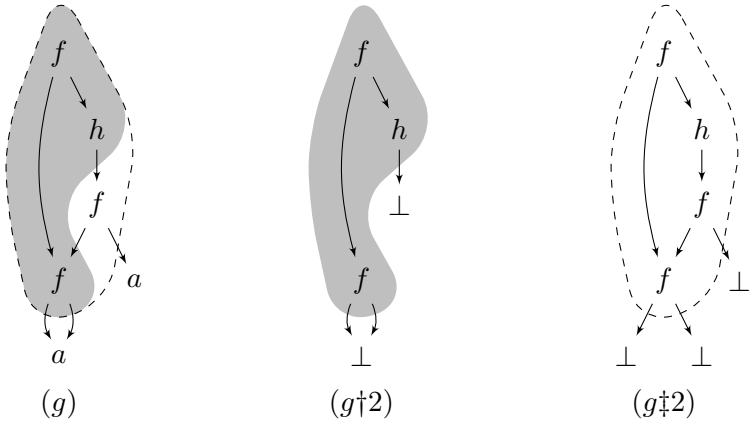


Figure 3: Comparing simple and rigid truncations.

all $\beta \leq \gamma, \iota < \alpha$. That is, $\tau_e(g_\gamma) \cong \tau_e(g_\iota)$ for some $e > d$. According to axiom (c) of Definition 5.1, we can then conclude that $\tau_d(g_\gamma) \cong \tau_d(g_\iota)$ for all $\beta \leq \gamma, \iota < \alpha$.

For the "if" direction assume some positive real number $\varepsilon \in \mathbb{R}^+$. Then there is some $d < \omega$ with $2^{-d} \leq \varepsilon$. By the initial assumption we find some $\beta < \alpha$ with $\tau_d(g_\gamma) \cong \tau_d(g_\iota)$ for all $\beta \leq \gamma, \iota < \alpha$, i.e. $\mathsf{sim}_\tau(g_\gamma, g_\iota) \geq d$. Hence, we have that $\mathbf{d}_\tau(g_\gamma, g_\iota) = 2^{-\mathsf{sim}_\tau(g_\gamma, g_\iota)} < 2^{-d} \leq \varepsilon$ for all $\beta \leq \gamma, \iota < \alpha$. □

5.2 The Simple Truncation and its Metric Space

In this section, we consider a straightforward truncation function that simply cuts off all nodes at the given depth d. The metric that we obtain from this truncation will be the companion metric for the simple partial order $\leq_\perp^{\mathsf{S}}$.

Definition 5.5 (simple truncation)**.** Let $g \in \mathcal{G}^\infty(\Sigma_\perp)$ and $d \leq \omega$. The *simple truncation* $g{\dagger}d$ of g at d is the term graph defined as follows:

$$N^{g\dagger d} = \left\{ n \in N^g \;\middle|\; \mathsf{depth}_g(n) \leq d \right\}$$
$$r^{g\dagger d} = r^g$$
$$\mathsf{lab}^{g\dagger d}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } \mathsf{depth}_g(n) < d \\ \perp & \text{if } \mathsf{depth}_g(n) = d \end{cases}$$
$$\mathsf{suc}^{g\dagger d}(n) = \begin{cases} \mathsf{suc}^g(n) & \text{if } \mathsf{depth}_g(n) < d \\ \langle\rangle & \text{if } \mathsf{depth}_g(n) = d \end{cases}$$

One can easily see that the truncated term graph $g{\dagger}d$ is obtained from g by relabelling all nodes at depth d to $\perp$, removing all their outgoing edges and then removing all nodes that thus become unreachable from the root. This makes the simple truncation a straightforward generalisation of the truncation on terms.

Figure 3 shows a term graph g and its simple truncation at depth $d = 2$. The shaded part of the term graph g comprises the nodes at depth $< d$. Note that a node can get truncated even though some its successor are retained.

The simple truncation indeed induces a truncation function:

Proposition 5.6. *Let † be the function with $\dagger_d(g) = g\dagger d$ for all $d \leq \omega$. Then † is a truncation function.*

Proof. (a) and (b) of Definition 5.1 follow immediately from the construction of the truncation. For (c) assume that $g\dagger d \cong h\dagger d$. Let $0 \leq e < d$ and let $\phi\colon g\dagger d \to h\dagger d$ be the witnessing isomorphism. Note that simple truncations preserve the depth of nodes, i.e. $\mathsf{depth}_{g\dagger d}(n) = \mathsf{depth}_g(n)$ for all $n \in N^{g\dagger d}$. This can be shown by a straightforward induction on $\mathsf{depth}_g(n)$. Moreover, by Corollary 3.14 also isomorphisms preserve the depth of nodes. Hence,

$$\mathsf{depth}_h(\phi(n)) = \mathsf{depth}_{h\dagger d}(\phi(n)) = \mathsf{depth}_{g\dagger d}(n) = \mathsf{depth}_g(n) \quad \text{for all } n \in N^{g\dagger d}$$

Restricting ϕ to the nodes in $g\dagger e$ thus yields an isomorphism from $g\dagger e$ to $h\dagger e$. □

Next we show that the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ that is induced by the truncation function † is in fact complete. To do this, we give a characterisation of the simple truncation in terms of labelled quotient trees.

Lemma 5.7 (labelled quotient tree of a simple truncation)**.** *Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \leq \omega$. The simple truncation $g\dagger d$ is uniquely determined up to isomorphism by the labelled quotient tree $(P, l, \sim)$ with*

(a) $P = \{\pi \in \mathcal{P}(g) \mid \forall \pi_1 < \pi \exists \pi_2 \sim_g \pi_1 \text{ with } |\pi_2| < d\}$,

(b) $l(\pi) = \begin{cases} g(\pi) & \text{if } \exists \pi' \sim_g \pi \text{ with } |\pi'| < d \\ \bot & \text{otherwise} \end{cases}$

(c) $\sim \; = \; \sim_g \cap\, P \times P$

Proof. We just have to show that $(P, l, \sim)$ is the labelled quotient tree induced by $g\dagger d$. Then the lemma follows from Lemma 3.18. The case $d = \omega$ is trivial. In the following we assume that $d < \omega$.

At first, note that

$$\text{for each } \pi \in \mathcal{P}(g\dagger d) \text{ we have that } \pi \in \mathcal{P}(g) \text{ and } \mathsf{node}_{g\dagger d}(\pi) = \mathsf{node}_g(\pi). \quad (*)$$

This can be shown by an induction on the length of π: the case $\pi = \langle\rangle$ is trivial. If $\pi = \pi' \cdot \langle i \rangle$, let $n = \mathsf{node}_{g\dagger d}(\pi')$ and $m = \mathsf{node}_{g\dagger d}(\pi)$. Hence, $m = \mathsf{suc}^{g\dagger d}_i(n)$ and, by construction of $g\dagger d$, also $m = \mathsf{suc}^g_i(n)$. Since by induction hypothesis $n = \mathsf{node}_g(\pi')$, we can thus conclude that $\pi \in \mathcal{P}(g)$ and that $\mathsf{node}_g(\pi) = m = \mathsf{node}_{g\dagger d}(\pi)$.

(a) $P = \mathcal{P}(g\dagger d)$. For the "$\subseteq$" direction let $\pi \in P$. We show by induction on the length of π that $\pi \in \mathcal{P}(g\dagger d)$. The case $\pi = \langle\rangle$ is trivial. If $\pi = \pi_1 \cdot \langle i \rangle$, then by induction hypothesis $\pi_1 \in \mathcal{P}(g\dagger d)$. Let $n = \mathsf{node}_{g\dagger d}(\pi_1)$. By $(*)$, we know that $n = \mathsf{node}_g(\pi_1)$. Since $\pi_1 \cdot \langle i \rangle \in P$, there is some $\pi_2 \sim_g \pi_1$ with $|\pi_2| < d$. That is, $\mathsf{depth}_g(n) < d$. Therefore, we have that $\mathsf{suc}^{g\dagger d}(n) = \mathsf{suc}^g(n)$. Since $\pi_1 \in \mathcal{P}_{g\dagger d}(n)$, this means that $\pi_1 \cdot \langle i \rangle \in \mathcal{P}(g\dagger d)$.

For the "$\supseteq$" direction, assume some $\pi \in \mathcal{P}(g\dagger d)$. By $(*)$, π is also a position in g. To show that $\pi \in P$, let $\pi_1 < \pi$. Since only nodes of depth smaller than d can have a successor node in $g\dagger d$, the node $\mathsf{node}_{g\dagger d}(\pi_1)$ in $g\dagger d$ is at depth smaller

than d. Hence, there is some $\pi_2 \sim_{g\dagger d} \pi_1$ with $|\pi_2| < d$. Because $\pi_2 \sim_{g\dagger d} \pi$ implies, by $(*)$, that $\pi_2 \sim_g \pi$, we can conclude that $\pi \in P$.

(b) $l(\pi) = g\dagger d(\pi)$ for all $\pi \in P$. Let $\pi \in P$ and $n = \mathsf{node}_g(\pi)$. We distinguish two cases. At first suppose that there is some $\pi' \sim_g \pi$ with $|\pi'| < d$. Then $l(\pi) = g(\pi)$. Since $n = \mathsf{node}_g(\pi')$, we have that $\mathsf{depth}_g(n) < d$. Consequently, $\mathsf{lab}^{g\dagger d}(n) = \mathsf{lab}^g(n)$ and, therefore, $g\dagger d(\pi) = g(\pi) = l(\pi)$. In the other case that there is no $\pi' \sim_g \pi$ with $|\pi| < d$, we have $l(\pi) = \bot$. This also means that $\mathsf{depth}_g(n) = d$. Consequently, $g\dagger d(\pi) = \mathsf{lab}^{g\dagger d}(n) = \bot = l(\pi)$.

(c) $\sim = \sim_{g\dagger d}$. Using the fact that $P = \mathcal{P}(g\dagger d)$, we can conclude the following for all $\pi_1, \pi_2 \in P$:

$$\begin{aligned} \pi_1 \sim_{g\dagger d} \pi_2 &\iff \mathsf{node}_{g\dagger d}(\pi_1) = \mathsf{node}_{g\dagger d}(\pi_2) \\ &\overset{(*)}{\iff} \mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2) \\ &\iff \pi_1 \sim_g \pi_2 \\ &\iff \pi_1 \sim \pi_2 \end{aligned}$$

□

Notice that a position π is retained by a truncation, i.e. $\pi \in P$, iff each node that π passes through is at a depth smaller than d (and is thus not truncated or relabelled).

From this characterisation we immediately obtain the following relation between a term graph and its simple truncations:

Corollary 5.8. *Given $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \leq \omega$, we have the following:*

(i)	$\pi \in \mathcal{P}(g)$ *iff* $\pi \in \mathcal{P}(g\dagger d)$	*for all* π *with* $\lvert\pi\rvert \leq d$.
(ii)	$g\dagger d(\pi) = g(\pi)$	*for all* $\pi \in \mathcal{P}(g)$ *with* $\lvert\pi\rvert < d$.
(iii)	$\pi_1 \sim_g \pi_2$ *iff* $\pi_1 \sim_{g\dagger d} \pi_2$	*for all* $\pi_1, \pi_2 \in \mathcal{P}(g)$ *with* $\lvert\pi_1\rvert, \lvert\pi_2\rvert \leq d$.

Proof. Using the reflexivity of $\sim_g$, (i) follows immediately from Lemma 5.7 (a). Using (i), we obtain (ii) and (iii) immediately from Lemma 5.7 (b) and (c), respectively. □

As expected, we also obtain the following relation between the simple truncation and the simple partial order:

Corollary 5.9. *For each $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \leq \omega$, we have that $g\dagger d \leq^{\mathsf{S}}_\bot g$.*

Proof. Immediate from the characterisation of the simple truncation and the simple partial order in Lemma 5.7 and Corollary 4.3, respectively. □

We can now show that the metric space induced by the simple truncation is complete:

Theorem 5.10. *The metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ is complete. In particular, each Cauchy sequence $(g_\iota)_{\iota<\alpha}$ in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ converges to the canonical term graph given by the following labelled quotient tree $(P, l, \sim)$:*

$$P = \liminf_{\iota\to\alpha} \mathcal{P}(g_\iota) = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \mathcal{P}(g_\iota) \qquad \sim = \liminf_{\iota\to\alpha} \sim_{g_\iota} = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota}$$

$$l(\pi) = g_\beta(\pi) \quad \textit{for some } \beta < \alpha \textit{ with } g_\iota(\pi) = g_\beta(\pi) \textit{ for each } \beta \leq \iota < \alpha$$

Proof. We need to check that $(P, l, \sim)$ is a well-defined labelled quotient tree. At first we show that l is a well-defined function on P. In order to show that l is functional, assume that there are $\beta_1, \beta_2 < \alpha$ such that there is a π with $g_\iota(\pi) = g_{\beta_k}(\pi)$ for all $\beta_k \leq \iota < \alpha$, $k \in \{1, 2\}$. But then we have $g_{\beta_1}(\pi) = g_\beta(\pi) = g_{\beta_2}(\pi)$ for $\beta = \max\{\beta_1, \beta_2\}$.

To show that l is total on P, let $\pi \in P$ and $d = |\pi|$. By Lemma 5.4, there is some $\beta < \alpha$ such that $g_\gamma \dagger d + 1 \cong g_\iota \dagger d + 1$ for all $\beta \leq \gamma, \iota < \alpha$. According to Corollary 5.8, this means that all g_ι for $\beta \leq \iota < \alpha$ agree on positions of length smaller than $d + 1$, in particular π. Hence, $g_\iota(\pi) = g_\beta(\pi)$ for all $\beta \leq \iota < \alpha$, and we have $l(\pi) = g_\beta(\pi)$.

One can easily see that $\sim$ is a binary relation on P: if $\pi_1 \sim \pi_2$, then there is some $\beta < \alpha$ with $\pi_1 \sim_{g_\iota} \pi_2$ for all $\beta \leq \iota < \alpha$. Hence, $\pi_1, \pi_2 \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$ and thus $\pi_1, \pi_2 \in P$.

Similarly, it follows that $\sim$ is an equivalence relation on P. To show reflexivity, assume $\pi \in P$. Then there is some $\beta < \alpha$ such that $\pi \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$. Hence, $\pi \sim_{g_\iota} \pi$ for all $\beta \leq \iota < \alpha$ and, therefore, $\pi \sim \pi$. In the same way symmetry and transitivity follow from the symmetry and transitivity of $\sim_{g_\iota}$.

Finally, we have to show the reachability and the congruence property from Definition 3.17. To show reachability assume some $\pi \cdot \langle i \rangle \in P$. Then there is some $\beta < \alpha$ such that $\pi \cdot \langle i \rangle \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$. Hence, since then also $\pi \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$, we have $\pi \in P$. According to the construction of l, there is also some $\beta \leq \gamma < \alpha$ with $g_\gamma(\pi) = l(\pi)$. Since $\pi \cdot \langle i \rangle \in \mathcal{P}(g_\gamma)$ we can conclude that $i < \mathsf{ar}(l(\pi))$.

To establish congruence assume that $\pi_1 \sim \pi_2$. Consequently, there is some $\beta < \gamma$ such that $\pi_1 \sim_{g_\iota} \pi_2$ for all $\beta \leq \iota < \alpha$. Therefore, we also have for each $\beta \leq \iota < \alpha$ that $\pi_1 \cdot \langle i \rangle \sim_{g_\iota} \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(g_\iota(\pi_1))$ and that $g_\iota(\pi_1) = g_\iota(\pi_2)$. According to the construction of l, there some $\beta \leq \gamma < \alpha$ such that $l(\pi_1) = g_\gamma(\pi_1) = g_\gamma(\pi_2) = l(\pi_2)$. Moreover, we can derive that $\pi_1 \cdot \langle i \rangle \sim \pi_2 \cdot \langle i \rangle$ for all $i < \mathsf{ar}(l(\pi_1))$.

This concludes the proof that $(P, l, \sim)$ is indeed a labelled quotient tree. Next, we show that the sequence $(g_\iota)_{\iota < \alpha}$ converges to the thus defined canonical term graph g. By Lemma 5.4, this amounts to giving for each $d < \omega$ some $\beta < \alpha$ such that $g \dagger d \cong g_\iota \dagger d$ for all $\beta \leq \iota < \alpha$.

To this end, let $d < \omega$. Since $(g_\iota)_{\iota < \alpha}$ is Cauchy, there is, according to Lemma 5.4, some $\beta < \alpha$ such that

$$g_\iota \dagger d \cong g_\gamma \dagger d \quad \text{for all } \beta \leq \iota, \gamma < \alpha. \tag{$*$}$$

In order to show that this implies that $g \dagger d \cong g_\iota \dagger d$ for all $\beta \leq \iota < \alpha$, we show that the respective labelled quotient trees of $g \dagger d$ and $g_\iota \dagger d$ as characterised by Lemma 5.7 coincide. The labelled quotient tree $(P_1, l_1, \sim_1)$ for $g \dagger d$ is given by

$$\begin{aligned} P_1 &= \{\pi \in P \mid \forall \pi_1 < \pi \exists \pi_2 \sim \pi_1 : |\pi_2| < d\} \\ \sim_1 &= \sim \cap\, P_1 \times P_1 \\ l_1(\pi) &= \begin{cases} l(\pi) & \text{if } \exists \pi' \sim \pi : |\pi'| < d \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

The labelled quotient tree $(P_2^\iota, l_2^\iota, \sim_2^\iota)$ for each $g_\iota \dagger d$ is given by

$$\begin{aligned} P_2^\iota &= \{\pi \in \mathcal{P}(g_\iota) \mid \forall \pi_1 < \pi \exists \pi_2 \sim_{g_\iota} \pi_1 : |\pi_2| < d\} \\ \sim_2^\iota &= \sim_{g_\iota} \cap\, P_2^\iota \times P_2^\iota \\ l_2^\iota(\pi) &= \begin{cases} g_\iota(\pi) & \text{if } \exists \pi' \sim_{g_\iota} \pi : |\pi'| < d \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

Due to $(*)$, all $(P_2^\iota, l_2^\iota, \sim_2^\iota)$ with $\beta \leq \iota < \alpha$ are pairwise equal. Therefore, we write $(P_2, l_2, \sim_2)$ for this common labelled quotient tree. That is, it remains to be shown that $(P_1, l_1, \sim_1)$ and $(P_2, l_2, \sim_2)$ are equal.

(a) $P_1 = P_2$. For the "$\subseteq$" direction let $\pi \in P_1$. If $\pi = \langle\rangle$, we immediately have that $\pi \in P_2$. Hence, we can assume that π is non-empty. Since $\pi \in P_1$ implies $\pi \in P$, there is some $\beta \leq \beta' < \alpha$ with $\pi \in \mathcal{P}(g_\iota)$ for all $\beta' \leq \iota < \alpha$. Moreover this means that for each $\pi_1 < \pi$ there is some $\pi_2 \sim \pi_1$ with $|\pi_2| < d$. That is, there is some $\beta' \leq \gamma_{\pi_1} < \alpha$ such that $\pi_2 \sim_{g_\iota} \pi_1$ for all $\gamma_{\pi_1} \leq \iota < \alpha$. Since there are only finitely many proper prefixes $\pi_1 < \pi$ but at least one, we can define $\gamma = \max\{\gamma_{\pi_1} \mid \pi_1 < \pi\}$ such that we have for each $\pi_1 < \pi$ some $\pi_2 \sim_{g_\gamma} \pi_1$ with $|\pi_2| < d$. Hence, $\pi \in P_2^\gamma = P_2$.

To show the converse direction, assume that $\pi \in P_2$. Then $\pi \in P_2^\iota \subseteq \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$. Hence, $\pi \in P$. To show that $\pi \in P_1$, assume some $\pi_1 < \pi$. Since $\pi \in P_2^\beta$, there is some $\pi_2 \sim_{g_\beta} \pi_1$ with $|\pi_2| < d$. Then $\pi_1 \in P_2$ because P_2 is closed under prefixes and $\pi_2 \in P_2$ because $|\pi_2| < d$. Thus, $\pi_2 \sim_2 \pi_1$ which implies $\pi_2 \sim_{g_\iota} \pi_1$ for all $\beta \leq \iota < \alpha$. Consequently, $\pi_2 \sim \pi_1$, which means that $\pi \in P_1$.

(c) $\sim_1 = \sim_2$. For the "$\subseteq$" direction assume $\pi_1 \sim_1 \pi_2$. Hence, $\pi_1 \sim \pi_2$ and $\pi_1, \pi_2 \in P_1 = P_2$. This means that there is some $\beta \leq \gamma < \alpha$ with $\pi_1 \sim_{g_\gamma} \pi_2$. Consequently, $\pi_1 \sim_2 \pi_2$. For the converse direction assume that $\pi_1 \sim_2 \pi_2$. Then $\pi_1, \pi_2 \in P_2 = P_1$ and $\pi_1 \sim_{g_\iota} \pi_2$ for all $\beta \leq \iota < \alpha$. Hence, $\pi_1 \sim \pi_2$ and we can conclude that $\pi_1 \sim_1 \pi_2$.

(b) $l_1 = l_2$. We show this by proving that, for all $\beta \leq \iota < \alpha$, the condition $\exists \pi' \sim \pi : |\pi'| < d$ from the definition of l_1 is equivalent to the condition $\exists \pi' \sim_{g_\iota} \pi : |\pi'| < d$ from the definition of l_2 and that $l(\pi) = g_\iota(\pi)$ if either condition is satisfied. The latter is simple: whenever there is some $\pi' \sim \pi$ with $|\pi'| < d$, then $g_\iota(\pi) = l_2^\iota(\pi) = l_2^\beta(\pi) = g_\beta(\pi)$ for all $\beta \leq \iota < \alpha$. Hence, $l(\pi) = g_\beta(\pi) = g_\iota(\pi)$ for all $\beta \leq \iota < \alpha$. For the former, we first consider the "only if" direction of the equivalence. Let $\pi \in P_1$ and $\pi' \sim \pi$ with $|\pi'| < d$. Then also $\pi' \in P_1$ which means that $\pi' \sim_1 \pi$. Since then $\pi' \sim_2 \pi$, we can conclude that $\pi' \sim_{g_\iota} \pi$ for all $\beta \leq \iota < \alpha$. For the converse direction assume that $\pi \in P_2$, $\pi' \sim_{g_\iota} \pi$ and $|\pi'| < d$. Then also $\pi' \in P_2$ which means that $\pi' \sim_2 \pi$. This implies $\pi' \sim_1 \pi$, which in turn implies $\pi' \sim \pi$. □

Example 5.11. Reconsider the two sequences of term graphs $(g_\iota)_{\iota<\omega}$ and $(h_\iota)_{\iota<\omega}$ from Figure 5c respectively 5d on page 407. The simple truncation of the term graphs g_ι at depth 2 alternates between the term trees $a :: \bot :: \bot$ and $b :: \bot :: \bot$. More precisely, $g_\iota \dagger 2 = a :: \bot :: \bot$ if ι is even and $g_\iota \dagger 2 = b :: \bot :: \bot$ if ι is odd. According to Lemma 5.4, this means that $(g_\iota)_{\iota<\omega}$ is not Cauchy in $(\mathcal{T}^\infty(\Sigma), \mathbf{d}_\dagger)$ and is consequently not convergent.

On the other hand, $(h_\iota)_{\iota<\omega}$ does converge to h_ω in $(\mathcal{T}^\infty(\Sigma), \mathbf{d}_\dagger)$: for each $d \in \mathbb{N}$ we have that $h_\omega \dagger d + 1 \cong h_\iota \dagger d + 1$ for all $d \leq \iota < \omega$. Lemma 5.4 then yields that $\lim_{\iota\to\omega} h_\iota = h_\omega$.

As we have seen in Example 4.8, the limit inferior induced by $\leq_\perp^{\mathsf{S}}$ showed some curios behaviour for the sequence of term graphs illustrated in Figure 2. This is not the case for the metric $\mathbf{d}_\dagger$. In fact, there is no topological space in which $(g_\iota)_{\iota<\omega}$ from Figure 2 converges to a unique limit. In particular, this means that there is no metric space in which $(g_\iota)_{\iota<\omega}$ converges.

5.3 Other Truncation Functions and Their Metric Spaces

Generalising concepts from terms to term graphs is not a straightforward matter as we have to decide how to deal with additional sharing that term graphs offer. The definition of simple truncation seems to be an obvious choice for a generalisation of tree truncation. In this section, we shall formally argue that it is in fact the case. More specifically, we show that no matter how we define the sharing of the $\perp$-nodes that fill the holes caused by the truncation, we obtain the same topology.

The following lemma is a handy tool for comparing metric spaces induced by truncation functions:

Lemma 5.12. *Let τ, υ be two truncation functions on $\mathcal{G}^\infty(\Sigma_\perp)$ and $f\colon \mathcal{G}_\mathcal{C}^\infty(\Sigma) \to \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ a function on $\mathcal{G}_\mathcal{C}^\infty(\Sigma)$. Then the following are equivalent*

(i) f is a continuous mapping $f\colon (\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\tau) \to (\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\upsilon)$

(ii) For each $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ and $d < \omega$ there is some $e < \omega$ such that

$$\mathsf{sim}_\tau(g,h) \geq e \quad \Longrightarrow \quad \mathsf{sim}_\upsilon(f(g), f(h)) \geq d \quad \text{for all } h \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$$

(iii) For each $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ and $d < \omega$ there is some $e < \omega$ such that

$$\tau_e(g) \cong \tau_e(h) \quad \Longrightarrow \quad \upsilon_d(f(g)) \cong \upsilon_d(f(h)) \quad \text{for all } h \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$$

Proof. Analogous to Lemma 5.4. □

An easy consequence of the above lemma is that if two truncation functions only differ by a constant depth, they induce the same topology:

Proposition 5.13. *Let τ, υ be two truncation functions on $\mathcal{G}^\infty(\Sigma_\perp)$ such that there is a $\delta < \omega$ with $|\mathsf{sim}_\tau(g,h) - \mathsf{sim}_\upsilon(g,h)| \leq \delta$ for all $g, h \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$. Then $(\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\tau)$ and $(\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\upsilon)$ are topologically equivalent, i.e. they induce the same topology.*

Proof. We show that the identity function $\mathsf{id}\colon \mathcal{G}_\mathcal{C}^\infty(\Sigma) \to \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ is a homeomorphism from $(\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\tau)$ to $(\mathcal{G}_\mathcal{C}^\infty(\Sigma), \mathbf{d}_\upsilon)$, i.e. both id and id^{-1} are continuous. Due to the symmetry of the setting it suffices to show that id is continuous. To this end, let $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ and $d < \omega$. Define $e = d + \delta$ and assume some $h \in \mathcal{G}_\mathcal{C}^\infty(\Sigma)$ such that $\mathsf{sim}_\tau(g,h) \geq e$. By Lemma 5.12, it remains to be shown that then $\mathsf{sim}_\upsilon(g,h) \geq d$. Indeed, we have $\mathsf{sim}_\upsilon(g,h) \geq \mathsf{sim}_\tau(g,h) - \delta \geq e - \delta = d$. □

This shows that metric spaces induced by truncation functions are essentially invariant under changes in the truncation function bounded by a constant margin.

Remark 5.14. We should point out that the original definition of the metric on terms by Arnold and Nivat [4] was slightly different from the one we showed here. Recall that we defined similarity as the maximum depth of truncation that ensures equality:

$$\mathsf{sim}_\tau(g,h) = \max\{d \leq \omega \mid \tau_d(g) \cong \tau_d(h)\}$$

Arnold and Nivat, on the other hand, defined it as the minimum truncation depth that still shows inequality:

$$\mathsf{sim}'_\tau(g,h) = \min\{d \leq \omega \mid \tau_d(g) \ncong \tau_d(h)\}$$

However, it is easy to see that either both $\mathsf{sim}_\tau(g,h)$ and $\mathsf{sim}'_\tau(g,h)$ are ω or $\mathsf{sim}'_\tau(g,h) = \mathsf{sim}_\tau(g,h)+1$. Hence, by Proposition 5.13, both definitions yield the same topology.

Proposition 5.13 also shows that two truncation functions induce the same topology if they only differ in way they treat "fringe nodes", i.e. nodes that are introduced in place of the nodes that have been cut off. Since the definition of truncation functions requires that $\tau_0(g) \cong \bot$ and $\tau_\omega(g) \cong g$, we do not give the explicit construction of the truncation for the depths 0 and ω in the examples below.

Example 5.15. Consider the following variant τ of the simple truncation function †. Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ be a term graph. For each $n \in N^g$ and $i \in \mathbb{N}$, we use n^i to denote a fresh node, i.e. $\{n^i \mid n \in N^g, i \in \mathbb{N}\}$ is a set of pairwise distinct nodes not occurring in N^g. Given a depth $0 < d < \omega$, we define the truncation $\tau_d(g)$ as follows:

$$\begin{aligned}
N^{\tau_d(g)} &= N^g_{<d} \uplus N^g_{=d} \\
N^g_{<d} &= \left\{ n \in N^g \,\middle|\, \mathsf{depth}_g(n) < d \right\} \\
N^g_{=d} &= \left\{ n^i \,\middle|\, n \in N^g_{<d}, 0 \leq i < \mathsf{ar}_g(n), \mathsf{suc}^g_i(n) \notin N^g_{<d} \right\} \\
\mathsf{lab}^{\tau_d(g)}(n) &= \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases} \qquad \mathsf{suc}^{\tau_d(g)}_i(n) = \begin{cases} \mathsf{suc}^g_i(n) & \text{if } n^i \notin N^g_{=d} \\ n^i & \text{if } n^i \in N^g_{=d} \end{cases}
\end{aligned}$$

One can easily show that τ is in fact a truncation function. The difference between † and τ is that in the latter we create a fresh node n^i whenever a node n has a successor $\mathsf{suc}^g_i(n)$ that lies at the fringe, i.e. at depth d. Since this only affects the nodes at the fringe and, therefore, only nodes at the same depth d we get the following:

$$\begin{aligned}
g\dagger d \cong h\dagger d \quad &\Longrightarrow \quad \tau_d(g) \cong \tau_d(h), \text{ and} \\
\tau_d(g) \cong \tau_d(h) \quad &\Longrightarrow \quad g\dagger d - 1 \cong h\dagger d - 1.
\end{aligned}$$

Hence, the respectively induced similarities only differ by a constant margin of 1, i.e. we have that $|\mathsf{sim}_\dagger(g,h) - \mathsf{sim}_\tau(g,h)| = 1$. According to Proposition 5.13, this means that $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ and $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\tau)$ are topologically equivalent.

Consider another variant υ of the simple truncation function †. Given a term graph $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and depth $0 < d < \omega$, we define the truncation $\upsilon_d(g)$ as follows:

$$
\begin{aligned}
N^{\upsilon_d(g)} &= N^g_{<d} \uplus N^g_{=d} \\
N^g_{<d} &= \left\{ n \in N^g \,\middle|\, \mathsf{depth}_g(n) < d \right\} \\
N^g_{=d} &= \left\{ n^i \,\middle|\, \begin{array}{l} n \in N^g, \mathsf{depth}_g(n) = d-1, 0 \le i < \mathsf{ar}_g(n) \\ \text{with} \quad \mathsf{suc}^g_i(n) \notin N^g_{<d} \quad \text{or} \quad n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n)) \end{array} \right\}
\end{aligned}
$$

$$
\mathsf{lab}^{\upsilon_d(g)}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases}
\qquad
\mathsf{suc}^{\upsilon_d(g)}(n) = \begin{cases} \mathsf{suc}^g_i(n) & \text{if } n^i \notin N^g_{=d} \\ n^i & \text{if } n^i \in N^g_{=d} \end{cases}
$$

Also υ forms a truncation function as one can easily show. In addition to creating fresh nodes n^i for each successor that is not in the retained nodes $N^g_{<d}$, the truncation function υ creates such new nodes n^i for each cycle that created by a node just above the fringe. Again, as for the truncation function τ, only the nodes at the fringe, i.e. at depth d are affected by this change. Hence, the respectively induced similarities of † and υ only differ by a constant margin of 1, which makes the metric spaces $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ and $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\upsilon)$ topologically equivalent as well.

The robustness of the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ under the changes illustrated above is due to the uniformity of the core definition of the simple truncation which only takes into account the depth. By simply increasing the depth by a constant number, we can compensate for changes in the way fringe nodes are dealt with.

This is much different for the rigid truncation function $g \ddagger d$ that we have used in our previous work [8] in order to derive a complete metric on term graph:

Definition 5.16 (rigid truncation of term graphs)**.** Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \in \mathbb{N}$.

(i) Given $n, m \in N^g$, m is an *acyclic predecessor* of n in g if there is an acyclic position $\pi \cdot \langle i \rangle \in \mathcal{P}^a_g(n)$ with $\pi \in \mathcal{P}_g(m)$. The set of acyclic predecessors of n in g is denoted $\mathsf{Pre}^a_g(n)$.

(ii) The set of *retained nodes* of g at d, denoted $N^g_{<d}$, is the least subset M of N^g satisfying the following conditions for all $n \in N^g$:

$$\text{(T1) } \mathsf{depth}_g(n) < d \implies n \in M \qquad \text{(T2) } n \in M \implies \mathsf{Pre}^a_g(n) \subseteq M$$

(iii) For each $n \in N^g$ and $i \in \mathbb{N}$, we use n^i to denote a fresh node, i.e. $\{ n^i \,|\, n \in N^g, i \in \mathbb{N} \}$ is a set of pairwise distinct nodes not occurring in N^g. The set of *fringe nodes* of g at d, denoted $N^g_{=d}$, is defined as the singleton set $\{r^g\}$ if $d = 0$, and otherwise as the set

$$\left\{ n^i \,\middle|\, \begin{array}{rl} n \in N^g_{<d}, 0 \le i < \mathsf{ar}_g(n) \text{ with} & \mathsf{suc}^g_i(n) \notin N^g_{<d} \\ \text{or} & \mathsf{depth}_g(n) \ge d-1, n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n)) \end{array} \right\}$$

(iv) The *rigid truncation* of g at d, denoted $g‡d$, is the term graph defined by

$$N^{g‡d} = N^g_{<d} \uplus N^g_{=d} \qquad r^{g‡d} = r^g$$

$$\mathsf{lab}^{g‡d}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases} \qquad \mathsf{suc}^{g‡d}_i(n) = \begin{cases} \mathsf{suc}^g_i(n) & \text{if } n^i \notin N^g_{=d} \\ n^i & \text{if } n^i \in N^g_{=d} \end{cases}$$

Additionally, we define $g‡\omega$ to be the term graph g itself.

The idea of this definition of truncation is that not only each node at depth $< d$ is kept – via the closure condition (T1) – but also every acyclic predecessor of such a node – via (T2). In sum, every node on an acyclic path from the root to a node at depth smaller than d is kept. The difference between the two truncation functions † and ‡ are illustrated in Figure 3.

In contrast to the simple truncation †, the rigid truncation function ‡ is quite vulnerable to small changes:

Example 5.17. Consider the following variant τ of the rigid truncation function ‡. Given a term graph $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and depth $d \in \mathbb{N}^+$, we define the truncation $\tau_d(g)$ as follows: the set of retained nodes $N^g_{<d}$ is defined as for the truncation $g‡d$. For the rest we define

$$N^g_{=d} = \{\mathsf{suc}^g_i(n) \mid n \in N^g_{<d}, 0 \leq i < \mathsf{ar}_g(n), \mathsf{suc}^g_i(n) \notin N^g_{<d}\}$$

$$N^{\tau_d(g)} = N^g_{<d} \uplus N^g_{=d}$$

$$\mathsf{lab}^{\tau_d(g)}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases} \qquad \mathsf{suc}^{\tau_d(g)}(n) = \begin{cases} \mathsf{suc}^g(n) & \text{if } n \in N^g_{<d} \\ \langle\rangle & \text{if } n \in N^g_{=d} \end{cases}$$

In this variant of truncation, some sharing of the retained nodes is preserved. Instead of creating fresh nodes for each successor node that is not in the set of retained nodes, we simply keep the successor node. Additionally loops back into the retained nodes are not cut off. This variant of the truncation deals with its retained nodes in essentially the same way as the simple truncation. However, opposed the simple truncation and their variants, this truncation function yields a topology different from the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_‡)$! To see this, consider the two families of term graphs g_n and h_n illustrated in Figure 4. For both families we have that the τ-truncations at depth 2 to $n+2$ are the same, i.e. $\tau_d(g_n) = \tau_2(g_n)$ and $\tau_d(h_n) = \tau_2(h_n)$ for all $2 \leq d \leq n+2$. The same holds for the truncation function ‡. Moreover, since the two leftmost successors of the h-node are not shared in g_n, both truncation functions coincide on g_n, i.e. $g_n‡d = \tau_d(g_n)$. This is not the case for h_n. In fact, they only coincide up to depth 1. In total, we can observe that $\mathsf{sim}_‡(g_n, h_n) = n+2$ but $\mathsf{sim}_\tau(g_n, h_n) = 1$. This means, however, that the sequence $\langle g_0, h_0, g_1, h_1, \ldots\rangle$ converges in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_‡)$ but not in $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\tau)$!

A similar example can be constructed that uses the difference in the way the two truncation functions deal with fringe nodes created by cycles back into the set of retained nodes.

The above discussion should give a first indication why the simple metric $\mathbf{d}_†$ should be preferred over the rigid partial order $\mathbf{d}_‡$: the metric $\mathbf{d}_†$ is not only simpler than $\mathbf{d}_‡$ but also more natural in the sense that we obtain the topology

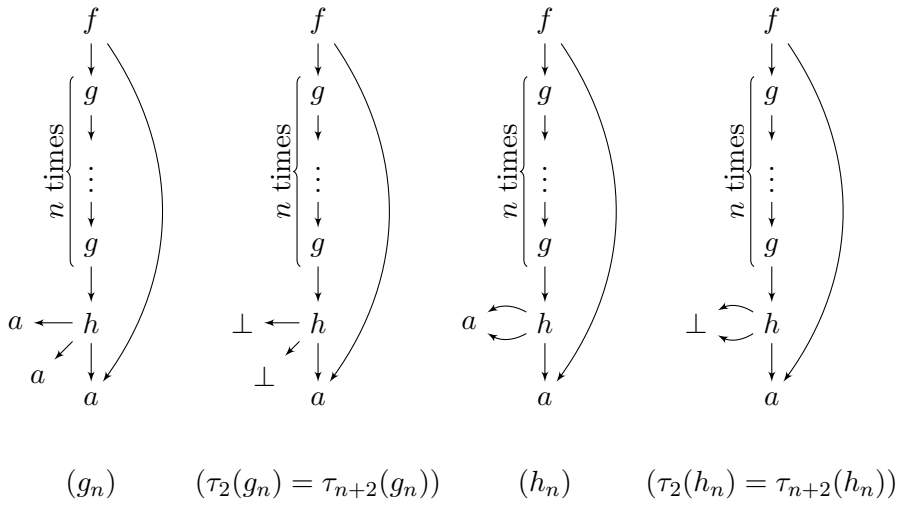


Figure 4: Variations in fringe nodes.

of the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$ without paying too much attention to the corner case details of the underlying truncation function. Small changes in the way we treat these corner cases do not affect the resulting topology as we have illustrated in Example 5.15. For the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d}_\dagger)$, on the other hand, we have to be very careful about how to deal with fringe nodes. As Example 5.17 shows, even small changes yield a different topology. This is part of the reason why the definition of the underlying rigid truncation ‡ is so convoluted.

In Section 8.3, we will give another reason to prefer the metric $\mathbf{d}_\dagger$ over the metric $\mathbf{d}_\ddagger$: while the former allows us to construct the set of term graphs from the set of finite term graphs via metric completion, the latter does not. That is, the rigid metric does not yield a representation of infinite term graphs as the limit of a sequence of finite term graphs.

6 Infinitary Term Graph Rewriting

In the previous sections, we have constructed and investigated the necessary metric and partial order structures upon which the infinitary calculus of term graph rewriting that we shall introduce in this section is based. After describing the framework of term graph rewriting that we consider, we will explore different modes convergence on term graphs. In the same way that infinitary term rewriting instantiates the abstract notions of weak m- and p-convergence [6], infinitary term graph rewriting is an instantiation of these abstract modes of convergence to term graphs.

6.1 Term Graph Rewriting Systems

We base our infinitary term rewriting calculus on the term graph rewriting framework of Barendregt et al. [10]. In order to represent placeholders in rewrite rules, this framework uses variables – in a manner much similar to term rewrite rules. However, instead of open graphs whose unlabelled nodes are interpreted as vari-

ables, we use explicit variable symbols. To this end, we consider a signature $\Sigma_{\mathcal{V}} = \Sigma \uplus \mathcal{V}$ that extends the signature Σ with a set $\mathcal{V}$ of nullary variable symbols.

Definition 6.1 (term graph rewriting system)**.**

(i) Given a signature Σ, a *term graph rule* ρ over Σ is a triple (g, l, r) where g is a graph over $\Sigma_{\mathcal{V}}$ and $l, r \in N^g$, such that all nodes in g reachable from l or r. We write ρ_l respectively ρ_r to denote the left- respectively right-hand side of ρ, i.e. the term graph $g|_l$ respectively $g|_r$. Additionally, we require that, for each variable $v \in \mathcal{V}$, there is at most one node n in g labelled v and n is different but still reachable from l.

(ii) A *term graph rewriting system (GRS)* $\mathcal{R}$ is a pair (Σ, R) with Σ a signature and R a set of term graph rules over Σ.

The requirement that the root l of the left-hand side is not labelled with a variable symbol is analogous to the requirement that the left-hand side of a term rule is not a variable. Similarly, the restriction that nodes labelled with variable symbols must be reachable from the root of the left-hand side corresponds to the restriction on term rules that every variable occurring on the right-hand side must also occur on the left-hand side.

Term graphs can be used to compactly represent terms. This representation of terms is defined by the unravelling of term graphs. This notion can be extended to term graph rules.

Definition 6.2 (unravelling of term graph rules)**.** Let ρ be a term graph rule with ρ_l and ρ_r left- respectively right-hand side term graph. The *unravelling* of ρ, denoted $\mathcal{U}(\rho)$, is the term rule $\mathcal{U}(\rho_l) \to \mathcal{U}(\rho_r)$. Let $\mathcal{R} = (\Sigma, R)$ be a GRS. The unravelling of $\mathcal{R}$, denoted $\mathcal{U}(\mathcal{R})$, is the TRS $(\Sigma, \mathcal{U}(R))$ with $\mathcal{U}(R) = \{\mathcal{U}(\rho) \mid \rho \in G\}$.

Figure 5a illustrates two term graph rules that both represent the term rule $x :: y :: z \to y :: x :: y :: z$ from Example 2.2, which they unravel to.

The application of a rewrite rule ρ (with root nodes l and r) to a term graph g is performed in four steps: at first a suitable sub-term graph of g rooted in some node n of g is *matched* against the left-hand side of ρ. This amounts to finding a $\mathcal{V}$-homomorphism $\phi\colon \rho_l \to_{\mathcal{V}} g|_n$ from the term graph rooted in l to the sub-term graph rooted in n, the *redex*. The $\mathcal{V}$-homomorphism ϕ allows to instantiate variables in the rule with sub-term graphs of the redex. In the second step, nodes and edges in ρ that are not reachable from l are copied into g, such that edges pointing to nodes in the term graph rooted in l are redirected to the image under ϕ. In the last two steps, all edges pointing to n are redirected to (the copy of) r and all nodes not reachable from the root of (the now modified version of) g are removed.

Definition 6.3 (application of a term graph rewrite rule, Barendregt et al. [10])**.** Let $\rho = (N^\rho, \mathsf{lab}^\rho, \mathsf{suc}^\rho, l^\rho, r^\rho)$ be a term graph rewrite rule in a GRS $\mathcal{R} = (\Sigma, R)$, $g \in \mathcal{G}^\infty(\Sigma)$ and $n \in N^g$. ρ is called *applicable* to g at n if there is a $\mathcal{V}$-homomorphism $\phi\colon \rho_l \to_{\mathcal{V}} g|_n$. ϕ is called the *matching $\mathcal{V}$-homomorphism* of

the rule application, and $g|_n$ is called a *ρ-redex*. Next, we define the *result* of the application of the rule ρ to g at n using the $\mathcal{V}$-homomorphism ϕ. This is done by constructing the intermediate graphs g_1 and g_2, and the final result g_3.

(i) The graph g_1 is obtained from g by adding the part of ρ not contained in the left-hand side:

$$N^{g_1} = N^g \uplus (N^\rho \setminus N^{\rho_l})$$

$$\mathsf{lab}^{g_1}(m) = \begin{cases} \mathsf{lab}^g(m) & \text{if } m \in N^g \\ \mathsf{lab}^\rho(m) & \text{if } m \in N^\rho \setminus N^{\rho_l} \end{cases}$$

$$\mathsf{suc}_i^{g_1}(m) = \begin{cases} \mathsf{suc}_i^g(m) & \text{if } m \in N^g \\ \mathsf{suc}_i^\rho(m) & \text{if } m, \mathsf{suc}_i^\rho(m) \in N^\rho \setminus N^{\rho_l} \\ \phi(\mathsf{suc}_i^\rho(m)) & \text{if } m \in N^\rho \setminus N^{\rho_l}, \mathsf{suc}_i^\rho(m) \in N^{\rho_l} \end{cases}$$

(ii) Let $n' = \phi(r^\rho)$ if $r^\rho \in N^{\rho_l}$ and $n' = r^\rho$ otherwise. The graph g_2 is obtained from g_1 by redirecting edges ending in n to n':

$$N^{g_2} = N^{g_1} \qquad \mathsf{lab}^{g_2} = \mathsf{lab}^{g_1} \qquad \mathsf{suc}_i^{g_2}(m) = \begin{cases} \mathsf{suc}_i^{g_1}(m) & \text{if } \mathsf{suc}_i^{g_1}(m) \neq n \\ n' & \text{if } \mathsf{suc}_i^{g_1}(m) = n \end{cases}$$

(iii) The term graph g_3 is obtained by setting the root node r', which is r if $l = r^g$, and otherwise r^g. That is, $g_3 = g_2|_{r'}$. This also means that all nodes not reachable from r' in g_2 are removed.

The above construction induces a *pre-reduction step* $\psi = (g, n, \rho, n', g_3)$ from g to g_3, written $\psi\colon\ g \mapsto_{n,\rho,n'} g_3$. In order to indicate the underlying GRS $\mathcal{R}$, we also write $\psi\colon\ g \mapsto_{\mathcal{R}} g_3$.

Examples for term graph (pre-)reduction steps are shown in Figure 5. We revisit them in more detail in Example 6.8 in the next section.

Note that term graph rules do not provide a duplication mechanism. Each variable is allowed to occur at most once. Duplication must always be simulated by sharing, i.e. with nodes reachable via multiple paths from any of the two roots. This means for example that a variable that should "occur" on the left- and the right-hand side must be shared between the left- and the right-hand side of the rule as seen in the term graph rules in Figure 5a. This sharing can be direct as in ρ_1 – the variable node has multiple ingoing edges – or indirect as in ρ_2 – the variable node is reachable from nodes with multiple ingoing edges. Likewise, for variables that are supposed to be duplicated on the right-hand side, e.g. the variable y in the term rule $x :: y :: z \to y :: x :: y :: z$, we have to use sharing in order to represent multiple occurrence of the same variable as seen in the corresponding term graph rules in Figure 5a: in both rules, the y-node is reachable by two distinct paths from the right-hand side root r.

The definition of term graph rewriting in the form of pre-reduction steps is very operational in style. The result of applying a rewrite rule to a term graph is constructed in several steps by manipulating nodes and edges explicitly. While this is beneficial for implementing a rewriting system this problematic for

reasoning on term graphs up to isomorphisms, which is necessary for introducing notions of convergence. In our case, however, this does not cause any harm since the construction in Definition 6.3 is invariant under isomorphism:

Proposition 6.4 (pre-reduction steps)**.** *Let* $\phi\colon g \mapsto_{n,\rho,m} h$ *be a pre-reduction step in some GRS* $\mathcal{R}$ *and* $\psi_1\colon g' \cong g$*. Then there is a pre-reduction step of the form* $\phi'\colon g' \mapsto_{n',\rho,m'} h'$ *with* $\psi_2\colon h' \cong h$ *such that* $\psi_1(n') = n$ *and* $\psi_1(m') = m$*.*

Proof. Immediate from the construction in Definition 6.3. □

This justifies the following definition of reduction steps:

Definition 6.5 (reduction steps)**.** Let $\mathcal{R} = (\Sigma, R)$ be GRS, $\rho \in R$ and $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ with $n \in N^g$ and $m \in N^h$. A tuple $\phi = (g, n, \rho, m, h)$ is called a *reduction step*, written $\phi\colon g \to_{n,\rho,m} h$, if there is a pre-reduction step $\phi'\colon g' \to_{n',\rho,m'} h'$ with $\mathcal{C}(g') = g$, $\mathcal{C}(h') = h$, $n = \mathcal{P}_{g'}(n')$, and $m = \mathcal{P}_{h'}(m')$. As for pre-reduction step, we also write $\phi\colon g \to_\mathcal{R} h$ or simply $\phi\colon g \to h$ for short.

In other words, a reduction step is a canonicalised pre-reduction step.

6.2 Convergence of Transfinite Reductions

In this section, we shall look at term graph reductions of potentially transfinite length.

Definition 6.6 (reduction)**.** Let $\mathcal{R} = (\Sigma, R)$ be a GRS. A *reduction* in $\mathcal{R}$ is a sequence $(g_\iota \to_\mathcal{R} g_{\iota+1})_{i<\alpha}$ of rewriting steps in $\mathcal{R}$. If S is finite, we write $S\colon g_0 \to^* g_\alpha$.

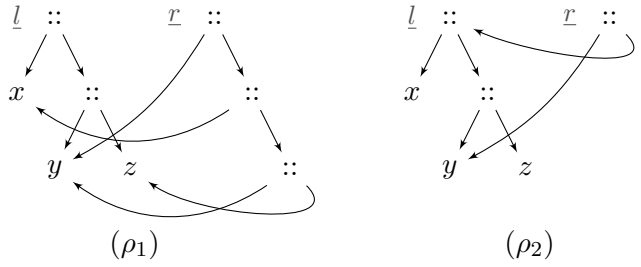
In analogy to infinitary term rewriting, we employ the partial order $\leq^\mathsf{S}_\bot$ and the metric $\mathbf{d}_\dagger$ for the purpose of defining convergence of transfinite term graph reductions.

Definition 6.7 (convergence of reductions)**.** Let $\mathcal{R} = (\Sigma, R)$ be a GRS.

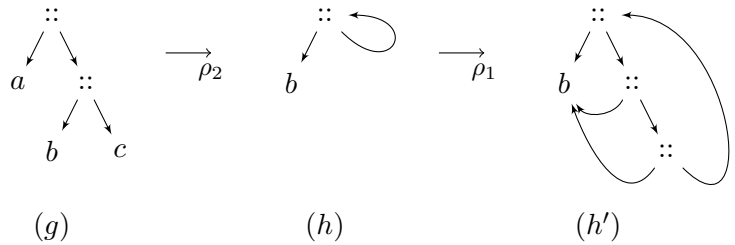
(i) Let $S = (g_\iota \to_\mathcal{R} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}$. S is *weakly m-continuous*, written $S\colon g_0 \xrightarrow{m}_\mathcal{R} \ldots$, if the underlying sequence of term graphs $(g_\iota)_{\iota<\widehat{\alpha}}$ is continuous, i.e. $\lim_{\iota\to\lambda} g_\iota = g_\lambda$ for each limit ordinal $\lambda < \alpha$. S *weakly m-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ in $\mathcal{R}$, written $S\colon g_0 \xrightarrow{m}_\mathcal{R} g$, if it is weakly m-continuous and $\lim_{\iota\to\widehat{\alpha}} g_\iota = g$.

(ii) Let $\mathcal{R}_\bot$ be the GRS $(\Sigma_\bot, R)$ over the extended signature $\Sigma_\bot$ and $S = (g_\iota \to_{\mathcal{R}_\bot} g_{\iota+1})_{\iota<\alpha}$ a reduction in $\mathcal{R}_\bot$. S is *weakly p-continuous*, written $S\colon g_0 \xrightarrow{p}_\mathcal{R} g$, if $\liminf_{\iota<\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$. S *weakly p-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$ in $\mathcal{R}$, written $S\colon g_0 \xrightarrow{p}_\mathcal{R} g$, if it is weakly p-continuous and $\liminf_{\iota<\widehat{\alpha}} g_i = g$.

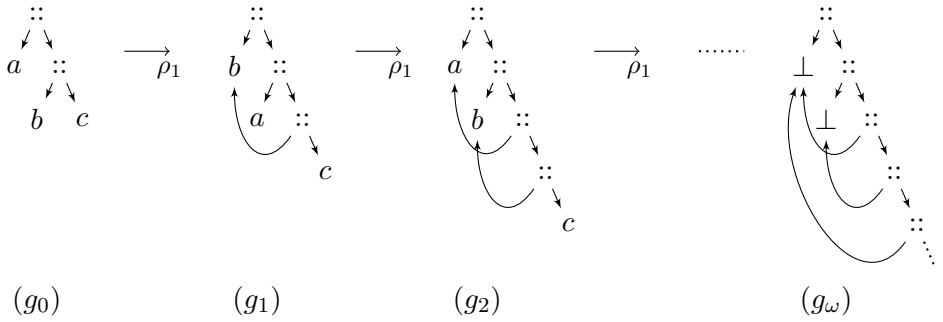
Note that we have to extend the signature of $\mathcal{R}$ to $\Sigma_\bot$ for the definition of weak p-convergence. Moreover, since the partial order $\leq^\mathsf{S}_\bot$ forms a complete semilattice on $\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$, weak p-continuity coincides with weak p-convergence



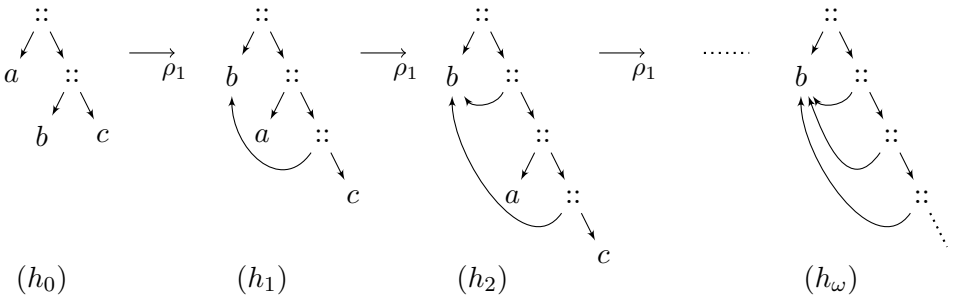
(a) Term graph rules that unravel to $x :: y :: z \to y :: x :: y :: z$.



(b) A ρ_2-step followed by a ρ_1-step.



(c) A term graph reduction over ρ_1 that does not weakly m-converge.



(d) A weakly m-converging term graph reduction over ρ_1.

Figure 5: Term graph rules and their reductions.

Example 6.8. Figure 5a shows two term graph rules that both unravel to the term rule $x :: y :: z \to y :: x :: y :: z$ from Example 2.2. The two rules differ only in their sharing with ρ_1 using "minimal sharing" and ρ_2 using "maximal sharing".

Figure 5c and Figure 5d illustrate term graph reductions that correspond to the term reductions T respectively T' from Example 2.2 and 2.3. All reductions – including the term graph reductions – start from the same term (tree) $a :: b :: c$.

Like the term reduction T, the corresponding term graph reduction in Figure 5c is not weakly m-convergent: as we have illustrated in Example 5.11, the underlying sequence of term graphs is not convergent. On the other hand, the reduction does weakly p-converge to the term graph g_ω, which unravels to the term t to which the reduction T weakly p-converges to.

Similarly, also the reduction in Figure 5d follows its term rewriting counterpart T' closely: It both weakly m- and p-converges to the term graph h_ω, which unravels to the term t' that T' weakly m- and p-converges to. Example 5.11 respectively 4.8 explain how these limits come about.

Due to its higher degree of sharing, the rule ρ_2 permits to arrive at essentially the same result by a single reduction step as seen in Figure 5b. The resulting cyclic term graph h unravels to the same term t' as h_ω. The ρ_1-step that follows illustrates the interaction of rewrite rules with cycles. In fact, if we continue applying the rule ρ_1 after h', we obtain a reduction that weakly m- and p-converges to h_ω.

6.3 m-Convergence vs. p-Convergence

Recall that weak p-convergence in term rewriting is a conservative extension of weak m-convergence (cf. Theorem 2.4). The key property that makes this possible is that for each sequence $(t_\iota)_{\iota<\alpha}$ in $\mathcal{T}^\infty(\Sigma)$, we have that $\lim_{\iota\to\alpha} t_\iota = \liminf_{\iota\to\alpha} t_\iota$ whenever $(t_\iota)_{\iota<\alpha}$ converges, or $\liminf_{\iota\to\alpha} t_\iota$ is a total term.

Unfortunately, this is not the case for the metric space and the partial order that we consider on term graphs. As we have shown in Example 5.11, the sequence of term graphs depicted in Figure 2 has a total term graph as its limit inferior although it does not converge in the metric space. In fact, since the sequence in Figure 2 alternates between two distinct term graphs, it does not converge in any Hausdorff space, i.e. in particular, it does not converge in any metric space.

This example shows that we cannot hope to generalise the compatibility property that we have for terms: even if a sequence of total term graphs has a total term graph as its limit inferior, it might not converge. However, the other direction of the compatibility does hold true:

Theorem 6.9. *If* $(g_\iota)_{\iota<\alpha}$ *converges, then* $\lim_{\iota\to\alpha} g_\iota = \liminf_{\iota\to\alpha} g_\iota$.

Proof. In order to prove this property, we will use the construction of the limit respectively the limit inferior of a sequence of term graphs, which we have shown in Theorem 5.10 respectively Corollary 4.7.

According to Theorem 5.10, we have that the canonical term graph $\lim_{\iota\to\alpha} g_\iota$

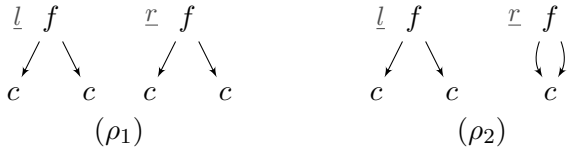


Figure 6: Two term graph rules.

is given by the following labelled quotient tree $(P, \sim, l)$:

$$P = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \mathcal{P}(g_\iota) \qquad \sim = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota}$$
$$l(\pi) = f \quad \text{iff} \quad \exists\beta < \alpha \forall \beta \leq \iota < \alpha : g_\iota(\pi) = f$$

We will show that $g = \liminf_{\iota\to\alpha} g_\iota$ induces the same labelled quotient tree.

From Corollary 4.7, we immediately obtain that $\mathcal{P}(g) \subseteq P$. To show the converse direction $\mathcal{P}(g) \supseteq P$, we assume some $\pi \in P$. According to Corollary 4.7, in order to show that $\pi \in \mathcal{P}(g)$, we have to find a $\beta < \alpha$ such that $\pi \in \mathcal{P}(g_\beta)$ and for each $\pi' < \pi$ there is some $f \in \Sigma_\perp$ such that $g_\iota(\pi') = f$ for all $\beta \leq \iota < \alpha$.

Because $\pi \in P$, there is some $\beta_1 < \alpha$ such that $\pi \in \mathcal{P}(g_\iota)$ for all $\beta_1 \leq \iota < \alpha$. Since $(g_\iota)_{\iota<\alpha}$ converges, it is also Cauchy. Hence, by Lemma 5.4, for each $d < \omega$, there is some $\beta_2 < \alpha$ such that $g_\gamma\dagger d \cong g_\iota\dagger d$ for all $\beta_2 \leq \gamma, \iota < \alpha$. By specialising this to $d = |\pi|$, we obtain some $\beta_2 < \alpha$ with $g_\gamma\dagger |\pi| \cong g_\iota\dagger |\pi|$ for all $\beta_2 \leq \gamma, \iota < \alpha$. Let $\beta = \max\{\beta_1, \beta_2\}$. Then we have $\pi \in \mathcal{P}(g_\iota)$ and $g_\beta\dagger |\pi| \cong g_\iota\dagger |\pi|$ for each $\beta \leq \iota < \alpha$. Hence, for each $\pi' < \pi$, the symbol $f = g_\beta(\pi')$ is well-defined, and, according to Corollary 5.8, we have that $g_\iota(\pi') = f$ for each $\beta \leq \iota < \alpha$.

The equalities $\sim = \sim_g$ and $l = g(\cdot)$ follow from Corollary 4.7 as $P = \mathcal{P}(g)$. □

From this property, we immediately obtain the following relation between weak m- and p-convergence:

Theorem 6.10. *Let S be a reduction in a GRS $\mathcal{R}$.*

$$\textit{If} \quad S\colon g \xrightarrow{m}_{\mathcal{R}} h \qquad \textit{then} \qquad S\colon g \xrightarrow{p}_{\mathcal{R}} h.$$

Proof. Follows straightforwardly from Theorem 6.9. □

However, as we have indicated, weak m-convergence is not the total fragment of weak p-convergence as it is the case for TRSs. The GRS with the two rules depicted in Figure 6 yields the reduction sequence shown in Figure 2. This reduction weakly p-converges to $f(c, c)$ but is not weakly m-convergent.

Yet, if we move from the weak notions of convergence considered here to strong convergence – in analogy to strong convergence in infinitary term rewriting [8, 22] – we do in fact regain the correspondence between metric and partial order convergence: strong p-convergence on term graphs is a conservative extension of strong m-convergence [9].

With the move to strong convergence it also possible to establish that infinitary term graph rewriting is sound and complete w.r.t. term rewriting [9]. Analysing the way in which infinitary term graph rewriting simulates infinitary

term rewriting becomes substantially more difficult in the setting of weak convergence. That is why we only have very limited results of soundness and completeness, which are presented in the following two sections.

7 Preservation of Convergence through Unravelling

In this section, we shall show that the convergence behaviour of term graph sequences – both in terms of metric limit and in terms of the limit inferior – is preserved by the unravelling of term graphs to terms. As we will also show that the metric $\mathbf{d}_\dagger$ and partial order $\leq_\perp^{\mathsf{S}}$ coincide with the metric $\mathbf{d}$ respectively the partial order $\leq_\perp$ if restricted to terms, the preservation of convergence will show that both modes of convergence are sound w.r.t. the modes of convergence used in infinitary term rewriting.

The cornerstone of the investigation of unravellings is the following characterisation in terms of labelled quotient trees:

Proposition 7.1. *The unravelling $\mathcal{U}(g)$ of a term graph $g \in \mathcal{G}^\infty(\Sigma)$ is given by the labelled quotient tree $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$.*

Proof. Since $\mathcal{I}_{\mathcal{P}(g)}$ is a subrelation of $\sim_g$, we know that $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$ is a labelled quotient tree and thus uniquely determines a term tree t. By Lemma 3.13, there is a homomorphism from t to g. Hence, $\mathcal{U}(g) = t$. □

7.1 Metric Convergence

We start with a specialisation of Lemma 5.7, which provides a characterisation of the simple truncation, to term trees:

Lemma 7.2. *Let $t \in \mathcal{T}^\infty(\Sigma_\perp)$ and $d \leq \omega + 1$. The simple truncation $t\dagger d$ is given by the labelled quotient tree $(P, l, \mathcal{I}_P)$ with*

$$P = \{\pi \in \mathcal{P}(t) \mid |\pi| \leq d\} \qquad l(\pi) = \begin{cases} t(\pi) & \text{if } |\pi| < d \\ \perp & \text{if } |\pi| \geq d \end{cases}$$

Proof. Immediate from Lemma 5.7 and the fact that $\sim_t$ is the identity relation $\mathcal{I}_{\mathcal{P}(t)}$ on $\mathcal{P}(t)$. □

This shows that the metric $\mathbf{d}_\dagger$ restricted to terms coincides with the metric $\mathbf{d}$ on terms. Moreover, we can use this in order to relate the metric distance between term graphs and the metric distance between their unravellings.

Lemma 7.3. *For all $g, h \in \mathcal{G}^\infty(\Sigma)$, we have that $\mathbf{d}_\dagger(g,h) \geq \mathbf{d}_\dagger(\mathcal{U}(g), \mathcal{U}(h))$.*

Proof. Let $d = \mathsf{sim}_\dagger(g,h)$. Hence, $g\dagger d \cong h\dagger d$ and we can assume that the corresponding labelled quotient trees as characterised by Lemma 5.7 coincide. We only need to show that $\mathcal{U}(g)\dagger d \cong \mathcal{U}(h)\dagger d$ since then $\mathsf{sim}_\dagger(\mathcal{U}(g), \mathcal{U}(h)) \geq d$ and thus $\mathbf{d}_\dagger(\mathcal{U}(g), \mathcal{U}(h)) \leq 2^{-d} = \mathbf{d}_\dagger(g,h)$. In order to show this, we show that

the labelled quotient trees of $\mathcal{U}(g)\dagger d$ and $\mathcal{U}(h)\dagger d$ as characterised by Lemma 7.2 coincide. For the set of positions we have the following:

$$\begin{aligned}
&\pi \in \mathcal{P}(\mathcal{U}(g)\dagger d) && \\
\iff &\pi \in \mathcal{P}(\mathcal{U}(g)), \quad |\pi| \leq d && \text{(Lemma 7.2)} \\
\iff &\pi \in \mathcal{P}(g), \quad |\pi| \leq d && \text{(Proposition 7.1)} \\
\iff &\pi \in \mathcal{P}(g\dagger d), \quad |\pi| \leq d && \text{(Corollary 5.8)} \\
\iff &\pi \in \mathcal{P}(h\dagger d), \quad |\pi| \leq d && (g\dagger d \cong h\dagger d) \\
\iff &\pi \in \mathcal{P}(h), \quad |\pi| \leq d && \text{(Corollary 5.8)} \\
\iff &\pi \in \mathcal{P}(\mathcal{U}(h)), \quad |\pi| \leq d && \text{(Proposition 7.1)} \\
\iff &\pi \in \mathcal{P}(\mathcal{U}(h)\dagger d) && \text{(Lemma 7.2)}
\end{aligned}$$

In order to show that the labellings are equal, consider some $\pi \in \mathcal{P}(\mathcal{U}(g)\dagger d)$ and assume at first that $|\pi| \geq d$. By Lemma 7.2, we then have $(\mathcal{U}(g)\dagger d)(\pi) = \bot = (\mathcal{U}(h)\dagger d)(\pi)$. Otherwise, if $|\pi| < d$, we obtain the following:

$$\begin{aligned}
(\mathcal{U}(g)\dagger d)(\pi) &\overset{\text{Lem. 7.2}}{=} \mathcal{U}(g)(\pi) \overset{\text{Prop. 7.1}}{=} g(\pi) \overset{\text{Cor. 5.8}}{=} g\dagger d(\pi) \\
&\overset{g\dagger d \cong h\dagger d}{=} h\dagger d(\pi) \overset{\text{Cor. 5.8}}{=} h(\pi) \overset{\text{Prop. 7.1}}{=} \mathcal{U}(h)(\pi) \overset{\text{Lem. 7.2}}{=} (\mathcal{U}(h)\dagger d)(\pi)
\end{aligned}$$

□

This immediately yields that Cauchy sequences are preserved by unravelling:

Lemma 7.4. *If $(g_\iota)_{\iota<\alpha}$ is a Cauchy sequence in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$, then $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ is too.*

Proof. This follows immediately from Lemma 7.3. □

Moreover, we obtain that limits in the metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$ are preserved by unravelling.

Theorem 7.5. *For every sequence $(g_\iota)_{\iota<\alpha}$ that converges to g in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$, we have that $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ converges to $\mathcal{U}(g)$.*

Proof. According to Theorem 5.10, we have that $\mathcal{P}(g) = \liminf_{\iota\to\alpha} \mathcal{P}(g_\iota)$, and that $g(\pi) = g_\beta(\pi)$ for some $\beta < \alpha$ with $g_\iota(\pi) = g_\beta(\pi)$ for all $\beta \leq \iota < \alpha$. By Proposition 7.1, we then obtain $\mathcal{P}(\mathcal{U}(g)) = \liminf_{\iota\to\alpha} \mathcal{P}(\mathcal{U}(g_\iota))$, and that $\mathcal{U}(g)(\pi) = \mathcal{U}(g_\beta)(\pi)$ for some $\beta < \alpha$ with $\mathcal{U}(g_\iota)(\pi) = \mathcal{U}(g_\beta)(\pi)$ for all $\beta \leq \iota < \alpha$. Since by Lemma 7.4, $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ is Cauchy, we can apply Theorem 5.10 to obtain that $\lim_{\iota\to\alpha} \mathcal{U}(g_\iota) = \mathcal{U}(g)$. □

Since Lemma 7.2 confirms that the metric $\mathbf{d}_\dagger$ restricted to terms coincides with the metric $\mathbf{d}$ on terms, we have that convergence on term graphs simulates convergence on terms: if $(g_\iota)_{\iota<\alpha}$ in converges to g in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$, then $(\mathcal{U}(g_\iota))_{\iota<\alpha}$ converges to $\mathcal{U}(g)$ in $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$.

7.2 Partial Order Convergence

At first we derive a characterisation of the partial order $\leq_\bot^{\mathsf{S}}$ on terms by specialising Corollary 4.3:

Lemma 7.6. *Given two terms $s, t \in \mathcal{T}^\infty(\Sigma_\bot)$, we have $s \leq_\bot^{\mathsf{S}} t$ iff $s(\pi) = t(\pi)$ for all $\pi \in \mathcal{P}(s)$ with $g(\pi) \in \Sigma$.*

Proof. Immediate from Corollary 4.3. □

This shows that the partial order $\leq_\bot^{\mathsf{S}}$ on term graphs generalises the partial order $\leq_\bot$ on terms, i.e. $\leq_\bot^{\mathsf{S}}$ restricted to $\mathcal{T}^\infty(\Sigma_\bot)$ coincides with $\leq_\bot$.

From the above finding we easily obtain that the partial order $\leq_\bot^{\mathsf{S}}$ as well as its induced limits are preserved by unravelling:

Theorem 7.7. *In the partially ordered set $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{S}})$ the following holds:*

(i) Given two term graphs g, h, we have that $g \leq_\bot^{\mathsf{S}} h$ implies $\mathcal{U}(g) \leq_\bot^{\mathsf{S}} \mathcal{U}(h)$.

(ii) For each directed set G, we have that $\mathcal{U}\left(\bigsqcup_{g \in G} g\right) = \bigsqcup_{g \in G} \mathcal{U}(g)$.

(iii) For each non-empty set G, we have that $\mathcal{U}\left(\bigsqcap_{g \in G} g\right) = \bigsqcap_{g \in G} \mathcal{U}(g)$.

(iv) For each sequence $(g_\iota)_{\iota < \alpha}$, we have that $\mathcal{U}(\liminf_{\iota \to \alpha} g_\iota) = \liminf_{\iota \to \alpha} \mathcal{U}(g_\iota)$.

Proof. (i) By Corollary 4.3, $g \leq_\bot^{\mathsf{S}} h$ implies that $g(\pi) = h(\pi)$ for all $\pi \in \mathcal{P}(g)$ with $g(\pi) \in \Sigma$. By Proposition 7.1, we then have $\mathcal{U}(g)(\pi) = \mathcal{U}(h)(\pi)$ for all $\pi \in \mathcal{P}(\mathcal{U}(g))$ with $\mathcal{U}(g)(\pi) \in \Sigma$ which, by Lemma 7.6, implies $\mathcal{U}(g) \leq_\bot^{\mathsf{S}} \mathcal{U}(h)$.

By a similar argument (ii) and (iii) follow from the characterisation of least upper bounds and greatest lower bounds in Theorem 4.4 respectively Proposition 4.5 by using Proposition 7.1.

(iv) Follows from (ii) and (iii). □

Since Lemma 7.6 shows that $\leq_\bot^{\mathsf{S}}$ and $\leq_\bot$ coincide on $\mathcal{T}^\infty(\Sigma_\bot)$, we thus obtain that the limit inferior on term graphs simulates the limit inferior on terms: if $\liminf_{\iota \to \alpha} g_\iota = g$ in $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^{\mathsf{S}})$, then $\liminf_{\iota \to \alpha} \mathcal{U}(g_\iota) = \mathcal{U}(g)$ in $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$.

8 Finite Term Graphs

In this section, we want to study the simple partial order $\leq_\bot^{\mathsf{S}}$ and the simple metric $\mathbf{d}_\dagger$ on finite term graphs. On terms, the partial order $\leq_\bot$ and the metric $\mathbf{d}$ allow us to reconstruct the set of (partial) terms from the set of finite (partial) terms via *ideal completion* and *metric completion*, respectively. In the following, we shall show that this generalises to the setting of canonical term graphs.

8.1 Finitary Properties

Since term graphs are finitely branching, we know that, in each term graph, there are only a finite number of positions of a bounded length:

Lemma 8.1 (bounded positions are finite)**.** *Let $g \in \mathcal{G}^\infty(\Sigma)$ and $d < \omega$. Then there are only finitely many positions of length at most d in g, i.e. the set $\{\pi \in \mathcal{P}(g) \mid |\pi| \leq d\}$ is finite.*

Proof. Straightforward induction on d. □

From this we can immediately conclude that the simple truncation of a term graph yields a finite term graph:

Proposition 8.2 (simple truncations are finite)**.** *For each $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d < \omega$, the simple truncation $g{\dagger}d$ is finite, i.e. $g{\dagger}d \in \mathcal{G}(\Sigma_\bot)$.*

Proof. By Lemma 8.1, the set $P = \{\pi \in \mathcal{P}(g) \mid |\pi| \leq d\}$ is finite. Since the function $f\colon P \to N^{g\dagger d}$ defined by $f(\pi) = \mathsf{node}_g(\pi)$ is surjective, we can conclude that $N^{g\dagger d}$ is finite. □

We know that positions describe the structure of a term graph. However, cycles cause infinite repetition of essentially the same structure of a position. Therefore, a finite term graph may have infinitely many positions. In the following, we want to avoid this by considering only *essential positions*:

Definition 8.3 (redundant/essential positions)**.** A position $\pi \in \mathcal{P}(g)$ in a term graph $g \in \mathcal{G}^\infty(\Sigma)$ is called *redundant* if there are $\pi_1, \pi_2 \in \mathcal{P}(g)$ with $\pi_1 < \pi_2 < \pi$ such that $\pi_1 \sim_g \pi_2$. A position that is not redundant is called *essential*. The set of all essential positions of g are denoted $\mathcal{P}^e(g)$; the set of all essential positions of a node n in g are denoted $\mathcal{P}^e_g(n)$.

Note that a position is redundant iff one of its proper prefixes is cyclic. This means that the set $\mathcal{P}^e(g)$ of essential positions is closed under prefixes.

Lemma 8.4 (decomposition of redundant positions)**.** *For each $g \in \mathcal{G}^\infty(\Sigma)$ and $\pi \in \mathcal{P}(g)$, we have that π is redundant iff there are $\pi_1, \pi_2 \in \mathcal{P}^e(g)$ such that $\pi_1 < \pi_2 < \pi$ and $\pi_1 \sim_g \pi_2$.*

Proof. The "if" direction follows immediately from the definition of redundancy. We will show the "only if" direction by induction on the length of π.

If π is redundant in g, then there are $\pi_1, \pi_2 \in \mathcal{P}(g)$ with $\pi_1 < \pi_2 < \pi$ and $\pi_1 \sim_g \pi_2$. If π_2 is essential, then also π_1 is essential since it is a prefix of π_2. Otherwise, if π_2 is redundant, we can apply the induction hypothesis to π_2 to obtain $\pi_1', \pi_2' \in \mathcal{P}^e(g)$ with $\pi_1' < \pi_2' < \pi_2$ and $\pi_1' \sim_g \pi_2'$. □

With essential positions, we have a finite representation of the structure of term graphs even if the term graph is cyclic.

Proposition 8.5 (essential positions characterise finiteness)**.** *A term graph $g \in \mathcal{G}^\infty(\Sigma)$ is finite iff $\mathcal{P}^e(g)$ is finite.*

Proof. If g is finite, then let $n = |N^g|$. Whenever a position $\pi \in \mathcal{P}(g)$ is longer than n, then a proper prefix of π passes more than n nodes. By the pigeon hole principle we thus know that there is a node that a proper prefix of π passes twice. Hence, π is redundant. Therefore, we know that every essential position must be of length at most n. Since, according to Lemma 8.1, there are only finitely many such positions in g, we know that $\mathcal{P}^e(g)$ is finite.

If g is infinite, we can apply König's Lemma to obtain an infinite acyclic path (starting in the root of g) that does not pass a node twice. Since each finite prefix of this path is an essential position, there are infinitely many essential positions. □

Indeed, the essential positions of a term graph are sufficient in order to characterise the structure of term graphs in the form of Δ-homomorphisms:

Proposition 8.6 (essential positions characterise Δ-homomorphisms)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, there is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff, for all $\pi, \pi' \in \mathcal{P}^e(g)$, we have*

$$\textit{(a) } \pi \sim_g \pi' \quad \Longrightarrow \quad \pi \sim_h \pi', \textit{ and} \quad \textit{(b) } g(\pi) = h(\pi) \quad \textit{whenever} \quad g(\pi) \notin \Delta.$$

Proof. The "only if" direction follows immediately from Lemma 3.13. For the converse direction, assume that both (a) and (b) hold. Define the function $\phi\colon N^g \to N^h$ by $\phi(n) = m$ iff $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$ for all $n \in N^g$ and $m \in N^h$. To confirm that this is well-defined, we show at first that, for each $n \in N^g$, there is at most one $m \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m)$. Suppose there is another node $m' \in N^h$ with $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(m')$. Since $\mathcal{P}_g(n) \neq \emptyset$, this implies $\mathcal{P}_h(m) \cap \mathcal{P}_h(m') \neq \emptyset$. Hence, $m = m'$. Secondly, we show that there is at least one such node m. We know that each node has at least one essential position. Choose some $\pi^* \in \mathcal{P}^e_g(n)$. Since then $\pi^* \sim_g \pi^*$ and, by (a), also $\pi^* \sim_h \pi^*$ holds, there is some $m \in N^h$ with $\pi^* \in \mathcal{P}_h(m)$. Next we show by induction on the length of π that $\pi \in \mathcal{P}_g(n)$ implies $\pi \in \mathcal{P}_h(m)$. If $\pi \in \mathcal{P}_g(n)$, then $\pi \sim_g \pi^*$. In case that π is essential in g, we obtain $\pi \sim_h \pi^*$ from (a) and thus $\pi \in \mathcal{P}_h(m)$. Otherwise, i.e. if π is redundant in g, we can decompose π into $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3$ such that π_2 and π_3 are non-empty and $\pi_1 \sim_g \pi_1 \cdot \pi_2$. By Lemma 8.4, we can assume that π_1 and $\pi_1 \cdot \pi_2$ are essential in g. Hence, $\pi_1 \sim_g \pi_1 \cdot \pi_2$ implies, by (a), that $\pi_1 \sim_h \pi_1 \cdot \pi_2$. Moreover, $\pi_1 \sim_g \pi_1 \cdot \pi_2$ means that the prefix $\pi_1 \cdot \pi_2$ of π can be replaced by π_1 in g, i.e. $\pi_1 \cdot \pi_3 \in \mathcal{P}_g(n)$. Since $\pi_1 \cdot \pi_3$ is strictly shorter than π, we can apply the induction hypothesis to obtain that $\pi_1 \cdot \pi_3 \in \mathcal{P}_h(m)$. From this and from $\pi_1 \sim_h \pi_1 \cdot \pi_2$ we can then conclude that $\pi_1 \cdot \pi_2 \cdot \pi_3 \in \mathcal{P}_h(m)$.

Using Lemma 3.12, we can see that ϕ is a Δ-homomorphism from g to h: condition (a) of Lemma 3.12 follows immediately from the construction of ϕ and condition (b) of Lemma 3.12 follows from (b) since each node has at least one essential position. □

Consequently, we immediately obtain a characterisation of the simple partial order $\leq^{\mathsf{S}}_\bot$ in terms of essential positions:

Corollary 8.7 (essential positions characterise $\leq^{\mathsf{S}}_\bot$)**.** *Let $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$. Then $g \leq^{\mathsf{S}}_\bot h$ iff the following conditions are met:*

(a) $\pi \sim_g \pi' \quad \Longrightarrow \quad \pi \sim_h \pi' \quad$ *for all* $\pi, \pi' \in \mathcal{P}^e(g)$

(b) $g(\pi) = h(\pi) \quad$ *for all* $\pi \in \mathcal{P}^e(g)$ *with* $g(\pi) \in \Sigma$.

The above characterisation allows us to prove that the lub of a finite number of finite term graphs can only be finite as well:

Proposition 8.8 (lub of finite term graphs is finite)**.** *For each finite set* $G \subseteq_{fin} \mathcal{G}_{\mathcal{C}}(\Sigma_\bot)$ *with an upper bound in* $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{S}}_\bot)$*, we have* $\bigsqcup G \in \mathcal{G}_{\mathcal{C}}(\Sigma_\bot)$.

Proof. Let $G \subseteq_{fin} \mathcal{G}_{\mathcal{C}}(\Sigma_\bot)$ be a finite set with upper bound $\widehat{g}$. If G is empty, then $\bigsqcup G = \bot \in \mathcal{G}_{\mathcal{C}}(\Sigma_\bot)$. Otherwise, we know, by Proposition 2.1, that the complete semilattice $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{S}}_\bot)$ is also bounded complete. Hence, G has a least upper bound $\overline{g}$. Since $\overline{g}$ is an upper bound of G, we find for each $g \in G$ a $\bot$-homomorphism $\phi_g \colon g \to_\bot \overline{g}$. Let $N = \bigcup_{g \in G} Im(\phi_g)$ be the combined image of those $\bot$-homomorphisms. Since each $g \in G$ is finite, also their image $Im(\phi_g)$ is finite and thus so is N. We conclude the proof by showing that $N^{\overline{g}} \subseteq N$, which proves that $\overline{g}$ is finite.

We show that $n \in N^{\overline{g}}$ implies $n \in N$ by induction on $\mathsf{depth}_{\overline{g}}(n)$. If $\mathsf{depth}_{\overline{g}}(n) = 0$, then $n = r^{\overline{g}}$. Choose some $g \in G$. Since then $\phi_g(r^g) = r^{\overline{g}}$, we have that $n \in Im(\phi_g) \subseteq N$. If $\mathsf{depth}_{\overline{g}}(n) > 0$, then there is some $m \in N^{\overline{g}}$ with $\mathsf{depth}_{\overline{g}}(m) < \mathsf{depth}_{\overline{g}}(n)$ and $\mathsf{suc}^{\overline{g}}_i(m) = n$ for some i. Hence, we can apply the induction hypothesis which yields that $m \in N$. Since m has a successor in $\overline{g}$, we have that $\mathsf{lab}^{\overline{g}}(m) \in \Sigma$. Construct the term graph $\widehat{g}$ from $\overline{g}$ by relabelling m to $\bot$ and removing all its outgoing edges as well as all nodes that thus become unreachable. The mapping $\phi \colon N^{\widehat{g}} \to N^{\overline{g}}$ given by $\phi(\widehat{n}) = \widehat{n}$ for all $\widehat{n} \in N^{\widehat{g}}$ is a $\bot$-homomorphism. Thus $\mathcal{C}(\widehat{g}) <^{\mathsf{S}}_\bot \overline{g}$. However, since $\overline{g}$ is the least upper bound of G, $\mathcal{C}(\widehat{g})$ cannot be an upper bound of G. But, for each $g \in G$, the mapping ϕ_g is also a $\bot$-homomorphism from g to $\widehat{g}$ provided each $m' \in N^g$ with $\phi_g(m') = m$ is labelled $\bot$ in g. Since this cannot be the case for all $g \in G$, we find some $g \in G, m' \in N^g$ such that $\phi_g(m') = m$ and $\mathsf{lab}^g(m') \in \Sigma$. Since ϕ_g is then homomorphic in m', we know that m' has an i-th successor in g such that

$$\phi_g(\mathsf{suc}^g_i(m')) = \mathsf{suc}^{\overline{g}}_i(\phi_g(m')) = \mathsf{suc}^{\overline{g}}_i(m) = n.$$

Hence, $n \in Im(\phi_g) \subseteq N$. □

8.2 Ideal Completion

In this section, we shall show that the set $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$ of (potentially infinite) canonical term graphs can be constructed from the set $\mathcal{G}_{\mathcal{C}}(\Sigma_\bot)$ of finite canonical term graphs via the ideal completion of the partially ordered set $(\mathcal{G}_{\mathcal{C}}(\Sigma_\bot), \leq^{\mathsf{S}}_\bot)$.

Given a partially order set, its ideal completion provides an extension of the original partially ordered set that is a cpo.

Definition 8.9 (ideal, ideal completion)**.** Let $(A, \leq)$ be a partially ordered set and $B \subseteq A$.

(i) The set B is called *downward-closed* if for all $a \in A, b \in B$ with $a \leq b$, we have that $a \in B$.

(ii) The set B is called an *ideal* if it is directed and downward-closed. We write $\mathsf{Idl}(A, \leq)$ to denoted the set of all ideals of $(A, \leq)$.

(iii) The *ideal completion* of $(A, \leq)$, is the partially ordered set $(\mathsf{Idl}(A, \leq), \subseteq)$.

For terms, we already know that the set of (potentially infinite) terms can be constructed by forming the ideal completion of the partially ordered set $(\mathcal{T}(\Sigma_\bot), \leq_\bot)$ of finite terms.

Theorem 8.10 (ideal completion of terms, Berry and Lévy [12])**.** *The ideal completion of* $(\mathcal{T}(\Sigma_\bot), \leq_\bot)$ *is order isomorphic to* $(\mathcal{T}^\infty(\Sigma_\bot), \leq_\bot)$.

We show an analogous result for term graphs:

Theorem 8.11 (ideal completion of term graphs)**.** *The ideal completion of the partially ordered set* $(\mathcal{G}_\mathcal{C}(\Sigma_\bot), \leq_\bot^\mathsf{S})$ *is order isomorphic to* $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^\mathsf{S})$.

Proof. Let I be the set $\mathsf{Idl}(\mathcal{G}_\mathcal{C}(\Sigma_\bot), \leq_\bot^\mathsf{S})$ of ideals in $(\mathcal{G}_\mathcal{C}(\Sigma_\bot), \leq_\bot^\mathsf{S})$. To prove that $(I, \subseteq)$ and $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^\mathsf{S})$ are order isomorphic, we will construct two monotonic functions $\phi\colon\ \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot) \to I$ and $\psi\colon\ I \to \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$, and show that they are inverses of each other.

Define the function ϕ as follows: $\phi(g) = \left\{ h \in \mathcal{G}_\mathcal{C}(\Sigma_\bot) \,\middle|\, h \leq_\bot^\mathsf{S} g \right\}$ for all $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$. We have to show that $\phi(g)$ is indeed an ideal for each $g \in \mathcal{G}_\mathcal{C}(\Sigma_\bot)$. By definition, $\phi(g)$ is downward-closed. To show that it is directed, let $h_1, h_2 \in \phi(g)$, i.e. $h_1, h_2 \leq_\bot^\mathsf{S} g$. By Proposition 8.8, $\{h_1, h_2\}$ has a least upper bound h in $\mathcal{G}_\mathcal{C}(\Sigma_\bot)$. Since g is an upper bound of $\{h_1, h_2\}$, we have $h \leq_\bot^\mathsf{S} g$ and thus $h \in \phi(g)$.

Monotonicity of ϕ follows immediately from its definition.

Define the function ψ as follows: $\psi(G) = \bigsqcup G$ for all $G \in I$. Since, according to Theorem 4.4, $(\mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot), \leq_\bot^\mathsf{S})$ is a cpo, we know that ψ is well-defined. The monotonicity of ψ follows immediately from its definition.

Finally, we show that ϕ and ψ are inverses of each other. At first we show that $\psi(\phi(g)) = g$ for all $g \in \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$, i.e. $g = \bigsqcup \phi(g)$. By definition of ϕ, we already know that g is an upper bound of $\phi(g)$. To show that it is the least upper bound, we assume that $\overline{g} \in \mathcal{G}_\mathcal{C}^\infty(\Sigma_\bot)$ is an upper bound of $\phi(g)$ and show that $g \leq_\bot^\mathsf{S} \overline{g}$. We will do that by using Corollary 4.3.

(a) Let $\pi_1 \sim_g \pi_2$ and let $d = \max\{|\pi_1|, |\pi_2|\}$. Then, according to Corollary 5.8, also $\pi_1 \sim_{g\dagger d} \pi_2$. Moreover, by Proposition 8.2, $g\dagger d$ is finite and, by Corollary 5.9, $g\dagger d \leq_\bot^\mathsf{S} g$. Hence, since $g\dagger d \in \phi(g)$ and thus $g\dagger d \leq_\bot^\mathsf{S} \overline{g}$. This means that $\pi_1 \sim_{g\dagger d} \pi_2$ implies $\pi_1 \sim_{\overline{g}} \pi_2$, according to Corollary 4.3.

(b) Let $g(\pi) = f \in \Sigma$ and let $d = 1 + |\pi|$. Then, according to Corollary 5.8, also $g\dagger d(\pi) = f$. As for (a), we know that $g\dagger d \leq_\bot^\mathsf{S} \overline{g}$, which implies $\overline{g}(\pi) = f$, by Corollary 4.3.

Lastly, we show that $\phi(\psi(G)) = G$ for all $G \in I$. The inclusion $\phi(\psi(G)) \supseteq G$ is easy to prove: if $g \in G$, then $g \leq_\bot^\mathsf{S} \bigsqcup G$ and therefore $g \in \phi(\psi(G))$. For the converse inclusion assume that $h \in \phi(\psi(G))$, i.e. $h \in \mathcal{G}_\mathcal{C}(\Sigma_\bot)$ with $h \leq_\bot^\mathsf{S} \bigsqcup G$. We claim that there is some $\widehat{h} \in G$ with $h \leq_\bot^\mathsf{S} \widehat{h}$. Since G is downward-closed, this then implies $h \in G$. We conclude this proof by constructing a $\widehat{h} \in G$ with $h \leq_\bot^\mathsf{S} \widehat{h}$.

Let $\overline{g} = \bigsqcup G$. Since $h \leq_\bot^\mathsf{S} \overline{g}$, we have by Corollary 8.7 that $\pi \sim_h \pi'$ implies $\pi \sim_{\overline{g}} \pi'$ for all $\pi, \pi' \in \mathcal{P}^e(h)$. In turn, $\pi \sim_{\overline{g}} \pi'$ implies by Theorem 4.4, that there

is some $g \in G$ with $\pi \sim_g \pi'$. According to Proposition 8.5, the set $\mathcal{P}^e(h)$ is finite and thus there are only finitely many pairs $\pi, \pi' \in \mathcal{P}^e(h)$. Hence, we find a finite set $H \subseteq G$ such that for each $\pi, \pi' \in \mathcal{P}^e(h)$ with $\pi \sim_h \pi'$ there is a $g \in H$ with $\pi \sim_g \pi'$. Since H is a finite subset of the directed set G, there is some $h_1 \in G$ that is an upper bound of H. Consequently, for each $\pi, \pi' \in \mathcal{P}^e(h)$ with $\pi \sim_h \pi'$, we have $\pi \sim_{h_1} \pi'$ by Corollary 8.7.

By a similar argument we find some $h_2 \in G$ such that for each $\pi \in \mathcal{P}^e(h)$ with $h(\pi) = f \in \Sigma$, we have $h_2(\pi) = f$. Since G is directed, we find some $\widehat{h} \in G$ with $h_1, h_2 \leq^{\mathsf{S}}_{\perp} \widehat{h}$. Hence, by Corollary 8.7, for all $\pi, \pi' \in \mathcal{P}^e(h)$, we have that $\pi \sim_h \pi'$ implies $\pi \sim_{\widehat{h}} \pi'$ and that $h(\pi) = f \in \Sigma$ implies $\widehat{h}(\pi) = f$. According to Corollary 8.7, this means that $h \leq^{\mathsf{S}}_{\perp} \widehat{h}$. □

The above theorem show a certain completeness of the partial order $\leq^{\mathsf{S}}_{\perp}$ in the sense that it allows us to canonically construct the set of term graphs $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\perp})$ from the set of finite term graphs $\mathcal{G}_{\mathcal{C}}(\Sigma_{\perp})$. More concretely, an infinite term graph $g \in \mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma_{\perp})$ can be constructed by a limit construction involving only finite term graphs, viz. $g = \bigsqcup \left\{ h \in \mathcal{G}_{\mathcal{C}}(\Sigma_{\perp}) \,\middle|\, h \leq^{\mathsf{S}}_{\perp} g \right\}$. In fact, such a construction can also be achieved by the limit inferior of a sequence of finite graphs since we have that $g = \liminf_{d \to \omega} g{\dagger}d$.

Such a representation of infinite term graphs as a lub or a limit inferior of a sequence of finite term graphs is not possible for the rigid partial order $\leq^{\mathsf{R}}_{\perp}$. For example, there is no set of finite term graphs G whose lub is the term graph h_ω from Figure 5d w.r.t. the partial order $\leq^{\mathsf{R}}_{\perp}$. The reason is that no finite term graph g with $g \leq^{\mathsf{R}}_{\perp} h_\omega$ has a node labelled b at position $\langle 0 \rangle$.

8.3 Metric Completion

In this section, we shall show that the set $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$ of (potentially infinite) canonical term graphs can also be obtained as the metric completion of the metric space $(\mathcal{G}_{\mathcal{C}}(\Sigma), \mathbf{d}_{\dagger})$ of finite term graphs endowed with the simple metric $\mathbf{d}_{\dagger}$.

Analogous to the ideal completion of partially ordered sets, the metric completion extends a metric spaces to a complete metric space.

Definition 8.12. Let $(M, \mathbf{d})$ be a metric space. The *closure* of a subset $N \subseteq M$, denoted $\mathcal{C}l\,(N)$, is the set $\{x \in M \mid x \text{ is the limit of a sequence in } N\}$. A subset $N \subseteq M$ is called dense if $\mathcal{C}l\,(N) = M$. A complete metric space $(M^{\bullet}, \mathbf{d}^{\bullet})$ is called the *metric completion* of $(M, \mathbf{d})$ if there is an isometric embedding ϕ from $(M, \mathbf{d})$ into $(M^{\bullet}, \mathbf{d}^{\bullet})$ and if the image $Im(\phi)$ of ϕ is dense in $(M^{\bullet}, \mathbf{d}^{\bullet})$.

The metric completion of a metric space is unique up to isometry.

Again, for terms, we already know that we can construct the set of (potentially infinite) terms $\mathcal{T}^{\infty}(\Sigma)$ as the metric completion of the metric space $(\mathcal{T}(\Sigma), \mathbf{d})$ of finite terms.

Theorem 8.13 (metric completion of terms, Barr [11])**.** *The metric completion of* $(\mathcal{T}(\Sigma), \mathbf{d})$ *is the metric space* $(\mathcal{T}^{\infty}(\Sigma), \mathbf{d})$.

Analogously, we can show that the metric space $(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma), \mathbf{d}_{\dagger})$ of (potentially infinite) term graphs arises as the metric completion of the metric space $(\mathcal{G}_{\mathcal{C}}(\Sigma), \mathbf{d}_{\dagger})$ of finite term graphs.

Theorem 8.14 (metric completion of term graphs)**.** *The metric completion of* $(\mathcal{G}_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$ *is the metric space* $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$.

Proof. Since $\mathcal{G}_{\mathcal{C}}(\Sigma)$ is a subset of $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, we can define the isometric embedding $\phi\colon \mathcal{G}_{\mathcal{C}}(\Sigma) \to \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ by setting $\phi(g) = g$. It only remains to be shown that $Im(\phi) = \mathcal{G}_{\mathcal{C}}(\Sigma)$ is dense in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$. This is achieved by showing that for each $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ we find a sequence $(g_i)_{i<\omega}$ in $\mathcal{G}_{\mathcal{C}}(\Sigma)$ that converges to g. From its definition it is clear that the simple truncation is idempotent, i.e. $(g\dagger d)\dagger d = g\dagger d$. for all $d < \omega$. Hence, by Lemma 5.4, the sequence $(g\dagger d)_{d<\omega}$ converges to g in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$. Moreover, according to Proposition 8.2, $(g\dagger d)_{d<\omega}$ is a sequence in $\mathcal{G}_{\mathcal{C}}(\Sigma)$. □

The above theorem shows that the metric $\mathbf{d}_\dagger$ is complete in the sense that it allows us to construct the set of term graphs $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ from the set of finite term graphs $\mathcal{G}_{\mathcal{C}}(\Sigma)$ in a canonical way. More concretely, each term graph $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ can be constructed as the limit of a sequence of finite term graphs, viz. $g = \lim_{d\to\omega} g\dagger d$.

We cannot obtain such a completeness result for the rigid metric $\mathbf{d}_\ddagger$. For instance, consider the term graph h_ω from Figure 5d. For each $d > 1$, the rigid truncation $h_\omega\ddagger d$ of h_ω is equal to h_ω itself. Hence, there is no finite term graph g with a similarity $\mathsf{sim}_\ddagger(g, h_\omega) > 1$, which means, according to Lemma 5.4, that there is no sequence of finite term graphs that converges to h_ω in $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\ddagger)$.

9 Conclusions & Outlook

We have devised two independently defined but closely related infinitary calculi of term graph rewriting. Whilst this is not the first proposal for infinitary term graph rewriting calculi, we gave several arguments why the present approach is superior to our previous approach [8]: it is more natural, simpler and less restrictive. Due to the findings we have obtained here, we are very confident that we found two appropriate notions of convergence that generalise the corresponding notions of convergence on terms. Further evidence for that can be obtained by investigating strong notions of convergence that can be derived from the weak notions that we have studied here [9].

There is, however, one aspect of our notion of convergence that might be interpreted as an argument against its appropriateness. On term graphs, we do not obtain the correspondence between p- and m-convergence known from infinitary term rewriting; cf. Theorem 2.4. The underlying reason for the discrepancy is the fact that the partial order on term graphs $\leq^{\mathsf{S}}_\bot$ does not only capture the level of partiality – like $\leq_\bot$ does on terms – but also the degree of sharing. However, this discrepancy might just be a manifestation of the fundamental difference between terms and term graphs – namely sharing. And, in fact, when turning to strong convergence, we regain the correspondence between p- and m-convergence [9].

Unfortunately, we do not have solid soundness or completeness results apart from the preservation of convergence under unravelling and the metric/ideal completion construction of the set of term graphs. Even establishing soundness turns out to be difficult in the setting of weak convergence. Again the picture changes considerably once we move to strong convergence [9].

Acknowledgement

The author wishes to thank Clemens Grabmayer for his remarks on an earlier version of this paper.

Bibliography

[1] Z. Ariola and S. Blom. Skew and ω-Skew Confluence and Abstract Böhm Semantics. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838 of *Lecture Notes in Computer Science*, pages 368–403. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30911-6. doi: 10.1007/11601548_19.

[2] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117(1-3):95–168, 2002. ISSN 0168-0072. doi: 10.1016/S0168-0072(01)00104-X.

[3] Z. M. Ariola and J. W. Klop. Lambda Calculus with Explicit Recursion. *Information and Computation*, 139(2):154–233, 1997. ISSN 0890-5401. doi: 10.1006/inco.1997.2651.

[4] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[5] P. Bahr. Infinitary Rewriting - Theory and Applications. Master's thesis, Vienna University of Technology, Vienna, 2009.

[6] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[7] P. Bahr. Partial Order Infinitary Term Rewriting and Böhm Trees. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.67.

[8] P. Bahr. Modes of Convergence for Term Graph Rewriting. In M. Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 139–154, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.RTA.2011.139.

[9] P. Bahr. Infinitary Term Graph Rewriting is Simple, Sound and Complete. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69–84, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2012.69.

[10] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In P. C. T. de Bakker A. J. Nijman, editor, *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Berlin / Heidelberg, 1987. doi: 10.1007/3-540-17945-3_8.

[11] M. Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114(2):299–315, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90076-6.

[12] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 215–226, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512971.

[13] N. Dershowitz, S. Kaplan, and D. A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90040-9.

[14] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1): 68–95, 1977. ISSN 0004-5411. doi: 10.1145/321992.321997.

[15] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 95–103, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811543.

[16] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

[17] G. Kahn and G. D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2):187–277, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90090-G.

[18] J. L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955. ISBN 0387901256.

[19] R. Kennaway. On transfinite abstract reduction systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

[20] R. Kennaway and F.-J. de Vries. Infinitary Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 12, pages 668–711. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[21] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. ISSN 0164-0925. doi: 10.1145/177492.177577.

[22] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[23] S. Marlow. Haskell 2010 Language Report, 2010.

[24] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. ISBN 013453333X.

[25] R. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0201416638.

[26] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 3–61. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. ISBN 981-02-4020-1.

[27] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

Infinitary Term Graph Rewriting is Simple, Sound and Complete

Patrick Bahr

Department of Computer Science, University of Copenhagen

Abstract

Based on a simple metric and a simple partial order on term graphs, we develop two infinitary calculi of term graph rewriting. We show that, similarly to infinitary term rewriting, the partial order formalisation yields a conservative extension of the metric formalisation of the calculus. By showing that the resulting calculi simulate the corresponding well-established infinitary calculi of term rewriting in a sound and complete manner, we argue for the appropriateness of our approach to capture the notion of infinitary term graph rewriting.

Contents

Introduction

Term graph rewriting provides an efficient technique for implementing term rewriting by avoiding duplication of terms and instead relying on pointers in order to refer to a term several times [7]. Due to cycles, *finite* term graphs may represent *infinite* terms, and, correspondingly, *finite* term graph reductions may represent *transfinite* term reductions. Kennaway et al. [15] showed that finite term graph reductions simulate a restricted class of transfinite term reductions, called *rational reductions*, in a sound and complete manner via the unravelling mapping $\mathcal{U}(\cdot)$ from term graphs to terms. More precisely, given a term graph rewriting system $\mathcal{R}$ and a finite term graph g, we have for each finite term graph reduction $g \to^*_{\mathcal{R}} h$, a rational term reduction $\mathcal{U}(g) \twoheadrightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ (soundness), and conversely, for each rational term reduction $\mathcal{U}(g) \twoheadrightarrow_{\mathcal{U}(\mathcal{R})} t$, there is a term graph reduction $g \to^*_{\mathcal{R}} h$ and a rational term reduction $t \twoheadrightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ (completeness). Since term graph reduction steps may contract several term redexes simultaneously, the completeness result has to be formulated in this weaker form. Note, however, that this completeness property subsumes completeness of normalising reductions: for each rational reduction $\mathcal{U}(g) \twoheadrightarrow_{\mathcal{U}(\mathcal{R})} t$ to a normal form t, there is a reduction $g \to^*_{\mathcal{R}} h$ with $\mathcal{U}(h) = t$.

In this paper, we aim to resolve the asymmetry in the comparison of term rewriting and term graph rewriting by studying transfinite term graph reductions. To this end, we develop two infinitary calculi of term graph rewriting by generalising the notions of strong convergence on terms, based on a metric [14] resp. partial order [4], to term graphs. Instead of the complicated structures that we have used in our previous approach to weak convergence on term graphs [5], we adopt a rather simple and intuitive metric resp. partial order [6].

After summarising the basic theory of infinitary term rewriting (Section 1) and the fundamental concepts concerning term graphs (Section 2), we present a metric and a partial order on term graphs (Section 3). Based on these two structures, we define the notions of strong m-convergence resp. strong p-convergence and show that – akin to term rewriting – both coincide on total term graphs and that strong p-convergence is normalising (Section 4).

In Section 5, we present the main result of this paper: strongly p-converging term graph reductions are sound and complete w.r.t. strongly p-converging term reductions in the sense of Kennaway et al. [15] explained above.

This result comes with some surprise, though, as Kennaway et al. [15] argued that infinitary term graph rewriting cannot adequately simulate infinitary term rewriting. In particular, they present a counterexample for the completeness of an informally defined infinitary calculus of term graph rewriting. This counterexample indeed shows that strongly m-converging term graph reductions are not complete for strongly m-converging term reductions.

However, using the correspondence between strong p-convergence and m-convergence, we can derive soundness of the metric calculus from the soundness of the partial order calculus. Moreover, we prove that the metric calculus is still complete for *normalising* reductions. We thus argue that strong m-convergence, too, can be adequately simulated by term graph rewriting. In fact, in their original work on term graph rewriting [7], Barendregt et al. showed completeness only for normalising reductions in order to argue for the adequacy of acyclic finite term graph rewriting for simulating finite term rewriting.

We did not include all proofs in the main body of this paper. The missing proofs can be found in Appendix A.

1 Infinitary Term Rewriting

We assume familiarity with the basic theory of term rewriting [18], ordinal numbers, orders and topological spaces [13]. Below, we give an outline of infinitary term rewriting [4, 14].

We denote ordinal numbers by lower case Greek letters $\alpha, \beta, \gamma, \lambda, \iota$. A *sequence* S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We write $|S|$ for the length α of S. If α is a limit ordinal, S is called *open*; otherwise it is called *closed*. Given two sequences S, T, we write $S \cdot T$ to denote their concatenation and $S \leq T$ (resp. $S < T$) if S is a (proper) prefix of T. The prefix of T of length $\beta \leq |T|$ is denoted $T|_\beta$. For a set A, we write A^* to denote the set of finite sequences over A. For a finite sequence $(a_i)_{i<n} \in A^*$, we also write $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. In particular, $\langle\rangle$ denotes the empty sequence.

We consider the sets $\mathcal{T}^\infty(\Sigma)$ and $\mathcal{T}(\Sigma)$ of (possibly infinite) *terms* resp. *finite terms* over a signature Σ. Each symbol f has an associated arity $\mathsf{ar}(f)$, and we write $\Sigma^{(n)}$ for the set of symbols in Σ with arity n. For rewrite rules, we consider the signature $\Sigma_\mathcal{V} = \Sigma \uplus \mathcal{V}$ that extends the signature Σ with a set $\mathcal{V}$ of nullary variable symbols. For terms $s, t \in \mathcal{T}^\infty(\Sigma)$ and a position $\pi \in \mathcal{P}(t)$ in t, we write $t|_\pi$ for the *subterm* of t at π, $t(\pi)$ for the symbol in t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s.

A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}^\infty(\Sigma_\mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma_\mathcal{V})$ such that all variables occurring in r also occur in l. If the left-hand side of each rule in a TRS $\mathcal{R}$ is finite, then $\mathcal{R}$ is called *left-finite*. Every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$ as usual: $s \to_\mathcal{R} t$ iff there is a position $\pi \in \mathcal{P}(s)$, a rule $\rho\colon\ l \to r \in R$, and a substitution σ such that $s|_\pi = l\sigma$ and $t = s[r\sigma]_\pi$. We write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π. The subterm $s|_\pi$ is called a *redex* and is said to be *contracted* to $r\sigma$.

The metric $\mathbf{d}$ on $\mathcal{T}^\infty(\Sigma)$ that is used in the setting of infinitary term rewriting is defined by $\mathbf{d}(s,t) = 0$ if $s = t$ and $\mathbf{d}(s,t) = 2^{-k}$ if $s \neq t$, where k is the minimal depth at which s and t differ. The pair $(\mathcal{T}^\infty(\Sigma), \mathbf{d})$ is known to form a *complete ultrametric space* [2].

A *reduction* in a term rewriting system $\mathcal{R}$, is a sequence $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ of reduction steps in $\mathcal{R}$. The reduction S is called strongly *m-continuous* if $\lim_{\iota\to\lambda} t_\iota = t_\lambda$ and the depths of contracted redexes $(|\pi_\iota|)_{\iota<\lambda}$ tend to infinity, for each limit ordinal $\lambda < \alpha$. A reduction S is said to strongly m-converge to t,

written $S\colon t_0 \overset{m}{\twoheadrightarrow}_{\mathcal{R}} t$, if it is strongly m-continuous and either S is closed with $t = t_\alpha$ or S is open with $t = \lim_{\iota\to\alpha} t_\iota$ and the depths of contracted redexes $(|\pi_\iota|)_{\iota<\alpha}$ tend to infinity.

Example 1.1. Consider the rule $\rho\colon Y\,x \to x\,(Y\,x)$ defining the fixed point combinator Y in an applicative language. If we use an explicit function symbol @ instead of juxtaposition to denote application, ρ reads $@(Y,x) \to @(x,@(Y,x))$. Given a term t, we get the reduction

$$S\colon Y\,t \to_\rho t\,(Y\,t) \to_\rho t\,(t\,(Y\,t)) \to_\rho t\,(t\,(t\,(Y\,t))) \to_\rho \dots$$

which strongly m-converges to the infinite term $t\,(t\,(\dots))$.

As another example, consider the rule $\rho'\colon f(x) \to f(g(x))$ and its induced reduction

$$T\colon h(c,f(c)) \to_{\rho'} h(c,f(g(c))) \to_{\rho'} h(c,f(g(g(c)))) \to h(c,f(g(g(g(c))))) \to_{\rho'} \dots$$

Although the underlying sequence of terms converges in the metric space $(\mathcal{T}^\infty(\Sigma),\mathbf{d})$, viz. to the infinite term $h(c,f(g(g(\dots))))$, the reduction T does not strongly m-converges since the depth of the contracted redexes does not tend to infinity but instead stays at 1.

The partial order $\leq_\perp$ is defined on *partial terms*, i.e. terms over signature $\Sigma_\perp = \Sigma \uplus \{\perp\}$, with $\perp$ a nullary symbol. It is characterised as follows: $s \leq_\perp t$ iff t can be obtained from s by replacing each occurrence of $\perp$ by some partial term. The pair $(\mathcal{T}^\infty(\Sigma_\perp),\leq_\perp)$ forms a *complete semilattice* [12]. A partially ordered set $(A,\leq)$ is called a *complete partial order* (*cpo*) if it has a *least element* and every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$ in A. If, additionally, every *non-empty* subset B of A has a *greatest lower bound* (*glb*) $\bigsqcap B$, then $(A,\leq)$ is called a *complete semilattice*. This means that for complete semilattices the *limit inferior* $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha}\left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$ of a sequence $(a_\iota)_{\iota<\alpha}$ is always defined.

In the partial order approach to infinitary rewriting, convergence is defined by the limit inferior. Since we are considering strong convergence, the positions π_ι at which reductions take place are taken into consideration as well. In particular, we consider, for each reduction step $t_\iota \to_{\pi_\iota} t_{\iota+1}$ at position π_ι, the *reduction context* $c_\iota = t_\iota[\perp]_{\pi_\iota}$, i.e. the starting term with the redex at π_ι replaced by $\perp$. To indicate the reduction context c_ι of a reduction step, we also write $t_\iota \to_{c_\iota} t_{\iota+1}$. A reduction $S = (t_\iota \to_{c_\iota} t_{\iota+1})_{\iota<\alpha}$ is called *strongly p-continuous* if $\liminf_{\iota<\lambda} c_\iota = t_\lambda$ for each limit ordinal $\lambda<\alpha$. The reduction S is said to *strongly p-converge* to a term t, written $S\colon t_0 \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$, if it is strongly p-continuous and either S is closed with $t = t_\alpha$, or S is open with $\liminf_{\iota<\alpha} c_\iota = t$. If $S\colon t_0 \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ and t as well as all t_ι with $\iota<\alpha$ are total, i.e. contained in $\mathcal{T}^\infty(\Sigma)$, then we say that S strongly p-converges to t in $\mathcal{T}^\infty(\Sigma)$.

The distinguishing feature of the partial order approach is that, since the partial order on terms forms a complete semilattice, each continuous reduction also converges. It provides a conservative extension to strong m-convergence that allows rewriting modulo *meaningless terms* [4] by rewriting terms to $\perp$ if they are divergent according to the metric calculus.

Example 1.2. Reconsider S and T from Example 1.1. S has the same convergence behaviour in the partial order setting, viz. $S\colon\ Y\,t \overset{p}{\twoheadrightarrow} t\,(t\,(\dots))$. However, while the reduction T does not strongly m-converge, it does strongly p-converge, viz. $T\colon\ h(c, f(c)) \overset{p}{\twoheadrightarrow} h(c, \bot)$.

The relation between m- and p-convergence illustrated in the examples above is characteristic: strong p-convergence is a conservative extension of strong m-convergence.

Theorem 1.3 ([4])**.** *For every reduction S in a TRS the following equivalence holds:*

$$S\colon\ s \overset{m}{\twoheadrightarrow}_{\mathcal{R}} t \qquad \textit{iff} \qquad S\colon\ s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t \textit{ in } \mathcal{T}^{\infty}(\Sigma).$$

In the remainder of this paper, we shall develop a generalisation of both strong m- and p-convergence to term graphs that maintains the above correspondence, and additionally simulates term reductions in a sound and complete way.

2 Graphs and Term Graphs

The notion of term graphs that we employ in this paper is taken from Barendregt et al. [7].

Definition 2.1 (graphs)**.** Let Σ be a signature. A *graph* over Σ is a tuple $g = (N, \mathsf{lab}, \mathsf{suc})$ consisting of a set N (of *nodes*), a *labelling function* $\mathsf{lab}\colon\ N \to \Sigma$, and a *successor function* $\mathsf{suc}\colon\ N \to N^*$ such that $|\mathsf{suc}(n)| = \mathsf{ar}(\mathsf{lab}(n))$ for each node $n \in N$, i.e. a node labelled with a k-ary symbol has precisely k successors. If $\mathsf{suc}(n) = \langle n_0, \dots, n_{k-1}\rangle$, then we write $\mathsf{suc}_i(n)$ for n_i. Moreover, we use the abbreviation $\mathsf{ar}_g(n)$ for the arity $\mathsf{ar}(\mathsf{lab}(n))$ of n in g.

Definition 2.2 (paths, reachability)**.** Let $g = (N, \mathsf{lab}, \mathsf{suc})$ be a graph and $n, m \in N$. A *path* in g from n to m is a finite sequence $\pi \in \mathbb{N}^*$ such that either π is empty and $n = m$, or $\pi = \langle i\rangle \cdot \pi'$ with $0 \le i < \mathsf{ar}_g(n)$ and the suffix π' is a path in g from $\mathsf{suc}_i(n)$ to m. If there exists a path from n to m in g, we say that m is *reachable* from n in g.

Definition 2.3 (term graphs)**.** Given a signature Σ, a *term graph* g over Σ is a quadruple $(N, \mathsf{lab}, \mathsf{suc}, r)$ consisting of an *underlying* graph $(N, \mathsf{lab}, \mathsf{suc})$ over Σ whose nodes are all reachable from the *root node* $r \in N$. The class of all term graphs over Σ is denoted $\mathcal{G}^{\infty}(\Sigma)$. We use the notation N^g, lab^g, suc^g and r^g to refer to the respective components N,lab, suc and r of g. Given a graph or a term graph h and a node n in h, we write $h|_n$ to denote the *sub-term graph* of h rooted in n, which consists of all nodes reachable from n in h.

Paths in a graph are not absolute but relative to a starting node. In term graphs, however, we have a distinguished root node from which each node is reachable. Paths relative to the root node are central for dealing with term graphs modulo isomorphism:

Definition 2.4 (positions, depth, trees)**.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $n \in N^g$. A *position* of n in g is a path in the underlying graph of g from r^g to n. The set of all positions in g is denoted $\mathcal{P}(g)$; the set of all positions of n in g is denoted $\mathcal{P}_g(n)$. A position $\pi \in \mathcal{P}_g(n)$ is called *minimal* if no proper prefix $\pi' < \pi$ is in $\mathcal{P}_g(n)$. The set of all minimal positions of n in g is denoted $\mathcal{P}^m_g(n)$. The *depth* of n in g, denoted $\mathsf{depth}_g(n)$, is the minimum of the lengths of the positions of n in g. For a position $\pi \in \mathcal{P}(g)$, we write $\mathsf{node}_g(\pi)$ for the unique node $n \in N^g$ with $\pi \in \mathcal{P}_g(n)$, $g(\pi)$ for its symbol $\mathsf{lab}^g(n)$, and $g|_\pi$ for the sub-term graph $g|_n$. The term graph g is called a *term tree* if each node in g has exactly one position.

Note that the labelling function of graphs – and thus term graphs – is *total*. In contrast, Barendregt et al. [7] considered *open* (term) graphs with a *partial* labelling function such that unlabelled nodes denote holes or variables. This partiality is reflected in their notion of homomorphisms in which the homomorphism condition is suspended for unlabelled nodes.

Instead of a partial node labelling function, we chose a *syntactic* approach that is more flexible and closer to the representation in terms. Variables, holes and "bottoms" are labelled by a distinguished set of constant symbols and the notion of homomorphisms is parametrised by a set of constant symbols Δ for which the homomorphism condition is suspended:

Definition 2.5 (Δ-homomorphisms)**.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$, and $g, h \in \mathcal{G}^\infty(\Sigma)$. A function $\phi\colon N^g \to N^h$ is called *homomorphic* in $n \in N^g$ if the following holds:

$$\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n)) \qquad \text{(labelling)}$$

$$\phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^h_i(\phi(n)) \quad \text{for all } 0 \leq i < \mathsf{ar}_g(n) \qquad \text{(successor)}$$

A Δ-*homomorphism* ϕ from g to h, denoted $\phi\colon g \to_\Delta h$, is a function $\phi\colon N^g \to N^h$ that is homomorphic in n for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$ and satisfies $\phi(r^g) = r^h$.

Note that, in contrast to Barendregt et al. [7], we require that root nodes are mapped to root nodes. This additional requirement makes our generalised notion of homomorphisms more akin to that of Barendsen [8]: for $\Delta = \emptyset$, we obtain his notion of homomorphisms.

Nodes labelled with a symbol from Δ can be thought of as holes in the term graphs, which can be filled with other term graphs. For example, if we have a distinguished set of variable symbols $\mathcal{V} \subseteq \Sigma^{(0)}$, we can use $\mathcal{V}$-homomorphisms to formalise the matching of a term graph against a term graph rule, which requires the instantiation of variables.

Note that Δ-homomorphisms are unique [5], i.e. there is at most one Δ-homomorphism from one term graph to another. Consequently, whenever there are two Δ-homomorphisms $\phi\colon g \to_\Delta h$ and $\psi\colon h \to_\Delta g$, they are inverses of each other, i.e. Δ-*isomorphisms*. If two term graphs are Δ-*isomorphic*, we write $g \cong_\Delta h$.

For the two special cases $\Delta = \emptyset$ and $\Delta = \{\sigma\}$, we write $\phi\colon g \to h$ resp. $\phi\colon g \to_\sigma h$ instead of $\phi\colon g \to_\Delta h$ and call ϕ a homomorphism resp. a σ-homomorphism. The same convention applies to Δ-isomorphisms.

Since we are studying modes of convergence over term graphs, we want to reason modulo isomorphism. The following notion of canonical term graphs will allow us to do that:

Definition 2.6 (canonical term graphs)**.** A term graph g is called *canonical* if $n = \mathcal{P}_g(n)$ for each $n \in N^g$. The set of all canonical term graphs over Σ is denoted $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$.

For each term graph g, we can give a unique canonical term graph $\mathcal{C}(g)$ isomorphic to g:

$$\begin{aligned} N^{\mathcal{C}(g)} &= \{\mathcal{P}_g(n) \mid n \in N\} \\ r^{\mathcal{C}(g)} &= \mathcal{P}_g(r) \\ \mathsf{lab}^{\mathcal{C}(g)}(\mathcal{P}_g(n)) &= \mathsf{lab}(n) \\ \mathsf{suc}_i^{\mathcal{C}(g)}(\mathcal{P}_g(n)) &= \mathcal{P}_g(\mathsf{suc}_i(n)) \quad \text{for all } n \in N, 0 \leq i < \mathsf{ar}_g(n) \end{aligned}$$

As we have shown previously [5], this indeed yields a canonical representation of term graphs, viz. $g \cong h$ iff $\mathcal{C}(g) = \mathcal{C}(h)$ for all term graphs g, h.

Note that the set of nodes $N^{\mathcal{C}(g)}$ above forms a partition of the set of positions in g. We write $\sim_g$ for the equivalence relation on $\mathcal{P}(g)$ that is induced by this partition. That is, $\pi_1 \sim_g \pi_2$ iff $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. The structure of a term graph g is uniquely determined by its set of positions $\mathcal{P}(g)$, the labelling $g(\cdot)\colon \pi \mapsto g(\pi)$, and the equivalence $\sim_g$. We will call such a triple $(\mathcal{P}(g), g(\cdot), \sim_g)$ a *labelled quotient tree*. Labelled quotient trees uniquely represent term graphs up to isomorphism. In other words: labelled quotient trees uniquely represent canonical term graphs. For a more axiomatic treatment of labelled quotient tree that studies these relationships, we refer to our previous work [5].

We can characterise Δ-homomorphisms in terms of labelled quotient trees:

Lemma 2.7 ([5])**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, there is a $\phi\colon g \to_\Delta h$ iff the following holds for all $\pi, \pi' \in \mathcal{P}(g)$:*

$$\textit{(a) } \pi \sim_g \pi' \quad \Longrightarrow \quad \pi \sim_h \pi', \textit{ and} \quad \textit{(b) } g(\pi) = h(\pi) \quad \textit{whenever} \quad g(\pi) \notin \Delta.$$

Intuitively, (a) means that h has at least as much sharing of nodes as g has, whereas (b) means that h has at least the same non-Δ-symbols as g.

Given a term tree g, the equivalence $\sim_g$ is the identity relation $\mathcal{I}_{\mathcal{P}(g)}$ on $\mathcal{P}(g)$, i.e. $\pi_1 \sim_g \pi_2$ iff $\pi_1 = \pi_2$. There is an obvious one-to-one correspondence between canonical term *trees* and terms: a term $t \in \mathcal{T}^\infty(\Sigma)$ corresponds to the canonical term tree given by the labelled quotient tree $(\mathcal{P}(t), t(\cdot), \mathcal{I}_{\mathcal{P}(t)})$. We thus consider the set of terms $\mathcal{T}^\infty(\Sigma)$ as the subset of term trees in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$.

With this correspondence in mind, we define the *unravelling* of a term graph g, denoted $\mathcal{U}(g)$, as the unique term t such that there is a homomorphism $\phi\colon t \to g$.

Example 2.8. Consider the term graphs g_2 and h_0 illustrated in Figure 1. The unravelling of g_2 is the term $@(f, @(f, @(Y, f)))$ whereas the unravelling of the cyclic term graph h_0 is the infinite term $@(f, @(f, \dots))$.

3 Two Simple Modes of Convergence for Term Graphs

In a previous attempt to generalise the modes of convergence of term rewriting to term graphs, we developed a metric and a partial order on term graphs that were both rather complicated [5]. While the resulting notions of weak convergence have a correspondence similar to that for terms (cf. Theorem 1.3), they are also limited as we explain below. In this paper, we shall use a much simpler and more intuitive approach that we recently developed [6], and which we summarise briefly below.

Like for terms, we move to a signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$ to define a partial order on term graphs. Term graphs over signature $\Sigma_\bot$ are also referred to as *partial* whereas term graphs over Σ are referred to as *total*. In order to generalise the partial order $\leq_\bot$ on terms to term graphs, we make use of the observation that $\bot$-homomorphisms characterise the partial order $\leq_\bot$: given two terms $s, t \in \mathcal{T}^\infty(\Sigma_\bot)$, we have $s \leq_\bot t$ iff there is a $\bot$-homomorphism $\phi\colon s \to_\bot t$. In our previous work, we have used a restricted form of $\bot$-homomorphisms in order to define a partial order on term graphs [5]. In this paper, however, we simply take $\bot$-homomorphism as the definition of the partial order on term graphs. The *simple partial order* $\leq_\bot^{\mathsf{S}}$ on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$ is defined as follows: $g \leq_\bot^{\mathsf{S}} h$ iff there is a $\bot$-homomorphism $\phi\colon s \to_\bot t$. Hence, we get the following characterisation, according to Lemma 2.7:

Corollary 3.1. *Let $g, h \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$. Then $g \leq_\bot^{\mathsf{S}} h$ iff, for all $\pi, \pi' \in \mathcal{P}(g)$, we have*

$$\text{(a) } \pi \sim_g \pi' \quad \Longrightarrow \quad \pi \sim_h \pi' \qquad \text{(b) } g(\pi) = h(\pi) \quad \text{if } g(\pi) \in \Sigma.$$

With this partial order on term graphs, we indeed get a complete semilattice:

Theorem 3.2 ([6])**.** *The pair $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq_\bot^{\mathsf{S}})$ forms a complete semilattice. In particular, the limit inferior of a sequence $(g_\iota)_{\iota<\alpha}$ is given by the labelled quotient tree $(P, \sim, l)$:*

$$\begin{aligned} P &= \textstyle\bigcup_{\beta<\alpha} \{\pi \in \mathcal{P}(g_\beta) \mid \forall \pi' < \pi \forall \beta \leq \iota < \alpha\colon g_\iota(\pi') = g_\beta(\pi')\} \\ \sim &= (P \times P) \cap \textstyle\bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota} \\ l(\pi) &= \begin{cases} g_\beta(\pi) & \text{if } \exists \beta < \alpha \forall \beta \leq \iota < \alpha\colon g_\iota(\pi) = g_\beta(\pi) \\ \bot & \text{otherwise} \end{cases} \qquad \text{for all } \pi \in P \end{aligned}$$

In order to generalise the metric $\mathbf{d}$ on terms to term graphs, we need to formalise what it means for two term graphs to be "equal" up to a certain depth. To this end, we define for each term graph $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \in \mathbb{N}$ the *simple truncation* $g{\dagger}d$ as the term graph obtained from g by relabelling each node at depth d with $\bot$ and (thus) removing all nodes at depth greater than d. The distance $\mathbf{d}_\dagger(g, h)$ between two term graphs $g, h \in \mathcal{G}^\infty(\Sigma)$ is then defined as 0 if $g \cong h$ and otherwise as 2^{-d} with d the greatest $d \in \mathbb{N}$ with $g{\dagger}d \cong h{\dagger}d$. This definition indeed yields a complete ultrametric space:

Theorem 3.3 ([6]). *The pair $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma), \mathbf{d}_\dagger)$ forms a complete ultrametric space. In particular, the limit of each Cauchy sequence $(g_\iota)_{\iota<\alpha}$ is given by the labelled quotient tree $(P, l, \sim)$:*

$$P = \liminf_{\iota\to\alpha} \mathcal{P}(g_\iota) = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \mathcal{P}(g_\iota) \qquad \sim = \liminf_{\iota\to\alpha} \sim_{g_\iota} = \bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota}$$

$$l(\pi) = g_\beta(\pi) \quad \textit{for some } \beta < \alpha \textit{ with } g_\iota(\pi) = g_\beta(\pi) \textit{ for each } \beta \leq \iota < \alpha$$

The metric space that we have previously studied [5] was similarly defined in terms of a truncation. However, we used a much more complicated notion of truncation that would retain certain nodes of depth greater than d.

Similarly to the corresponding modes of convergence on terms, we have the equality $\lim_{\iota\to\alpha} g_\iota = \liminf_{\iota\to\alpha} g_\iota$ for any sequence of total term graphs $(g_\iota)_{\iota<\alpha}$ that converges in the metric space $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \mathbf{d}_\dagger)$. However, unlike in the setting of terms, the converse is not true! That is, if $\liminf_{\iota\to\alpha} g_\iota$ is a total term graph, then it is not necessarily equal to $\lim_{\iota\to\alpha} g_\iota$ – in fact, $(g_\iota)_{\iota<\alpha}$ might not even converge at all. As a consequence, we are not able to obtain a correspondence in the vein of Theorem 1.3 for weak convergence. In the next section, we will show that we do, however, obtain such a correspondence for strong convergence.

Note that the more restrictive partial order and metric space that we have studied in our previous work [5] does yield the above described correspondence for weak convergence. However, this result comes at the expense of generality and intuition: the convergence behaviour illustrated in Figure 1c, which is intuitively expected and also captured by the partial order $\leq^{\mathsf{S}}_\bot$ and the metric $\mathbf{d}_\uparrow$, is not possible in these more restrictive structures [6].

4 Infinitary Term Graph Rewriting

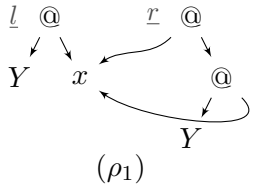
In this paper, we adopt the term graph rewriting framework of Barendregt et al. [7]. In order to represent placeholders in rewrite rules, this framework uses variables – in a manner much similar to term rewrite rules. To this end, we consider a signature $\Sigma_\mathcal{V} = \Sigma \uplus \mathcal{V}$ that extends the signature Σ with a set $\mathcal{V}$ of nullary variable symbols.

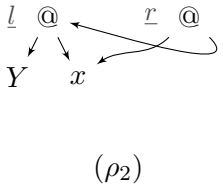
Definition 4.1 (term graph rewriting systems).

(i) Given a signature Σ, a *term graph rule* ρ over Σ is a triple (g, l, r) where g is a graph over $\Sigma_\mathcal{V}$ and $l, r \in N^g$ such that all nodes in g are reachable from l or r. We write ρ_l resp. ρ_r to denote the left- resp. right-hand side of ρ, i.e. the term graph $g|_l$ resp. $g|_r$. Additionally, we require that for each variable $v \in \mathcal{V}$ there is at most one node n in g labelled v and that n is different but still reachable from l.

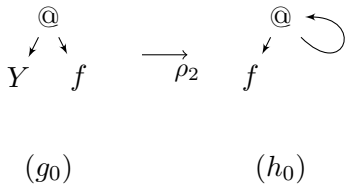
(ii) A *term graph rewriting system (GRS)* $\mathcal{R}$ is a pair (Σ, R) with Σ a signature and R a set of term graph rules over Σ.

The notion of unravelling straightforwardly extends to term graph rules: let ρ be a term graph rule with ρ_l and ρ_r its left- resp. right-hand side term graph. The *unravelling* of ρ, denoted $\mathcal{U}(\rho)$ is the term rule $\mathcal{U}(\rho_l) \to \mathcal{U}(\rho_r)$. The unravelling of a GRS $\mathcal{R} = (\Sigma, R)$, denoted $\mathcal{U}(\mathcal{R})$, is the TRS $(\Sigma, \{\mathcal{U}(\rho) \mid \rho \in R\})$.

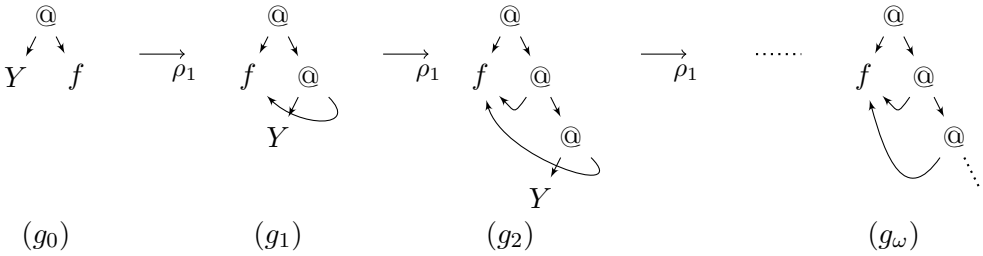




(a) Term graph rules that unravel to $Y\,x \to x\,(Y\,x)$.



(b) A single ρ_2-step.



(c) A strongly m-convergent term graph reduction over ρ_1.

Figure 1: Implementation of the fixed point combinator as a term graph rewrite rule.

Example 4.2. Figure 1a shows two term graph rules which both unravel to the term rule $\rho\colon\ Y\,x \to x\,(Y\,x)$ from Example 1.1. Note that sharing of nodes is used both to refer to variables in the left-hand side from the right-hand side, and in order to simulate duplication.

Without going into all details of the construction, we describe the application of a rewrite rule ρ with root nodes l and r to a term graph g in four steps: at first a suitable sub-term graph of g rooted in some node n of g is *matched* against the left-hand side of ρ. This matching amounts to finding a $\mathcal{V}$-homomorphism ϕ from the left-hand side ρ_l to the sub-term graph in g rooted in n, the *redex*. The $\mathcal{V}$-homomorphism ϕ allows us to instantiate variables in the rule with sub-term graphs of the redex. In the second step, nodes and edges in ρ that are not in ρ_l are copied into g, such that each edge pointing to a node m in ρ_l is redirected to $\phi(m)$. In the next step, all edges pointing to the root n of the redex are redirected to the root n' of the *contractum*, which is either r or $\phi(r)$, depending on whether r has been copied into g or not (because it is reachable from l in ρ). Finally, all nodes not reachable from the root of (the now modified version of) g are removed.

With h the result of the above construction, this induces a *pre-reduction step* $\psi\colon\ g \mapsto_{n,\rho,n'} h$ from g to h. In order to indicate the underlying GRS $\mathcal{R}$, we also write $\psi\colon\ g \mapsto_{\mathcal{R}} h$.

The definition of term graph rewriting in the form of pre-reduction steps is very operational in style. The result of applying a rewrite rule to a term graph is constructed in several steps by manipulating nodes and edges explicitly. While this is beneficial for implementing a rewriting system, it is problematic for reasoning on term graphs modulo isomorphism, which is necessary for introducing notions of convergence. In our case, however, this does not cause any harm since the construction of the result term graph of a pre-reduction step is invariant under

isomorphism. This observation justifies the following definition of reduction steps:

Definition 4.3. Let $\mathcal{R} = (\Sigma, R)$ be GRS, $\rho \in R$ and $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ with $n \in N^g$ and $m \in N^h$. A tuple $\phi = (g, n, \rho, m, h)$ is called a *reduction step*, written $\phi\colon\ g \to_{n,\rho,m} h$, if there is a pre-reduction step $\phi'\colon\ g' \mapsto_{n',\rho,m'} h'$ with $\mathcal{C}(g') = g$, $\mathcal{C}(h') = h$, $n = \mathcal{P}_{g'}(n')$, and $m = \mathcal{P}_{h'}(m')$. Similarly to pre-reduction steps, we write $\phi\colon\ g \to_\mathcal{R} h$ or $\phi\colon\ g \to h$ for short.

In other words, a reduction step is a canonicalised pre-reduction step. Figures 1b and 1c show various (pre-)reduction steps derived from the rules in Figure 1a.

4.1 Reduction Contexts

The idea of strong convergence is to conservatively approximate the convergence behaviour somewhat independently from the actual rules that are applied. Strong m-convergence in TRSs requires that the depth of the redexes tends to infinity thereby assuming that anything at the depth of the redex or below is potentially affected by a reduction step. Strong p-convergence, on the other hand, uses a better approximation that only assumes that the redex is affected by a reduction step – not however other subterms at the same depth. To this end strong p-convergence uses a notion of reduction contexts – essentially the term minus the redex – for the formation of limits. In this section, we shall devise a corresponding notion of reduction contexts on term graphs and argue for its adequacy for formalising strong p-convergence. The following definition provides the basic construction that we shall use:

Definition 4.4. Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $n \in N^g$. The *local truncation* of g at n, denoted $g\backslash n$, is obtained from g by labelling n with $\bot$ and removing all outgoing edges from n as well as all nodes that thus become unreachable from the root.

Lemma 4.5. *For each $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $n \in N^g$, the local truncation $g\backslash n$ has the following labelled quotient tree $(P, l, \sim)$:*

$$P = \{\pi \in \mathcal{P}(g) \mid \forall \pi' < \pi\colon\ \pi' \notin \mathcal{P}_g(n)\}$$
$$\sim\ = \sim_g \cap\ P \times P$$
$$l(\pi) = \begin{cases} g(\pi) & \text{if } \pi \notin \mathcal{P}_g(n) \\ \bot & \text{if } \pi \in \mathcal{P}_g(n) \end{cases} \quad \text{for all } \pi \in P$$

As a corollary of Lemma 4.5 and Corollary 3.1 we obtain the following:

Corollary 4.6. *For each $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $n \in N^g$, we have $g\backslash n \leq^\mathsf{S}_\bot g$.*

It is also possible – although cumbersome – to show that, given a reduction step $g \to_n h$ at node n, the local truncation $g\backslash n$ is isomorphic to the term graph that is obtained from h by essentially relabelling the positions $\mathcal{P}_g(n)$ occurring in h with $\bot$. For this term graph, denoted $h\backslash[\mathcal{P}_g(n)]$, we then also have $h\backslash[\mathcal{P}_g(n)] \leq^\mathsf{S}_\bot h$. By combining this with Corollary 4.6, we eventually obtain the following fundamental property of reduction contexts:

Proposition 4.7. *Given a reduction step $g \to_n h$, we have $g\backslash n \leq^{\mathsf{S}}_{\bot} g, h$.*

This means that the local truncation at the root of the redex is preserved by reduction steps and is therefore an adequate notion of reduction context for strong p-convergence [3].

4.2 Strong Convergence

Now that we have an adequate notion of reduction contexts, we define strong p-convergence on term graphs analogously to strong p-convergence on terms. For strong m-convergence, we simply take the same notion of depth that we already used for the definition of the simple truncation $g\dagger d$ and thus the simple metric $\mathbf{d}_\dagger$.

Definition 4.8. Let $\mathcal{R} = (\Sigma, R)$ be a GRS.

(i) The *reduction context* c of a graph reduction step $\phi\colon\ g \to_n h$ is the term graph $\mathcal{C}(g\backslash n)$. We write $\phi\colon\ g \to_c h$ to indicate the reduction context of a graph reduction step.

(ii) Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}$. S is *strongly m-continuous* in $\mathcal{R}$ if $\lim_{\iota\to\lambda} g_\iota = g_\lambda$ and $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\lambda}$ tends to infinity for each limit ordinal $\lambda < \alpha$. S *strongly m-converges* to g in $\mathcal{R}$, denoted $S\colon\ g_0 \xrightarrow{m}_{\mathcal{R}} g$, if it is strongly m-continuous and either S is closed with $g = g_\alpha$ or S is open with $g = \lim_{\iota\to\alpha} g_\iota$ and $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tending to infinity.

(iii) Let $S = (g_\iota \to_{c_\iota} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}_\bot = (\Sigma_\bot, R)$. S is *strongly p-continuous* in $\mathcal{R}$ if $\liminf_{\iota\to\lambda} c_\iota = g_\lambda$ for each limit ordinal $\lambda < \alpha$. S *strongly p-converges* to g in $\mathcal{R}$, denoted $S\colon\ g_0 \xrightarrow{p}_{\mathcal{R}} g$, if it is strongly p-continuous and either S is closed with $g = g_\alpha$ or S is open with $g = \liminf_{\iota\to\alpha} c_\iota$.

Note that we have to extend the signature of $\mathcal{R}$ to $\Sigma_\bot$ for the definition of strong p-convergence. However, we can obtain the total fragment of strong p-convergence if we restrict ourselves to total term graphs: a reduction $(g_\iota \to_{\mathcal{R}_\bot} g_{\iota+1})_{\iota<\alpha}$ strongly p-converging to g is called strongly p-converging to g in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ if g as well as each g_ι is total, i.e. an element of $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$.

Example 4.9. Figure 1c illustrates an infinite reduction derived from the rule ρ_1 in Figure 1a. Note that the reduction rule is applied to sub-term graphs at increasingly large depth. Since additionally, $g_i\dagger(i+1) \cong g_\omega\dagger(i+1)$ for all $i < \omega$, i.e. $\lim_{i\to\omega} g_i = g_\omega$, the reduction strongly m-converges to the term graph g_ω. Moreover, since each node in g_ω eventually appears in a reduction context and remains stable afterwards, we have $\liminf_{i\to\omega} g_\iota = g_\omega$. Consequently, the reduction also strongly p-converges to g_ω.

The rest of this section is concerned with proving that the above correspondence in convergence behaviour – similarly to infinitary term rewriting (cf. Theorem 1.3) – is characteristic: strong p-convergence in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ coincides with strong m-convergence.

Since the partial order $\leq^{\mathsf{S}}_{\bot}$ forms a complete semilattice on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot)$ according to Theorem 3.2, we know that strong p-continuity coincides with strong p-convergence:

Proposition 4.10. *Each strongly p-continuous reduction in a GRS is strongly p-convergent.*

The two lemmas below form the central properties that link strong m- and p-convergence:

Lemma 4.11. *Let $(g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open reduction in a GRS $\mathcal{R}_\bot$. If S strongly p-converges to a total term graph, then $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity.*

Lemma 4.12. *Let $(g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open reduction strongly p-converging to g in a GRS $\mathcal{R}_\bot$. If $(g_\iota)_{\iota<\alpha}$ is Cauchy and $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity, then $g \cong \lim_{\iota\to\alpha} g_\iota$.*

The following property, which relates strong m-convergence and -continuity, follows from the fact that our definition of strong m-convergence on term graphs instantiates the abstract notion of strong m-convergence developed in our previous work [3]:

Lemma 4.13. *Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open strongly m-continuous reduction in a GRS. If $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity, then S is strongly m-convergent.*

Proof. Special case of Proposition 5.5 from [3]; cf. [9, Thm. B.2.5] for the correct proof. □

Now we have everything in place to prove that strong p-convergence conservatively extends strong m-convergence.

Theorem 4.14. *Let $\mathcal{R}$ be a GRS and S a reduction in $\mathcal{R}_\bot$. We then have that*

$$S\colon g \xrightarrow{m}_{\mathcal{R}} h \qquad \textit{iff} \qquad S\colon g \xrightarrow{p}_{\mathcal{R}} h \textit{ in } \mathcal{G}^\infty_{\mathcal{C}}(\Sigma).$$

Proof. Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{R}_\bot$. We prove the "only if" direction by induction on α. The case $\alpha = 0$ is trivial. If α is a successor ordinal, then the statement follows immediately from the induction hypothesis.

Let α be a limit ordinal. As $S\colon g \xrightarrow{m}_{\mathcal{R}} g_\alpha$, we know that $S|_\gamma\colon g \xrightarrow{m}_{\mathcal{R}} g_\gamma$ for all $\gamma < \alpha$. Hence, we can apply the induction hypothesis to obtain that $S|_\gamma\colon g \xrightarrow{p}_{\mathcal{R}} g_\gamma$ for each $\gamma < \alpha$. Thus, S is strongly p-continuous, which means, by Proposition 4.10, that S strongly p-converges to some term graph h'. As S strongly m-converges, we know that $(g_\iota)_{\iota<\alpha}$ is Cauchy and that $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity. Hence, we can apply Lemma 4.12 to obtain that $h' = \lim_{\iota\to\alpha} g_\iota = h$, i.e. $S\colon g \xrightarrow{p}_{\mathcal{R}} h$. The "in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$" part follows from $S\colon g \xrightarrow{m}_{\mathcal{R}} h$.

We will also prove the "if" direction by induction on α: again, the case $\alpha = 0$ is trivial and the case that α is a successor ordinal follows immediately from the induction hypothesis.

Let α be a limit ordinal. As S is strongly p-convergent in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, we know that $S|_\gamma\colon g \xrightarrow{p}_{\mathcal{R}} g_\gamma$ in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ for all $\gamma < \alpha$. Thus, we can apply the induction hypothesis to obtain that $S|_\gamma\colon g \xrightarrow{m}_{\mathcal{R}} g_\gamma$ for each $\gamma < \alpha$. Hence, S is strongly m-continuous. As S strongly p-converges in $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, we know from Lemma 4.11 that $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity. With the strong m-continuity of S, this yields, according to Lemma 4.13, that S strongly m-converges to some h'. By Lemma 4.12, we conclude that $h' = h$, i.e. $S\colon g \xrightarrow{m}_{\mathcal{R}} h$. □

4.3 Normalisation of Strong p-convergence

In this section we shall show that – similarly to TRSs [4] – GRSs are normalising w.r.t. strong p-convergence. As for terms, this is a distinguishing feature of strong p-convergence. For example, the term graph rule (that unravels to) $c \to c$, for some constant c, yields a system in which c has no normal form w.r.t. strong m-convergence (or finite reduction or weak p-/m-convergence). If we consider strong p-convergence however, repeatedly applying the rule to c yields the normalising reduction $c \xrightarrow{p}\!\!\!\!\twoheadrightarrow \bot$. Term graphs which can be infinitely often contracted at the root – such as c – are called *root-active*:

Definition 4.15. Let $\mathcal{R}$ be a GRS over Σ and $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$. Then g is called *root-active* if, for each reduction $g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} g'$, there is a reduction $g' \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} h$ to a redex h in $\mathcal{R}$. The term graph g is called *root-stable* if, for each reduction $g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} h$, h is not a redex in $\mathcal{R}$.

Similar to the construction of Böhm normal forms [17], the strategy for rewriting a term graph into normal form is to rewrite root-active sub-term graphs to $\bot$ and non-root-active sub-term graphs to root-stable terms. The following lemma will allow us to do that:

Lemma 4.16. *Let $\mathcal{R}$ be a GRS over Σ and $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$.*

(i) If g is root-active, then there is a reduction $g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} \bot$.

(ii) If g is not root-active, then there is a reduction $g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} h$ to a root-stable term graph h.

(iii) If g is root-stable, then so is every term graph h with a reduction $g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} h$.

In the following, we need to generalise the concatenation of sequences. To this end, we make use of the fact that the prefix order $\leq$ on sequences forms a cpo and thus has lubs for directed sets: let $(S_\iota)_{\iota<\alpha}$ be a sequence of sequences in a common set. The concatenation of $(S_\iota)_{\iota<\alpha}$, written $\prod_{\iota<\alpha} S_\iota$, is recursively defined as the empty sequence $\langle\rangle$ if $\alpha = 0$, $(\prod_{\iota<\alpha'} S_\iota) \cdot S_{\alpha'}$ if $\alpha = \alpha' + 1$, and $\bigsqcup_{\gamma<\alpha} \prod_{\iota<\gamma} S_\iota$ if α is a limit ordinal.

The following lemma shows that we can use the reductions from Lemma 4.16 in order to turn the sub-term graphs of a term graph into root-stable form level by level:

Lemma 4.17. *Let $\mathcal{R}$ be a GRS over Σ, $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$ and $d < \omega$ such that $g|_n$ is root-stable for all $n \in N^g$ with $\mathsf{depth}_g(n) < d$. Then there is a reduction $S_d\colon g \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} h$ such that $h|_n$ is root-stable for each $n \in N^g$ with $\mathsf{depth}_g(n) \leq d$.*

Proof. There are only finitely many nodes in g at depth d, say, $n_0, n_1, \ldots, n_k$. Let π_i be a minimal position of n_i in g for each $i \leq k$. For each $i \leq k$, we construct a reduction $T_i\colon g_i \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} g_{i+1}$ with $g_0 = g$. Since all sub-term graphs at depth $< d$ are root stable, each step in T_i takes place at depth $\geq d$ and thus π_{i+1} is still a position in g_{i+1} of a node at depth d. If $g_i|_{\pi_i}$ is root-active, then Lemma 4.16 yields a reduction $g_i|_{\pi_i} \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} \bot$. Let T_i be the embedding of this reduction into g_i at position π_i. Hence, $g_{i+1}|_{\pi_i} = \bot$ is root-stable. If $g_i|_{\pi_i}$ is not root-active, then Lemma 4.16 yields a reduction $g_i|_{\pi_i} \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} g'_i$ to a root-stable term graph g'_i. Let

T_i be the embedding of this reduction into g_i at position π_i. Hence, $g_{i+1}|_{\pi_i} = g'_i$ is root-stable.

Define $S_d\colon = \prod_{i \leq k} T_i$. Since, by Lemma 4.16, root-stability is preserved by strongly p-converging reductions, we can conclude that $S_d\colon g \xrightarrow{p}\!\!\!\!\rightarrow_{\mathcal{R}} g_{k+1}$ such that all sub-term graphs at depth at most d in g_{k+1} are root-stable. □

Note that the assumption that all sub-term graphs at depth $< d$ are root-stable is crucial. Otherwise, reductions within sub-term graphs at depth d may take place at depth $< d$!

Finally, the strategy for rewriting a term graph into normal form is to simply iterate the reductions that are given by Lemma 4.17 above.

Theorem 4.18. *Every GRS $\mathcal{R}$ is normalising w.r.t. strongly p-converging reductions. That is, for each partial term graph g, there is a reduction $g \xrightarrow{p}\!\!\!\!\rightarrow_{\mathcal{R}} h$ to a normal form h in $\mathcal{R}$.*

Proof. Given a partial term graph g_0, take the reductions $S_d\colon g_d \xrightarrow{p}\!\!\!\!\rightarrow g_{d+1}$ from Lemma 4.17 for each $d \in \mathbb{N}$ and construct $S = \prod_{d<\omega} S_d$. By Proposition 4.10, we have $S\colon g_0 \xrightarrow{p}\!\!\!\!\rightarrow g_\omega$ for some g_ω. As, by Lemma 4.16, root-stability is preserved by strongly p-converging reductions, and each reduction S_d increases the depth up to which sub-term graphs are root-stable, we know that each sub-term graph of g_ω is root-stable, i.e. g_ω is a normal form. □

The ability of strong p-convergence to normalise any term graph will be a crucial component of the proof of completeness of infinitary term graph rewriting.

5 Soundness and Completeness Properties

In this section, we will study the relationship between GRSs and the corresponding TRSs they simulate. In particular, we will show the soundness of GRSs w.r.t. strong convergence and a restricted form of completeness. To this end we make use of the isomorphism between terms and canonical term trees as outlined at the end of Section 2.

Proposition 5.1. *The unravelling $\mathcal{U}(g)$ of a term graph $g \in \mathcal{G}^\infty(\Sigma)$ is given by the labelled quotient tree $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$.*

Proof. Since $\mathcal{I}_{\mathcal{P}(g)}$ is a subrelation of $\sim_g$, we know that $(\mathcal{P}(g), g(\cdot), \mathcal{I}_{\mathcal{P}(g)})$ is a labelled quotient tree and thus uniquely determines a term t. By Lemma 2.7, there is a homomorphism from t to g. Hence, $\mathcal{U}(g) = t$. □

Before we start investigating the correspondences between term rewriting and term graph rewriting, we need to transfer the notions of left-linearity and orthogonality to GRSs:

Definition 5.2. Let $\mathcal{R} = (\Sigma, R)$ be a GRS. A rule $\rho \in R$ is called *left-linear* resp. *left-finite* if its left-hand side ρ_l is a term tree resp. a finite term graph. The GRS $\mathcal{R}$ is called *left-linear* resp. *left-finite* if all its rules are left-linear resp. *left-finite*. The GRS $\mathcal{R}$ is called *orthogonal* if it is left-linear and the TRS $\mathcal{U}(\mathcal{R})$ is *non-overlapping*.

Note that the unravelling $\mathcal{U}(\mathcal{R})$ of a GRS $\mathcal{R}$ is left-linear if $\mathcal{R}$ is left-linear, that $\mathcal{U}(\mathcal{R})$ is left-finite if $\mathcal{R}$ is left-linear and left-finite, and that $\mathcal{U}(\mathcal{R})$ is orthogonal if $\mathcal{R}$ is orthogonal.

We have to single out a particular kind of redex that manifests a peculiar behaviour:

Definition 5.3. A redex of a rule (g, l, r) is called *circular* if l and r are distinct but the matching $\mathcal{V}$-homomorphism ϕ maps them to the same node, i.e. $l \neq r$ but $\phi(l) = \phi(r)$.

Kennaway et al. [15] show that circular redexes only reduce to themselves:

Proposition 5.4. *For every circular ρ-redex $g|_n$, we have $g \mapsto_{n,\rho} g$.*

However, contracting the unravelling of a circular redex also yields the same term:

Lemma 5.5. *For every circular ρ-redex $g|_n$, we have $\mathcal{U}(g) \to_{\pi,\mathcal{U}(\rho)} \mathcal{U}(g)$ for all $\pi \in \mathcal{P}_g(n)$.*

Proof. Since there is a circular ρ-redex, we know that the right-hand side root r^ρ is reachable but different from the left-hand side root l^ρ of ρ. Hence, there is a non-empty path $\widehat{\pi}$ from l^ρ to r^ρ. Because $g|_n$ is a circular ρ-redex, the corresponding matching $\mathcal{V}$-homomorphism maps both l^ρ and r^ρ to n. Since Δ-homomorphisms preserve paths, we thus know that $\widehat{\pi}$ is also a path from n to itself in g. In other words, $\pi \in \mathcal{P}_g(n)$ implies $\pi \cdot \widehat{\pi} \in \mathcal{P}_g(n)$. Consequently, for each $\pi \in \mathcal{P}_g(n)$, we have that $\mathcal{U}(g)|_\pi = \mathcal{U}(g)|_{\pi\cdot\widehat{\pi}}$.

Since there is a path $\widehat{\pi}$ from l^ρ to r^ρ, the unravelling $\mathcal{U}(\rho)$ of ρ is of the form $s \to s|_{\widehat{\pi}}$. Hence, we know that each application of $\mathcal{U}(\rho)$ at a position π in some term t replaces the subterm at π with the subterm at $\pi \cdot \widehat{\pi}$ in t, i.e. $t \to_{\pi,\mathcal{U}(\rho)} t[t|_{\pi\cdot\widehat{\pi}}]_\pi$.

Combining the two findings above, we obtain that

$$\mathcal{U}(g) \to_{\pi,\mathcal{U}(\rho)} \mathcal{U}(g)[\mathcal{U}(g)|_{\pi\cdot\widehat{\pi}}]_\pi = \mathcal{U}(g)[\mathcal{U}(g)|_\pi]_\pi = \mathcal{U}(g) \quad \text{for all } \pi \in \mathcal{P}_g(n) \qquad \square$$

The following two properties due to Kennaway et al. [15] show how single term graph reduction steps relate to term reductions in the corresponding unravelling.[1]

Proposition 5.6. *Given a left-linear GRS $\mathcal{R}$ and a term graph g in $\mathcal{R}$, it holds that g is a normal form in $\mathcal{R}$ iff $\mathcal{U}(g)$ is a normal form in $\mathcal{U}(\mathcal{R})$.*

Theorem 5.7. *Let $\mathcal{R}$ be a left-linear, left-finite GRS with a reduction step $g \to_{n,\rho} h$. Then $S\colon \mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ such that the depth of every redex contracted in S is greater or equal to $\mathsf{depth}_g(n)$. In particular, if the ρ-redex $g|_n$ is not circular, then S is a complete development of the set of redex occurrences $\mathcal{P}_g(n)$ in $\mathcal{U}(g)$.*

In the following, we will generalise the above soundness theorem to strongly p-converging term graph reductions. We will then use the correspondence between strong m-convergence and strong p-convergence in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ to transfer that result to strongly m-converging reductions.

At first, we can observe that the limit inferior commutes with the unravelling:

[1] The original results are on finite term graphs. However, for the correspondence of normal forms, this restriction is not necessary, and for the soundness, only the finiteness of left-hand sides is crucial.

Proposition 5.8. *For each sequence* $(g_\iota)_{\iota<\alpha}$ *in* $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\perp), \leq^{\mathsf{S}}_\perp)$*, we have that* $\mathcal{U}(\liminf_{\iota\to\alpha} g_\iota) = \liminf_{\iota\to\alpha} \mathcal{U}(g_\iota)$*.*

Proof. This is an immediate consequence of Theorem 3.2 and Proposition 5.1. □

In order to prove soundness w.r.t. strong p-convergence, we need to turn the statement about the depth of redexes in Theorem 5.7 into a statement about the corresponding reduction contexts. To this end, we make use of the fact that the semilattice structure of $\leq^{\mathsf{S}}_\perp$ admits greatest lower bounds for non-empty sets of term graphs:

Proposition 5.9 ([6])**.** *In the partially ordered set* $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\perp), \leq^{\mathsf{S}}_\perp)$ *every non-empty set* G *has a greatest lower bound* $\bigsqcap G$ *given by the following labelled quotient tree* $(P, l, \sim)$*:*

$$P = \left\{ \pi \in \bigcap\nolimits_{g\in G} \mathcal{P}(g) \;\middle|\; \forall \pi' < \pi \exists f \in \Sigma_\perp \forall g \in G : g(\pi') = f \right\}$$

$$l(\pi) = \begin{cases} f & \text{if } \forall g \in G : f = g(\pi) \\ \perp & \text{otherwise} \end{cases} \qquad \sim \;=\; \bigcap\nolimits_{g\in G} \sim_g \cap\, P \times P$$

In particular, the glb of a set of term trees is again a term tree.

We can then prove the following proposition that relates the reduction context of a term graph reduction step with the reduction contexts of the corresponding term reduction:

Proposition 5.10. *For each reduction step* $g \to_c h$ *in a left-linear, left-finite GRS* $\mathcal{R}$*, there is a non-empty reduction* $S = (t_\iota \to_{c_\iota} t_{\iota+1})_{\iota<\alpha}$ *with* $S\colon \mathcal{U}(g) \overset{p}{\twoheadrightarrow}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ *and* $\mathcal{U}(c) = \bigsqcap_{\iota<\alpha} c_\iota$*.*

Proof. By Theorem 5.7, there is a reduction $S\colon \mathcal{U}(g) \overset{m}{\twoheadrightarrow}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. At first we assume that the redex $g|_n$ contracted in $g \to_n h$ is not a circular redex. Hence, S is a complete development of the set of redex occurrences $\mathcal{P}_g(n)$ in $\mathcal{U}(g)$. By Theorem 1.3, we then obtain $S\colon \mathcal{U}(g) \overset{p}{\twoheadrightarrow}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. From Lemma 4.5 and Proposition 5.1 it follows that $\mathcal{U}(g\backslash n)$ is obtained from $\mathcal{U}(g)$ by replacing each subterm of $\mathcal{U}(g)$ at a position in $\mathcal{P}^m_g(n)$, i.e. a minimal position of n, by $\perp$. Since each step $t_\iota \to_{\pi_\iota} t_{\iota+1}$ in S contracts a redex at a position π_ι that has a prefix in $\mathcal{P}^m_g(n)$, we have, by Proposition 5.9 and Corollary 3.1, that $\mathcal{U}(g\backslash n) \leq^{\mathsf{S}}_\perp \bigsqcap_{\iota<\alpha} t_\iota[\perp]_{\pi_\iota} = \bigsqcap_{\iota<\alpha} c_\iota$. Moreover, for each $\pi \in \mathcal{P}^m_g(n)$ there is a step at $\iota_\pi < \alpha$ in S that takes place at π. From Proposition 5.9, it is thus clear that $\mathcal{U}(g\backslash n) = \bigsqcap_{\pi\in\mathcal{P}^m_g(n)} c_{\iota_\pi}$, which means that $\mathcal{U}(g\backslash n) \geq^{\mathsf{S}}_\perp \bigsqcap_{\iota<\alpha} c_\iota$. Due to the antisymmetry of $\leq^{\mathsf{S}}_\perp$, we thus know that $\mathcal{U}(g\backslash n) = \bigsqcap_{\iota<\alpha} c_\iota$. Then $\mathcal{U}(c) = \bigsqcap_{\iota<\alpha} c_\iota$ follows from the fact that $c \cong g\backslash n$.

If the ρ-redex $g|_n$ contracted in $g \to_{\rho,n} h$ is a circular redex, then $g = h$ according to Proposition 5.4. However, by Lemma 5.5, each $\mathcal{U}(\rho)$-redex at positions in $\mathcal{P}_g(n)$ in $\mathcal{U}(g)$ reduces to itself as well. Hence, we get a reduction $\mathcal{U}(g) \overset{p}{\twoheadrightarrow}_{\mathcal{U}(\rho)} \mathcal{U}(h)$ via a complete development of the redexes at the minimal positions $\mathcal{P}^m_g(n)$ of n in g. The equality $\mathcal{U}(c) = \bigsqcap_{\iota<\alpha} c_\iota$ then follows as for the first case above. □

In order to prove the soundness of strongly p-converging term graph reductions, we need the following technical lemma, which can be proved easily:

Lemma 5.11. *Let $(a_\iota)_{\iota<\alpha}$ be a sequence in a complete semilattice $(A,\leq)$ and $(\gamma_\iota)_{\iota<\delta}$ a strictly monotone sequence in the ordinal α such that $\bigsqcup_{\iota<\delta}\gamma_\iota = \alpha$. Then*

$$\liminf_{\iota\to\alpha} a_\iota = \liminf_{\beta\to\delta}\left(\bigsqcap_{\gamma_\beta\leq\iota<\gamma_{\beta+1}} a_\iota\right).$$

Theorem 5.12. *If $g \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{R}} h$ in a left-linear, left-finite GRS $\mathcal{R}$, then $\mathcal{U}(g) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$.*

Proof. Let $S = (g_\iota \to_{c_\iota} g_{\iota+1})_{\iota<\alpha}$ be a reduction strongly p-converging to g_α in $\mathcal{R}$. By Proposition 5.10, there is, for each $\gamma<\alpha$, a reduction $T_\gamma\colon \mathcal{U}(g_\gamma) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(g_{\gamma+1})$ such that

$$\bigsqcap\nolimits_{\iota<|T_\gamma|} \overline{c}_\iota = \mathcal{U}(c_\gamma) \qquad (*)$$

where $(\overline{c}_\iota)_{\iota<|T_\gamma|}$ is the sequence of reduction contexts in T_γ.

Define for each $\delta\leq\alpha$ the concatenation $U_\delta = \prod_{\iota<\delta} T_\iota$. We will show that $U_\delta\colon \mathcal{U}(g_0) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(g_\delta)$ for each $\delta\leq\alpha$ by induction on δ. The theorem is then obtained from the case $\delta=\alpha$.

The case $\delta=0$ is trivial, and the case $\delta=\delta'+1$ follows from the induction hypothesis.

For the case that δ is a limit ordinal, let $U_\delta = (t_\iota \to_{c'_\iota} t_{\iota+1})_{\iota<\beta}$. For each $\gamma<\beta$ we find some $\delta'<\delta$ with $U_\delta|_\gamma < U_{\delta'}$. By induction hypothesis, we can assume that $U_{\delta'}$ is strongly p-continuous. Thus, the proper prefix $U_\delta|_\gamma$ strongly p-converges to t_γ. This shows that each proper prefix $U_\delta|_\gamma$ of U_δ strongly p-converges to t_γ. Hence, U_δ is strongly p-continuous.

In order to show that $U_\delta\colon \mathcal{U}(g_0) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(g_\delta)$, it remains to be shown that $\liminf_{\iota\to\beta} c'_\iota = \mathcal{U}(g_\delta)$. Since S is strongly p-converging, we know that $\liminf_{\iota\to\delta} c_\iota = g_\delta$. By Proposition 5.8, we thus have $\liminf_{\iota\to\delta}\mathcal{U}(c_\iota) = \mathcal{U}(g_\delta)$. By $(*)$ and the construction of U_δ, there is a strictly monotone sequence $(\gamma_\iota)_{\iota<\delta}$ with $\gamma_0=0$ and $\bigsqcup_{\iota<\delta}\gamma_\iota=\beta$ such that $\mathcal{U}(c_\iota) = \bigsqcap_{\gamma_\iota\leq\gamma<\gamma_{\iota+1}} c'_\gamma$ for all $\iota<\delta$. Thus, we can complete the proof as follows:

$$\mathcal{U}(g_\delta) = \liminf_{\iota\to\delta}\mathcal{U}(c_\iota) = \liminf_{\iota\to\delta}\left(\bigsqcap_{\gamma_\iota\leq\gamma<\gamma_{\iota+1}} c'_\gamma\right) \overset{\text{Lem. 5.11}}{=} \liminf_{\iota\to\beta} c'_\iota \qquad \square$$

By combining the soundness result above with the normalisation of strong p-convergence, we obtain the following completeness result:

Theorem 5.13. *Given an orthogonal, left-finite GRS $\mathcal{R}$, we find for each reduction $\mathcal{U}(g) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} t$, a reduction $g \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{R}} h$ such that $t \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$.*

Proof. Let $\mathcal{U}(g) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} t$. By Theorem 4.18 there is a normalising reduction $g \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{R}} h$. According to Theorem 5.12, $g \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{R}} h$ implies $\mathcal{U}(g) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. By Proposition 5.6, $\mathcal{U}(h)$ is a normal form in $\mathcal{U}(\mathcal{R})$. Since orthogonal, left-finite TRSs are confluent w.r.t. strong p-convergence [4], the reduction $\mathcal{U}(g) \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$ together with $\mathcal{U}(g) \xrightarrow{m}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} t$ yields a reduction $t \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. $\square$

The results above make strongly p-converging term graph reductions sound and complete for strongly p-converging term reductions in the sense of *adequacy* of Kennaway et al. [15].

The notion of adequacy of Kennaway et al. [15] does not only comprise soundness and completeness but also demands that the unravelling $\mathcal{U}(\cdot)$ is surjective and both preserves and reflects normal forms. For infinitary term graph rewriting, surjectivity of $\mathcal{U}(\cdot)$ is trivial since each term is the image of itself under $\mathcal{U}(\cdot)$ and the preservation and reflection of normal forms is given for left-linear GRSs by Proposition 5.6.

From the soundness result for strong p-convergence, we can straightforwardly derive a corresponding result for strong m-convergence:

Theorem 5.14. *If $g \xrightarrow{m}_{\mathcal{R}} h$ in a left-linear, left-finite GRS $\mathcal{R}$, then $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$.*

Proof. Given a reduction $S\colon g \xrightarrow{m}_{\mathcal{R}} h$, we know, by Theorem 4.14, that $S\colon g \xrightarrow{p}_{\mathcal{R}} h$ in $\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma)$. According to Theorem 5.12, we then find a reduction $\mathcal{U}(g) \xrightarrow{p}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. Since, g, h are total, so are $\mathcal{U}(g), \mathcal{U}(h)$. Hence, by Corollary 7.15 of [4], we obtain a reduction $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. □

Similar to the proof of Theorem 5.13, we can derive a weakened completeness property for strong m-convergence:

Theorem 5.15. *Given an orthogonal, left-finite GRS $\mathcal{R}$ that is normalising w.r.t. strongly m-converging reductions, we find for each normalising reduction $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} t$ a reduction $g \xrightarrow{m}_{\mathcal{R}} h$ such that $t = \mathcal{U}(h)$.*

Proof. Let $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} t$ with t a normal form in $\mathcal{U}(\mathcal{R})$. As $\mathcal{R}$ is normalising w.r.t. strongly m-converging reductions, there is a reduction $g \xrightarrow{m}_{\mathcal{R}} h$ with h a normal form in $\mathcal{R}$. According to Theorem 5.14, we then find a reduction $\mathcal{U}(g) \xrightarrow{m}_{\mathcal{U}(\mathcal{R})} \mathcal{U}(h)$. By Proposition 5.6, $\mathcal{U}(h)$ is a normal form in $\mathcal{U}(\mathcal{R})$. Since $\mathcal{U}(\mathcal{R})$ is left-finite and orthogonal, we know that, according to Theorem 7.15 in [16], $\mathcal{R}$ has unique normal forms w.r.t. $\xrightarrow{m}$. Consequently, $t = \mathcal{U}(h)$. □

While the above theorem is restricted to normalising GRSs, we conjecture that this restriction is not needed: as soon as we have a compression lemma for strong p-convergence, completeness of normalising strong m-convergence follows from the completeness of strong p-convergence.

Yet, as mentioned in the in the introduction, the restriction to normalising reductions is crucial. The counterexample that Kennaway et al. [15] give for their informal notion of term graph convergence in fact also applies to our notion of strong m-convergence.

6 Conclusions

By generalising the metric and partial order based notions of convergence from terms to term graphs, we have obtained two infinitary term graph rewriting calculi that simulate infinitary term rewriting adequately. Not only do these results show the appropriateness of our notions of infinitary term graph rewriting. They also

refute the claim of Kennaway et al. [15] that infinitary term graph rewriting cannot adequately simulate infinitary term rewriting.

Since reasoning over the rather operational style of term graph rewriting is tedious, we tried to simplify the proofs using labelled quotient trees. In future work, it would be helpful to characterise term graph rewriting itself in this way or to adopt a more declarative approach to term graph rewriting [1, 10, 11].

We think that, in this context, strong p-convergence may help to bridge the differences between the operational style of Barendregt et al. [7] and the declarative formalisms [1, 10, 11], which arise from the different way of contracting circular redexes. While in the operational approach that we adopted here, circular redexes are contracted to themselves, they are contracted to $\bot$ in the abovementioned declarative approaches. However, since circular redexes are root-active, they can be rewritten to $\bot$ in a strongly p-converging reduction.

Bibliography

[1] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3-4):207–240, 1996. ISSN 0169-2968. doi: 10.3233/FI-1996-263401.

[2] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[3] P. Bahr. Abstract Models of Transfinite Reductions. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.49.

[4] P. Bahr. Partial Order Infinitary Term Rewriting and Böhm Trees. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.RTA.2010.67.

[5] P. Bahr. Modes of Convergence for Term Graph Rewriting. In M. Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 139–154, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.RTA.2011.139.

[6] P. Bahr. Convergence in Infinitary Term Graph Rewriting Systems is Simple. Submitted to Math. Struct. in Comp. Science, 2012.

[7] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In P. C. T. de Bakker A. J. Nijman, editor, *Parallel Architectures and Languages Europe,*

Volume II: Parallel Languages, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Berlin / Heidelberg, 1987. doi: 10.1007/3-540-17945-3_8.

[8] E. Barendsen. Term Graph Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 13, pages 712–743. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[9] J. Bongaerts. Topological Convergence in Infinitary Abstract Rewriting. Master's thesis, Utrecht University, 2011.

[10] A. Corradini and F. Drewes. Term Graph Rewriting and Parallel Term Rewriting. In *TERMGRAPH*, pages 3–18, 2011. doi: 10.4204/EPTCS.48.3.

[11] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence between the operational and the categorical description. *RAIRO - Theoretical Informatics and Applications1*, 33(4):467–493, 1999. doi: 10.1051/ita:1999128.

[12] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1): 68–95, 1977. ISSN 0004-5411. doi: 10.1145/321992.321997.

[13] J. L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955. ISBN 0387901256.

[14] R. Kennaway and F.-J. de Vries. Infinitary Rewriting. In Terese, editor, *Term Rewriting Systems*, chapter 12, pages 668–711. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

[15] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. ISSN 0164-0925. doi: 10.1145/177492.177577.

[16] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1075.

[17] R. Kennaway, V. van Oostrom, and F.-J. de Vries. Meaningless Terms in Rewriting. *Journal of Functional and Logic Programming*, 1999(1):1–35, 1999.

[18] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003. ISBN 9780521391153.

Appendices

A Proofs

A.1 Homomorphisms

Proposition A.1 (Δ-homomorphism preorder)**.** *Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a category which is a preorder. That is, there is at most one Δ-homomorphism from one term graph to another.*

Proof. The identity Δ-homomorphism is obviously the identity mapping on the set of nodes. Moreover, an easy equational reasoning yields that the composition of two Δ-homomorphisms is again a Δ-homomorphism. Associativity of this composition follows from the fact that Δ-homomorphisms are functions.

To show that this category is a preorder, assume that there are two Δ-homomorphisms $\phi_1, \phi_2 \colon g \to_\Delta h$. We prove that $\phi_1 = \phi_2$ by showing that $\phi_1(n) = \phi_2(n)$ for all $n \in N^g$ by induction on the depth of n.

Let $\mathsf{depth}_g(n) = 0$, i.e. $n = r^g$. By the root condition, we have that $\phi_1(r^g) = r^h = \phi_2(r^g)$. Let $\mathsf{depth}_g(n) = d > 0$. Then n has a position $\pi \cdot \langle i \rangle$ in g such that $\mathsf{depth}_g(n') < d$ for $n' = \mathsf{node}_g(\pi)$. Hence, we can employ the induction hypothesis for n' to obtain the following:

$$\begin{aligned} \phi_1(n) &= \mathsf{suc}_i^h(\phi_1(n')) && \text{(successor condition for } \phi_1) \\ &= \mathsf{suc}_i^h(\phi_2(n')) && \text{(induction hypothesis)} \\ &= \phi_2(n) && \text{(successor condition for } \phi_2) \end{aligned}$$

□

Lemma A.2 (homomorphisms are surjective)**.** *Every homomorphism $\phi \colon g \to h$, with $g, h \in \mathcal{G}^\infty(\Sigma)$, is surjective.*

Proof. Follows from an easy induction on the depth of the nodes in h. □

Lemma A.3 (characterisation of Δ-homomorphisms)**.** *Given term graphs $g, h \in \mathcal{G}^\infty(\Sigma)$, a function $\phi \colon N^g \to N^h$ is a Δ-homomorphism $\phi \colon g \to_\Delta h$ iff the following holds for all $n \in N^g$:*

(a) $\mathcal{P}_g(n) \subseteq \mathcal{P}_h(\phi(n))$, and (b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ whenever $\mathsf{lab}^g(n) \notin \Delta$.

Proof. For the "only if" direction, assume that $\phi \colon g \to_\Delta h$. (b) is the labelling condition and is therefore satisfied by ϕ. To establish (a), we show the equivalent statement

$$\forall \pi \in \mathcal{P}(g).\, \forall n \in N^g.\, \pi \in \mathcal{P}_g(n) \implies \pi \in \mathcal{P}_h(\phi(n))$$

We do so by induction on the length of π. If $\pi = \langle\rangle$, then $\pi \in \mathcal{P}_g(n)$ implies $n = r^g$. By the root condition, we have $\phi(r^g) = r^h$ and, therefore, $\pi = \langle\rangle \in \phi(r^g)$. If $\pi = \pi' \cdot \langle i \rangle$, then let $n' = \mathsf{node}_g(\pi')$. Consequently, $\pi' \in \mathcal{P}_g(n')$ and, by induction hypothesis, $\pi' \in \mathcal{P}_h(\phi(n'))$. Since $\pi = \pi' \cdot \langle i \rangle$, we have $\mathsf{suc}_i^g(n') = n$. By the

successor condition we can conclude $\phi(n) = \mathsf{suc}_i^h(\phi(n'))$. This and $\pi' \in \mathcal{P}_h(\phi(n'))$ yields that $\pi' \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n))$.

For the "if" direction, we assume (a) and (b). The labelling condition follows immediately from (b). For the root condition, observe that since $\langle\rangle \in \mathcal{P}_g(r^g)$, we also have $\langle\rangle \in \mathcal{P}_h(\phi(r^g))$. Hence, $\phi(r^g) = r^h$. In order to show the successor condition, let $n, n' \in N^g$ and $0 \leq i < \mathsf{ar}_g(n)$ such that $\mathsf{suc}_i^g(n) = n'$. Then there is a position $\pi \in \mathcal{P}_g(n)$ with $\pi \cdot \langle i \rangle \in \mathcal{P}_g(n')$. By (a), we can conclude that $\pi \in \mathcal{P}_h(\phi(n))$ and $\pi \cdot \langle i \rangle \in \mathcal{P}_h(\phi(n'))$ which implies that $\mathsf{suc}_i^h(\phi(n)) = \phi(n')$. □

Corollary A.4 (characterisation of Δ-isomorphisms)**.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, the following holds:*

(i) $\phi\colon N^g \to N^h$ is a Δ-isomorphism iff for all $n \in N^g$

(a) $\mathcal{P}_h(\phi(n)) = \mathcal{P}_g(n)$, and
(b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ or $\mathsf{lab}^g(n), \mathsf{lab}^h(\phi(n)) \in \Delta$.

(ii) $g \cong_\Delta h$ iff (a) $\sim_g = \sim_h$, and (b) $g(\pi) = h(\pi)$ or $g(\pi), h(\pi) \in \Delta$.

Proof. Immediate consequence of Lemma A.3 resp. Lemma 2.7 and Proposition A.1. □

A.2 Reduction Contexts

We start with making the definition of local truncations – and thus reduction contexts – more precise by expanding Definition 4.4:

Definition 4.4 (local truncation)**.** Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $n \in N^g$. The *local truncation* of g at n, denoted $g\backslash n$, is obtained from g by labelling n with $\bot$ and removing all outgoing edges from n as well as all nodes that thus become unreachable from the root:

$$N^{g\backslash n} \text{ is the least set } M \text{ satisfying } \begin{array}{l}(a)\ r^g \in M, \text{and} \\ (b)\ m \in M \setminus \{n\} \implies \mathsf{suc}^g(m) \subseteq M.\end{array}$$

$$r^{g\backslash n} = r^g$$

$$\mathsf{lab}^{g\backslash n} = \begin{cases} \mathsf{lab}^g(m) & \text{if } m \neq n \\ \bot & \text{if } m = n \end{cases}$$

$$\mathsf{suc}^{g\backslash n}(m) = \begin{cases} \mathsf{suc}^g(m) & \text{if } m \neq n \\ \langle\rangle & \text{if } m = n \end{cases}$$

The following lemma shows that local truncations only remove positions from a term graph but do not alter them:

Lemma A.5. *Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$, $n \in N^g$ and $\pi \in \mathcal{P}(g\backslash n)$. Then $\mathsf{node}_g(\pi) = \mathsf{node}_{g\backslash n}(\pi)$.*

Proof. We proceed by induction on the length of π. The case $\pi = \langle\rangle$ follows from the definition $r^{g\backslash n} = r^g$. If $\pi = \pi' \cdot \langle i \rangle$, we can use the induction hypothesis

to obtain that $\mathsf{node}_g(\pi') = \mathsf{node}_{g\backslash n}(\pi')$. As $\pi' \cdot \langle i \rangle \in \mathcal{P}(g\backslash n)$, we know that $\mathsf{node}_{g\backslash n}(\pi') \neq n$. Hence:

$$\begin{aligned}\mathsf{node}_g(\pi) &= \mathsf{suc}_i^g(\mathsf{node}_g(\pi')) = \mathsf{suc}_i^g(\mathsf{node}_{g\backslash n}(\pi')) \\ &= \mathsf{suc}_i^{g\backslash n}(\mathsf{node}_{g\backslash n}(\pi')) = \mathsf{node}_{g\backslash n}(\pi)\end{aligned}$$

□

A.2.1 Proof of Lemma 4.5

Lemma 4.5. *For each $g \in \mathcal{G}^\infty(\Sigma_\perp)$ and $n \in N^g$, the local truncation $g\backslash n$ has the following labelled quotient tree $(P, l, \sim)$:*

$$\begin{aligned}P &= \{\pi \in \mathcal{P}(g) \mid \forall \pi' < \pi \colon \pi' \notin \mathcal{P}_g(n)\} \\ \sim &= \sim_g \cap\, P \times P \\ l(\pi) &= \begin{cases} g(\pi) & \text{if } \pi \notin \mathcal{P}_g(n) \\ \perp & \text{if } \pi \in \mathcal{P}_g(n) \end{cases} \quad \text{for all } \pi \in P\end{aligned}$$

Proof of Lemma 4.5. We will show in the following that triples $(P, l, \sim)$ and $(\mathcal{P}(g\backslash n), g\backslash n(\cdot), \sim_{g\backslash n})$ coincide.

By Lemma A.5 $\mathcal{P}(g\backslash n) \subseteq \mathcal{P}(g)$. Therefore, in order to prove that $\mathcal{P}(g\backslash n) \subseteq P$, we assume some $\pi \in \mathcal{P}(g\backslash n)$ and show by induction on the length of π that no proper prefix of π is a position of n in g. The case $\pi = \langle\rangle$ is trivial as $\langle\rangle$ has no proper prefixes. If $\pi = \pi' \cdot \langle i \rangle$, we can assume by induction that $\pi' \in P$ since $\pi' \in \mathcal{P}(g\backslash n)$. Consequently, no proper prefix of π' is in $\mathcal{P}_g(n)$. It thus remains to be shown that π' itself is not in $\mathcal{P}_g(n)$. Since $\pi' \cdot \langle i \rangle \in \mathcal{P}(g\backslash n)$, we know that $\mathsf{suc}_i^{g\backslash n}(\mathsf{node}_{g\backslash n}(\pi'))$ is defined. Therefore, $\mathsf{node}_{g\backslash n}(\pi')$ cannot be n, and since, by Lemma A.5, $\mathsf{node}_{g\backslash n}(\pi') = \mathsf{node}_g(\pi')$, neither can $\mathsf{node}_g(\pi')$. In other words, $\pi' \notin \mathcal{P}_g(n)$.

For the converse direction $P \subseteq \mathcal{P}(g\backslash n)$, assume some $\pi \in P$. We will show by induction on the length of π, that then $\pi \in \mathcal{P}(g\backslash n)$. The case $\pi = \langle\rangle$ is trivial. If $\pi = \pi' \cdot \langle i \rangle$, then also $\pi' \in P$ which, by induction, implies that $\pi' \in \mathcal{P}(g\backslash n)$. Let $m = \mathsf{node}_{g\backslash n}(\pi')$. Since $\pi \in P$, we have that $\pi' \notin \mathcal{P}_g(n)$. Consequently, as Lemma A.5 implies $m = \mathsf{node}_g(\pi')$, we can deduce that $m \neq n$. That means, according to the definition of $g\backslash n$, that $\mathsf{suc}^{g\backslash n}(m) = \mathsf{suc}^g(m)$. Hence, $\pi' \cdot \langle i \rangle \in \mathcal{P}_{g\backslash n}(\mathsf{suc}_i^{g\backslash n}(m))$ and thus $\pi \in \mathcal{P}(g\backslash n)$.

For the equality $\sim = \sim_{g\backslash n}$, assume some $\pi_1, \pi_2 \in P$. Since $P = \mathcal{P}(g\backslash n)$, we then have the following equivalences:

$$\begin{aligned}\pi_1 \sim \pi_2 &\iff \pi_1 \sim_g \pi_2 \\ &\iff \mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2) \\ &\iff \mathsf{node}_{g\backslash n}(\pi_1) = \mathsf{node}_{g\backslash n}(\pi_2) && \text{(Lemma A.5)} \\ &\iff \pi_1 \sim_{g\backslash n} \pi_2\end{aligned}$$

For the equality $l = g\backslash n(\cdot)$, consider some $\pi \in \mathcal{P}(g\backslash n)$. Since $\mathsf{node}_g(\pi) = n$ iff $\pi \in \mathcal{P}_g(n)$, we can reason as follows:

$$g\backslash n(\pi) = \mathsf{lab}^{g\backslash n}(\mathsf{node}_{g\backslash n}(\pi)) \overset{\text{Lem. A.5}}{=} \mathsf{lab}^{g\backslash n}(\mathsf{node}_g(\pi)) = \begin{cases} g(\pi) & \text{if } \pi \notin \mathcal{P}_g(n) \\ \perp & \text{if } \pi \in \mathcal{P}_g(n) \end{cases}$$

□

A.2.2 Proof of Proposition 4.7

As we have indicated in the main body of the paper, we need to consider a positional notion of truncation:

Definition A.6 (positional truncations)**.** Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $Q \subseteq \mathbb{N}^*$ a set of positions.

(i) The set Q is called *admissible for truncating* g if, for all $\pi_1, \pi_2 \in \mathbb{N}^*$ with $\pi_1 \sim_g \pi_2$ and $\pi \not\leq \pi_1, \pi_2$ for all $\pi \in Q$, we have that $\pi_1 \cdot \langle i \rangle \in Q$ implies $\pi_2 \cdot \langle i \rangle \in Q$ for all $i \in \mathbb{N}$.

(ii) Given that Q is admissible for truncating g, the *positional truncation* of g at Q, denoted $g\backslash[Q]$, is the canonical term graph given by the following labelled quotient tree $(P, l, \sim)$:

$$P = \{\, \pi \in \mathcal{P}(g) \,|\, \forall \pi' < \pi . \pi' \notin Q \,\}$$

$$l(\pi) = \begin{cases} g(\pi) & \text{if } \pi \notin Q \\ \bot & \text{if } \pi \in Q \end{cases} \quad \text{for all } \pi \in P$$

$$\sim \; = \; \sim_g \cap \Big(\Big(Q^+ \times Q^+ \Big) \cup \big(Q^- \times Q^- \big) \Big), \quad \text{where } Q^+ = Q \cap P, Q^- = P \setminus Q$$

In other words: $\pi_1 \sim \pi_2$ iff $\pi_1 \sim_g \pi_2$, $\pi_1, \pi_2 \in P$ and $\pi_1 \in Q$ iff $\pi_2 \in Q$.

The above definition yields a canonical term graph, given that the set Q is indeed admissible for truncating the term graph g:

Proposition A.7 (well-definedness of positional truncations)**.** *Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $Q \subseteq \mathbb{N}^*$ a set of positions admissible for truncating g. Then the triple $(P, l, \sim)$ defined in Definition A.6 indeed constitutes a labelled quotient tree and thus the canonical term graph $g\backslash[Q]$ is well-defined.*

Proof. One can easily check that the triple $(P, l, \sim)$ satisfies the axioms of labelled quotient trees; cf. [5]. □

As an immediate corollary of the definition of positional truncations, we obtain the following:

Corollary A.8. *Given a term graph $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and a set Q admissible for truncating g, we have that $g\backslash[Q] \leq^S_\bot g$.*

Proof. According to Corollary 3.1, this follows immediately from the definition of $g\backslash[Q]$. □

The following two lemmas show that local truncations are only a special case of positional truncations.

Lemma A.9. *For every term graph $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and node $n \in N^g$, the set $\mathcal{P}_g(n)$ is admissible for truncating g.*

Proof. Let $\pi_1 \sim_g \pi_2$ and $\pi_1 \cdot \langle i \rangle \in \mathcal{P}_g(n)$. Then there is a node $m \in N^g$ such that $\pi_1, \pi_2 \in \mathcal{P}_g(m)$ and $\mathsf{suc}_i^g(m) = n$. Consequently, also $\pi_2 \cdot \langle i \rangle \in \mathcal{P}_g(n)$. □

Lemma A.10. *For each $g \in \mathcal{G}^\infty(\Sigma_\perp)$ and $n \in N^g$, we have that $g\backslash n \cong g\backslash[\mathcal{P}_g(n)]$.*

Proof. Let $(P_1, l_1, \sim_1)$ and $(P_2, l_2, \sim_2)$ be the labelled quotient trees of $g\backslash n$ and $g\backslash[\mathcal{P}_g(n)]$ respectively. We have to show that both labelled quotient trees coincide.

The equalities $P_1 = P_2$ and $l_1 = l_2$, follow immediately from the characterisations in Lemma 4.5 and Definition A.6. For the equality $\sim_1 = \sim_2$, we can reason as follows:

$$
\begin{aligned}
& \pi_1 \sim_1 \pi_2 \\
\iff\ & \pi_1 \sim_g \pi_2 \text{ and } \pi_1, \pi_2 \in P_1 \\
\iff\ & \pi_1, \pi_2 \in \mathcal{P}_g(m) \cap P_1 \text{ for some } m \in N^g \\
\iff\ & \pi_1, \pi_2 \in \mathcal{P}_g(n) \cap P_1 \quad \text{or} \quad \pi_1, \pi_2 \in \mathcal{P}_g(m) \cap P_1 \text{ for some } m \in N^g \setminus \{n\} \\
\overset{P_1=P_2}{\iff}\ & \pi_1, \pi_2 \in \mathcal{P}_g(n) \cap P_2 \quad \text{or} \quad \pi_1, \pi_2 \in \mathcal{P}_g(m) \cap P_2 \text{ for some } m \in N^g \setminus \{n\} \\
\iff\ & \pi_1 \sim_g \pi_2 \text{ with } \pi_1, \pi_2 \in \mathcal{P}_g(n) \cap P_2 \quad \text{or} \\
& \pi_1 \sim_g \pi_2 \text{ with } \pi_1, \pi_2 \in P_2 \setminus \mathcal{P}_g(n) \\
\iff\ & \pi_1 \sim_2 \pi_2
\end{aligned}
$$

□

The following two lemmas show that the positional truncation of the initial and the result term graph of a pre-reduction step at the positions of the root of the redex are isomorphic.

Lemma A.11. *Let $g \mapsto_n h$ be a pre-reduction step in a GRS. Then $\mathcal{P}_g(n)$ is admissible for truncating h.*

Proof. Let $\pi_1 \sim_h \pi_2$ such that no prefix of π_1 or π_2 is in $\mathcal{P}_g(n)$ and let $\pi_1 \cdot \langle i \rangle \in \mathcal{P}_g(n)$. We have to show that then $\pi_2 \cdot \langle i \rangle \in \mathcal{P}_g(n)$, too. Since no prefix of π_1 or π_2 is a position of n in g, both π_1 and π_2 are unaffected by the pre-reduction step and thus each of them passes the same nodes in g as it does in h. Consequently, $\pi_1 \sim_h \pi_2$ implies $\pi_1 \sim_g \pi_2$. Since $\pi_1 \cdot \langle i \rangle \in \mathcal{P}_g(n)$, this means that also $\pi_2 \cdot \langle i \rangle \in \mathcal{P}_g(n)$. □

Lemma A.12. *Let $g \mapsto_n h$ be a pre-reduction step. Then $g\backslash[\mathcal{P}_g(n)] \cong h\backslash[\mathcal{P}_g(n)]$.*

Proof. Let $(P_1, l_1, \sim_1)$ and $(P_2, l_2, \sim_2)$ be the labelled quotient trees of $g\backslash[\mathcal{P}_g(n)]$ respectively $h\backslash[\mathcal{P}_g(n)]$. We will show that both labelled quotient trees coincide.

To show that $P_1 \subseteq P_2$, let $\pi \in P_1$. This means that $\pi' \notin \mathcal{P}_g(n)$ for all $\pi' < \pi$. Consequently, no proper prefix of π is affected by the pre-reduction step and thus $\pi \in P_2$. The inclusion $P_2 \subseteq P_1$ follows likewise.

Let $Q = \mathcal{P}_g(n)$. Due to the equality $P_1 = P_2$, the sets

$$Q^+ = \mathcal{P}_g^m(n) \text{ and } Q^- = \{\pi \in \mathcal{P}(g) \mid \forall \pi' \leq \pi . \pi' \notin \mathcal{P}_g(n)\}$$

are the same for both positional truncations. For the equality $l_1 = l_2$, note that according to the argument above, none of the nodes at positions in Q^- in g are

affected by the pre-reduction step. Hence, we have $l_1(\pi) = g(\pi) = h(\pi) = l_2(\pi)$ if $\pi \in Q^-$ and $l_1(\pi) = \bot = l_2(\pi)$ if $\pi \in Q^+$.

For the equality $\sim_1 = \sim_2$, assume that $\pi_1, \pi_2 \in Q^+$ or $\pi_1, \pi_2 \in Q^-$. In the first case, both π_1 and π_2 are positions of the root of the redex and the root of the reduct. In the second case, both π_1 and π_2 are unaffected by the pre-reduction step. In either case, we have the following:

$$\begin{aligned}\pi_1 \sim_1 \pi_2 &\iff \pi_1 \sim_g \pi_2 \iff \mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)\\ &\iff \mathsf{node}_h(\pi_1) = \mathsf{node}_h(\pi_2) \iff \pi_1 \sim_h \pi_2 \iff \pi_1 \sim_2 \pi_2 \qquad \square\end{aligned}$$

We can use the above findings to obtain that the reduction context is preserved through reduction steps:

Lemma A.13 (preservation of reduction contexts)**.** *Given a reduction step $g \to_n h$, we have $g\backslash n \cong h\backslash[\mathcal{P}_g(n)]$.*

Proof. Given a reduction step $g \to_n h$, there must be a pre-reduction step $g' \mapsto_{n'} h'$ with $g = \mathcal{C}(g')$, $h = \mathcal{C}(h')$ and $n = \mathcal{P}_{g'}(n')$. We then obtain the following isomorphisms:

$$g\backslash n \overset{(1)}{\cong} g\backslash[\mathcal{P}_g(n)] \overset{(2)}{\cong} g'\backslash[\mathcal{P}_{g'}(n')] \overset{(3)}{\cong} h'\backslash[\mathcal{P}_{g'}(n')] \overset{(4)}{\cong} h\backslash[\mathcal{P}_g(n)]$$

The well-definedness of the above positional truncations follows from Lemma A.9, for the first two positional truncation, by Lemma A.11, for the third one, respectively by the fact that $h \cong h'$ and $\mathcal{P}_g(n) = \mathcal{P}_{g'}(n')$, for the last one. Isomorphism (1) follows from Lemma A.10, isomorphism (2) from $g \cong g'$ and $\mathcal{P}_g(n) = \mathcal{P}_{g'}(n')$, Isomorphism (3) from Lemma A.12, and Isomorphism (4) from $h \cong h'$ and $\mathcal{P}_g(n) = \mathcal{P}_{g'}(n')$. $\square$

Finally, we can put everything together to prove Proposition 4.7.

Proposition 4.7. *Given a reduction step $g \to_n h$, we have $g\backslash n \leq^{\mathsf{S}}_{\bot} g, h$.*

Proof of Proposition 4.7. From Corollary 4.6, we immediately obtain the inequality $g\backslash n \leq^{\mathsf{S}}_{\bot} g$. Since, by Lemma A.13, $g\backslash n \cong h\backslash[\mathcal{P}_g(n)]$ and, by Corollary A.8, $h\backslash[\mathcal{P}_g(n)] \leq^{\mathsf{S}}_{\bot} h$, we can conclude that $g\backslash n \leq^{\mathsf{S}}_{\bot} h$. $\square$

A.3 Strong Convergence

A.3.1 Auxiliary Lemmas

The following technical lemma confirms the intuition that changes during a continuous reduction must be caused by a reduction step that was applied at the position where the difference is observed or above.

Lemma A.14. *Let $(g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be a strongly p-continuous reduction in a GRS with its reduction contexts $c_\iota = \mathcal{C}(g_\iota\backslash n_\iota)$ such that there are $\beta \leq \gamma < \alpha$ and $\pi \in \mathcal{P}(c_\beta) \cap \mathcal{P}(c_\gamma)$ with $c_\beta(\pi) \neq c_\gamma(\pi)$. Then there is a position $\pi' \leq \pi$ and an index $\beta \leq \iota \leq \gamma$ such that $\pi' \in \mathcal{P}_{g_\iota}(n_\iota)$.*

Proof. Given a reduction and β, γ and π as stated above, we can assume that

$$g_\iota(\pi) = c_\iota(\pi) \qquad \text{if } \beta \leq \iota \leq \gamma \text{ and } \pi \in \mathcal{P}(c_\iota). \tag{$*$}$$

If this would not be the case, then, by Lemma 4.5, $\pi \in \mathcal{P}_{g_\iota}(n_\iota)$, i.e. the statement that we want to prove already holds.

We proceed with an induction on γ. The case $\gamma = \beta$ is trivial.

Let $\gamma = \iota + 1 > \beta$ and $c'_\iota = g_\gamma \backslash [\mathcal{P}_{g_\iota}(n_\iota)]$. Note that since by assumption $\pi \in \mathcal{P}(c_\gamma)$, we also have that $\pi \in \mathcal{P}(g_\gamma)$, according to Lemma 4.5. Moreover, we can assume that $\pi \in \mathcal{P}(c'_\iota)$ since otherwise $\pi \in \mathcal{P}(g_\gamma)$ already implies that $\pi' \in \mathcal{P}_{g_\iota}(n_\iota)$ for some $\pi' < \pi$. According to Lemma A.13, $\pi \in \mathcal{P}(c'_\iota)$ implies that $\pi \in \mathcal{P}(c_\iota)$, too. Hence, we can assume that $c_\beta(\pi) = c_\iota(\pi)$ since otherwise the proof goal follows immediately from the induction hypothesis. We can thus reason as follows:

$$c'_\iota(\pi) \overset{\text{Lem. A.13}}{=} c_\iota(\pi) = c_\beta(\pi) \neq c_\gamma(\pi) \overset{(*)}{=} g_\gamma(\pi)$$

From the thus obtained inequality $c'_\iota(\pi) \neq g_\gamma(\pi)$ we can derive that $\pi \in \mathcal{P}_{g_\iota}(n_\iota)$.

Let γ be a limit ordinal. By $(*)$, we know that $g_\gamma(\pi) = c_\gamma(\pi) \neq c_\beta(\pi)$. According to Theorem 3.2, the inequality $g_\gamma(\pi) \neq c_\beta(\pi)$ is only possible if there is a $\beta \leq \iota < \gamma$ such that $c_\iota(\pi) \neq c_\beta(\pi)$. Hence, we can invoke the induction hypothesis, which immediately yields the proof goal. □

By combining the characterisation of the limit inferior from Theorem 3.2 and the characterisation of local truncations from Lemma 4.5, we obtain the following characterisation of the limit of a strongly p-convergent reduction:

Lemma A.15. *Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open reduction in a GRS strongly p-converging to g. Then g has the following labelled quotient tree $(P, l, \sim)$:*

$$P = \bigcup_{\beta<\alpha} \{\pi \in \mathcal{P}(g_\beta) \mid \forall \pi' < \pi \forall \beta \leq \iota < \alpha \colon \pi' \notin \mathcal{P}_{g_\iota}(n_\iota)\}$$

$$\sim = \left(\bigcup_{\beta<\alpha} \bigcap_{\beta\leq\iota<\alpha} \sim_{g_\iota} \right) \cap P \times P$$

$$l(\pi) = \begin{cases} g_\beta(\pi) & \text{if } \exists \beta < \alpha \forall \beta \leq \iota < \alpha \colon \pi \notin \mathcal{P}_{g_\iota}(n_\iota) \\ \bot & \text{otherwise} \end{cases} \qquad \text{for all } \pi \in P$$

Proof. Let $c_\iota = \mathcal{C}(g_\iota \backslash n_\iota)$ for each $\iota < \alpha$. We will show in the following that $(P, l, \sim)$ is equal to $(\mathcal{P}(g), g(\cdot), \sim_g)$.

At first we show that $\mathcal{P}(g) \subseteq P$. To this end let $\pi \in \mathcal{P}(g)$. Since $g = \liminf_{\iota\to\alpha} c_\iota$, this means, by Theorem 3.2, that there is some $\beta < \alpha$ such that

$$\pi \in \mathcal{P}(c_\beta) \text{ and } c_\iota(\pi') = c_\beta(\pi') \text{ for all } \pi' < \pi \text{ and } \beta \leq \iota < \alpha. \tag{1}$$

Since, according to Lemma 4.5, $\mathcal{P}(c_\beta) \subseteq \mathcal{P}(g_\beta)$, we also have $\pi \in \mathcal{P}(g_\beta)$. In order to prove that $\pi \in P$, we assume some $\pi' < \pi$ and $\beta \leq \iota < \alpha$ and show that $\pi' \notin \mathcal{P}_{g_\iota}(n_\iota)$. Since π' is a proper prefix of a position in c_β, we have that $c_\beta(\pi') \in \Sigma$. By (1), also $c_\iota(\pi') \in \Sigma$. Hence, according to Lemma 4.5, $\pi' \notin \mathcal{P}_{g_\iota}(n_\iota)$.

For the converse direction $P \subseteq \mathcal{P}(g)$, we assume some $\pi \in P$ and show that then $\pi \in \mathcal{P}(g)$. Since $\pi \in P$, we have that there is some $\beta < \alpha$ with

$$\pi \in \mathcal{P}(g_\beta) \text{ and } \pi' \notin \mathcal{P}_{g_\iota}(n_\iota) \text{ for all } \pi' < \pi \text{ and } \beta \leq \iota < \alpha. \tag{2}$$

In particular, we have that $\pi' \notin \mathcal{P}_{g_\beta}(n_\beta)$ for all $\pi' < \pi$. Hence, by Lemma 4.5, $\pi \in \mathcal{P}(c_\beta)$. According to Theorem 3.2, it remains to be shown that $c_\gamma(\pi') = c_\beta(\pi')$ for all $\pi' < \pi$ and $\beta \leq \gamma < \alpha$. We will do that by an induction on γ:

The case $\gamma = \beta$ is trivial. For $\gamma = \iota + 1 > \beta$, let $g_\iota \to_{n_\iota} g_\gamma$ be the ι-th reduction step, $c'_\iota = g_\gamma \backslash [\mathcal{P}_{g_\iota}(n_\iota)]$ and $\pi' < \pi$. We can then reason as follows:

$$c_\beta(\pi') \overset{\text{ind. hyp.}}{=} c_\iota(\pi') \overset{\text{Lem. A.13}}{=} c'_\iota(\pi') \overset{(*)}{=} g_\gamma(\pi') \overset{\text{Lem. 4.5}}{=} g_\gamma \backslash n_\gamma(\pi') = c_\gamma(\pi')$$

The equality $(*)$ above is justified by the fact that $\pi' < \pi \in \mathcal{P}(c_\beta)$ and thus $c_\beta(\pi') \in \Sigma$. The application of Lemma 4.5 is justified by (2).

If $\gamma > \beta$ is a limit ordinal, then $g_\gamma = \liminf_{\iota \to \gamma} c_\iota$. Since $\pi' \in \mathcal{P}(c_\beta)$ and, by induction hypothesis, $c_\iota(\pi'') = c_\beta(\pi'')$ for all $\pi'' \leq \pi'$, $\beta \leq \iota < \gamma$, we obtain, by Theorem 3.2, that $g_\gamma(\pi') = c_\beta(\pi')$. Since, according to (2), $\pi'' \notin \mathcal{P}_{g_\gamma}(n_\gamma)$ for each $\pi'' \leq \pi'$, we have by Lemma 4.5 that $g_\gamma(\pi') = c_\gamma(\pi')$. Hence, $c_\gamma(\pi') = c_\beta(\pi')$.

The inclusion $\sim_g \subseteq \sim$ follows from Theorem 3.2 and the equality $P = \mathcal{P}(g)$ since $\sim_{c_\iota} \subseteq \sim_{g_\iota}$ for all $\iota < \alpha$ according to Lemma 4.5.

For the reverse inclusion $\sim \subseteq \sim_g$, assume that $\pi_1 \sim \pi_2$. That is, $\pi_1, \pi_2 \in P$ and there is some $\beta_0 < \alpha$ such that $\pi_1 \sim_{g_\iota} \pi_2$ for all $\beta_0 \leq \iota < \alpha$. Since $\pi_1, \pi_2 \in P = \mathcal{P}(g)$, we know, by Theorem 3.2, that there are $\beta_1, \beta_2 < \alpha$ such that $\pi_k \in \mathcal{P}(c_\iota)$ for all $\beta_k \leq \iota < \alpha$, $k \in \{1, 2\}$. Let $\beta = \max\{\beta_0, \beta_1, \beta_2\}$. For each $\beta \leq \iota < \alpha$, we then obtain that $\pi_1 \sim_{g_\iota} \pi_2$ and $\pi_1, \pi_2 \in \mathcal{P}(c_\iota)$. By Lemma 4.5, this is equivalent to $\pi_1 \sim_{c_\iota} \pi_2$. Applying Theorem 3.2 then yields $\pi_1 \sim_g \pi_2$.

Finally, we show that $l = g(\cdot)$. To this end, let $\pi \in P$. We distinguish two mutually exclusive cases. For the first case, we assume that

$$\text{there is some } \beta < \alpha \text{ such that } c_\iota(\pi) = c_\beta(\pi) \text{ for all } \beta \leq \iota < \alpha. \tag{3}$$

By Theorem 3.2, we know that then $g(\pi) = c_\beta(\pi)$. Next, assume that there is some $\beta' < \alpha$ with $\pi \notin \mathcal{P}_{g_\iota}(n_\iota)$ for all $\beta' \leq \iota < \alpha$. W.l.o.g. we can assume that $\beta = \beta'$. Hence, $l(\pi) = g_\beta(\pi)$. Moreover, since $\pi \notin \mathcal{P}_{g_\beta}(n_\beta)$, we have that $g_\beta(\pi) = c_\beta(\pi)$ according to Lemma 4.5. We thus conclude that $l(\pi) = g_\beta(\pi) = c_\beta(\pi) = g(\pi)$. Now assume there is no such β', i.e. for each $\beta' < \alpha$ there is some $\beta' \leq \iota < \alpha$ with $\pi \in \mathcal{P}_{g_\iota}(n_\iota)$. Consequently, $l(\pi) = \bot$ and, by Lemma 4.5, we have for each $\beta' < \alpha$ some $\beta' \leq \iota < \alpha$ such that $c_\iota(\pi) = \bot$. According to (3), the latter implies that $c_\iota(\pi) = \bot$ for all $\beta \leq \iota < \alpha$. By Theorem 3.2, we thus obtain that $g(\pi) = \bot = l(\pi)$.

Next, we consider the negation of (3), i.e. that

$$\begin{gathered}\text{for all } \beta < \alpha \text{ there is a } \beta \leq \iota < \alpha \text{ such that} \\ \pi \in \mathcal{P}(c_\iota) \cap \mathcal{P}(c_\beta) \text{ implies } c_\iota(\pi) \neq c_\beta(\pi).\end{gathered} \tag{4}$$

By Theorem 3.2, we have that $g(\pi) = \bot$. Since $\pi \in P = \mathcal{P}(g)$, we can apply Theorem 3.2 again to obtain a $\gamma < \alpha$ with $\pi \in \mathcal{P}(c_\iota)$ and $c_\iota(\pi') = c_\gamma(\pi')$ for all $\pi' < \pi$ and $\gamma \leq \iota < \alpha$. Combining this with (4) yields that for each $\gamma \leq \beta < \alpha$

there is a $\beta \leq \iota < \alpha$ with $c_\iota(\pi) \neq c_\beta(\pi)$. According to Lemma A.14, this can only happen if there is a $\beta \leq \gamma' \leq \iota$ and a $\pi' \leq \pi$ such that $\pi' \in \mathcal{P}_{g_{\gamma'}}(n_{\gamma'})$. Since π has only finitely many prefixes, we can apply the infinite pigeon hole principle to obtain a single prefix $\pi' \leq \pi$ such that for each $\beta < \alpha$ there is some $\beta \leq \iota < \alpha$ with $\pi' \in \mathcal{P}_{g_\iota}(n_\iota)$. However, π' cannot be a proper prefix of π since this would imply that $\pi \notin P$. Thus we can conclude that for each $\beta < \alpha$ there is some $\beta \leq \iota < \alpha$ such that $\pi \in \mathcal{P}_{g_\iota}(n_\iota)$. Hence, $l(\pi) = \bot = g(\pi)$. □

In order to compare strong m- and p-convergence, we consider positions bounded by a certain depth.

Definition A.16 (bounded positions)**.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $d \in \mathbb{N}$. We write $\mathcal{P}_{\leq d}(g)$ for the set $\{\pi \in \mathcal{P}(g) \mid |\pi| \leq d\}$ of positions in g of length at most π.

Positional truncations do not change positions bounded by the same depth or above:

Lemma A.17. *Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$, Q admissible for truncating g and $d \in \mathbb{N}$ such that $d \leq \min\{|\pi| \mid \pi \in Q\}$. Then $\mathcal{P}_{\leq d}(g\backslash[Q]) = \mathcal{P}_{\leq d}(g)$.*

Proof. $\mathcal{P}_{\leq d}(g\backslash[Q]) \subseteq \mathcal{P}_{\leq d}(g)$ follows immediately from the definition of $g\backslash[Q]$. For the converse inclusion, assume some $\pi \in \mathcal{P}_{\leq d}(g)$. Since we then have that $|\pi| \leq d \leq \min\{|\pi| \mid \pi \in Q\}$, we know for each $\pi' < \pi$ that $|\pi'| < \min\{|\pi| \mid \pi \in Q\}$ and thus $\pi' \notin Q$. Consequently, π is in $\mathcal{P}(g\backslash[Q])$ and, therefore, also in $\mathcal{P}_{\leq d}(g\backslash[Q])$. □

From this we immediately obtain the analogous property for local truncations:

Corollary A.18. *If $g \in \mathcal{G}^\infty(\Sigma_\bot)$, $n \in N^G$ and $d \leq \mathsf{depth}_g(n)$, then $\mathcal{P}_{\leq d}(g\backslash n) = \mathcal{P}_{\leq d}(g)$.*

Proof. This follows from Lemma A.17 since $d \leq \mathsf{depth}_g(n)$ implies that $d \leq \min\{|\pi| \mid \pi \in \mathcal{P}_g(n)\}$, and $g\backslash n \cong g\backslash[\mathcal{P}_g(n)]$, according to Lemma A.10. □

Additionally, reductions that only contract redexes at a depth $\geq d$ do not affect the positions bounded by d.

Lemma A.19. *Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be a strongly p-convergent reduction in a GRS and $d \in \mathbb{N}$ such that $\mathsf{depth}_{g_\iota}(n_\iota) \geq d$ for all $\iota < \alpha$. Then $\mathcal{P}_{\leq d}(g_0) = \mathcal{P}_{\leq d}(g_\iota)$ for all $\iota \leq \alpha$.*

Proof. We prove the statement by an induction on α. The case $\alpha = 0$ is trivial.

Let $\alpha = \beta + 1$. Due to the induction hypothesis, it suffices to show that $\mathcal{P}_{\leq d}(g_0) = \mathcal{P}_{\leq d}(g_\alpha)$:

$$\begin{aligned}\mathcal{P}_{\leq d}(g_0) &\overset{\text{ind. hyp.}}{=} \mathcal{P}_{\leq d}(g_\beta) \overset{\text{Cor. A.18}}{=} \mathcal{P}_{\leq d}(g_\beta\backslash n_\beta)\\ &\overset{\text{Lem. A.13}}{=} \mathcal{P}_{\leq d}(g_\alpha\backslash\left[\mathcal{P}_{g_\beta}(n_\beta)\right]) \overset{\text{Lem. A.17}}{=} \mathcal{P}_{\leq d}(g_\alpha)\end{aligned}$$

The application of Lemma A.17 is justified since we have that

$$d \leq \mathsf{depth}_{g_\beta}(n_\beta) = \min\left\{|\pi| \,\middle|\, \pi \in \mathcal{P}_{g_\beta}(n_\beta)\right\}.$$

Lastly, let α be a limit ordinal. By the induction hypothesis, we only need to show $\mathcal{P}_{\leq d}(g_0) = \mathcal{P}_{\leq d}(g_\alpha)$. At first assume that $\pi \in \mathcal{P}_{\leq d}(g_\alpha)$. Hence, by Lemma A.15, there is some $\beta < \alpha$ such that $\pi \in \mathcal{P}(g_\beta)$. Therefore, π is in $\mathcal{P}_{\leq d}(g_\beta)$ and, by induction hypothesis, also in $\mathcal{P}_{\leq d}(g_0)$. Conversely, assume that $\pi \in \mathcal{P}_{\leq d}(g_0)$. Because $\mathsf{depth}_{g_\iota}(n_\iota) \geq d$ for all $\iota < \alpha$, we have that $\pi' \notin \mathcal{P}_{g_\iota}(n_\iota)$ for all $\pi' < \pi$ and $\iota < \alpha$. According to Lemma A.15, this implies that π is in $\mathcal{P}(g_\alpha)$ and thus also in $\mathcal{P}_{\leq d}(g_\alpha)$. □

A.3.2 Proof of Lemma 4.11

Lemma 4.11. *Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open reduction in a GRS $\mathcal{R}_\perp$. If S strongly p-converges to a total term graph, then $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity.*

Proof of Lemma 4.11. Let S be strongly p-converging to g. We will show that whenever the sequence $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ does not tend to infinity, then g is not total. If $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ does not tend to infinity, then there is some $d \in \mathbb{N}$ such that for each $\gamma < \alpha$ there is a $\gamma \leq \iota < \alpha$ with $\mathsf{depth}_{g_\iota}(n_\iota) \leq d$. Let d^* be the smallest such d. Hence, there is a $\beta < \alpha$ such that $\mathsf{depth}_{g_\iota}(n_\iota) \geq d^*$ for all $\beta \leq \iota < \alpha$. Thus we can apply Lemma A.19 to the suffix of S starting from β to obtain that $\mathcal{P}_{\leq d^*}(g_\beta) = \mathcal{P}_{\leq d^*}(g_\iota)$ for all $\beta \leq \iota < \alpha$. Note that according to Lemma A.15, this implies that $\mathcal{P}_{\leq d^*}(g_\beta) \subseteq \mathcal{P}(g)$. Moreover, since we find for each $\gamma < \alpha$ some $\gamma \leq \iota < \alpha$ with $\mathsf{depth}_{g_\iota}(n_\iota) \leq d^*$, we know that for each $\gamma < \alpha$ there is a $\gamma \leq \iota < \alpha$ and a $\pi \in \mathcal{P}_{\leq d^*}(g_\beta)$ with $\pi \in \mathcal{P}_{g_\iota}(n_\iota)$. Because $\mathcal{P}_{\leq d^*}(g_\beta)$ is finite, the infinite pigeon hole principle yields a single $\pi^* \in \mathcal{P}_{\leq d^*}(g_\beta)$ such that for each $\gamma < \alpha$ there is a $\gamma \leq \iota < \alpha$ with $\pi^* \in \mathcal{P}_{g_\iota}(n_\iota)$. Since we know that $\pi^* \in \mathcal{P}(g)$, this means, according to Lemma A.15, that $g(\pi^*) = \perp$, i.e. g is not total. □

A.3.3 Proof of Lemma 4.12

Lemma 4.12. *Let $S = (g_\iota \to_{n_\iota} g_{\iota+1})_{\iota<\alpha}$ be an open reduction in a GRS $\mathcal{R}_\perp$ that strongly p-converges to g. If $(g_\iota)_{\iota<\alpha}$ is Cauchy and $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity, then $g \cong \lim_{\iota\to\alpha} g_\iota$.*

Proof of Lemma 4.12. Let $h = \lim_{\iota\to\alpha} g_\iota$ and let $(c_\iota)_{\iota<\alpha}$ be the reduction contexts of S. We will prove that $g \cong h$ by showing that their respective labelled quotient trees coincide.

For the inclusion $\mathcal{P}(g) \subseteq \mathcal{P}(h)$, assume some $\pi \in \mathcal{P}(g)$. According to Theorem 3.2, there is some $\beta < \alpha$ such that $\pi \in \mathcal{P}(c_\beta)$ and $c_\iota(\pi) = c_\beta(\pi)$ for all $\pi' < \pi$ and $\beta \leq \iota < \alpha$. Thus, $\pi \in \mathcal{P}(c_\iota)$ for all $\beta \leq \iota < \alpha$. Since $c_\iota \cong g_\iota \backslash n_\iota$ and, therefore $\mathcal{P}(c_\iota) \subseteq \mathcal{P}(g_\iota)$ by Lemma 4.5, we have that $\pi \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$. This implies, by Theorem 3.3, that $\pi \in \mathcal{P}(h)$.

For the converse inclusion $\mathcal{P}(h) \subseteq \mathcal{P}(g)$, assume some $\pi \in \mathcal{P}(h)$. According to Theorem 3.3, there is some $\beta < \alpha$ such that $\pi \in \mathcal{P}(g_\iota)$ for all $\beta \leq \iota < \alpha$. Since $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity, we find some $\beta \leq \gamma < \alpha$ such that $\mathsf{depth}_{g_\iota}(n_\iota) \geq |\pi|$ for all $\gamma \leq \iota < \alpha$, i.e. $\pi' \notin \mathcal{P}_{g_\iota}(n_\iota)$ for all $\pi' < \pi$. This means, by Lemma A.15, that $\pi \in \mathcal{P}(g)$.

By Lemma A.15 and Theorem 3.3, $\sim_g = \sim_h$ follows from the equality $\mathcal{P}(g) = \mathcal{P}(h)$.

In order to show the equality $g(\cdot) = h(\cdot)$, assume some $\pi \in \mathcal{P}(h)$. According to Theorem 3.3, there is some $\beta < \alpha$ such that $h(\pi) = g_\iota(\pi)$ for all $\beta \leq \iota < \alpha$. Additionally, since $(\mathsf{depth}_{g_\iota}(n_\iota))_{\iota<\alpha}$ tends to infinity, there is some $\beta \leq \gamma < \alpha$ such that $\mathsf{depth}_{g_\iota}(n_\iota) > |\pi|$ for all $\gamma \leq \iota < \alpha$. As this means that $\pi' \notin \mathcal{P}_{g_\iota}(n_\iota)$ for all $\pi' \leq \pi$ and $\gamma \leq \iota < \alpha$, we obtain, by Lemma A.15, that $g(\pi) = g_\gamma(\pi)$. Since $h(\pi) = g_\gamma(\pi)$, we can conclude that $g(\pi) = h(\pi)$. □

A.4 Normalisation of Strong p-convergence

Lemma 4.16. *Let $\mathcal{R}$ be a GRS over Σ and $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\perp)$.*

(i) If g is root-active, then $g \overset{p}{\twoheadrightarrow}_{\mathcal{R}} \perp$.

(ii) If g is not root-active, then there is a reduction $g \overset{p}{\twoheadrightarrow}_{\mathcal{R}} h$ to a root-stable term graph h.

(iii) If g is root-stable, then so is every h with $g \overset{p}{\twoheadrightarrow}_{\mathcal{R}} h$.

Proof of Lemma 4.16. (i) At first, we will show that, for each root-active term graph g, we find a reduction $g \overset{p}{\twoheadrightarrow} g'$ and a reduction step $g' \to_c h$ with $c = \perp$ and h a root-active term graph. Since g is root-active, there is a reduction $S\colon g \overset{p}{\twoheadrightarrow} g'$ with g' a redex. Hence, there is a reduction step $\phi\colon g' \to_c h$ applied at the root node $r^{g'}$. That is, $c \cong g'\backslash r^{g'} \cong \perp$. To see that h is root active, let $T\colon h \overset{p}{\twoheadrightarrow} h'$. Then $S \cdot \langle\phi\rangle \cdot T\colon g \overset{p}{\twoheadrightarrow} h'$. Since g is root-active we find a reduction $h' \overset{p}{\twoheadrightarrow} h''$ to a redex h''. Hence, h is root-active.

Given a root-active term g_0, we obtain with the above finding, for each $i < \omega$, a reduction $S_i\colon g_i \overset{p}{\twoheadrightarrow} g_i'$ and a reduction step $\phi_i\colon g_i' \to_{c_i} g_{i+1}$ with $c_i = \perp$. Then the open sequence $S = \prod_{i<\omega} S_i \cdot \langle\phi_i\rangle$ is a strongly p-continuous reduction starting from g_0. Thus, according to Proposition 4.10, there is a term graph g_ω with $S\colon g_0 \overset{p}{\twoheadrightarrow} g_\omega$. That is, if $(\widehat{c}_\iota)_{\iota<|S|}$ is the sequence of reduction contexts of S, we have $g_\omega = \liminf_{\iota\to|S|} \widehat{c}_\iota$. Due to the construction of S, we find, for each $\alpha < |S|$, a $\alpha \leq \beta < |S|$ with $\widehat{c}_\beta = \perp$. Hence, $g_\omega = \perp$, which means that $S\colon g_0 \overset{p}{\twoheadrightarrow} \perp$.

(ii) If g is not root-active, then there has to be a reduction $g \overset{p}{\twoheadrightarrow} h$ such that no reduction starting from h strongly p-converges to a redex. That is, h is root-stable.

(iii) Let g be a root-stable term graph and $S\colon g \overset{p}{\twoheadrightarrow} h$. Given a reduction $T\colon h \overset{p}{\twoheadrightarrow} h'$, we have to show that h' is not a redex. Since g is root-stable and $S \cdot T\colon g \overset{p}{\twoheadrightarrow} h'$, we know that h' is not a redex. □

A.5 Soundness of Strong p-convergence

Lemma 5.11. *Let $(a_\iota)_{\iota<\alpha}$ be a sequence in a complete semilattice $(A, \leq)$ and $(\gamma_\iota)_{\iota<\delta}$ a strictly monotone sequence in the ordinal α such that $\bigsqcup_{\iota<\delta} \gamma_\iota = \alpha$. Then*

$$\liminf_{\iota\to\alpha} a_\iota = \liminf_{\beta\to\delta} \left(\bigsqcap_{\gamma_\beta \leq \iota < \gamma_{\beta+1}} a_\iota\right).$$

Proof of Lemma 5.11. At first we show that

$$\bigsqcap_{\beta\leq\beta'<\delta}\left(\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota\right) = \bigsqcap_{\gamma_\beta\leq\iota<\alpha} a_\iota \qquad \text{for all } \beta<\delta \tag{$*$}$$

by assuming an arbitrary ordinal $\beta<\delta$ and using the antisymmetry property of the partial order $\leq$ on A.

Since, for all $\beta\leq\beta'<\delta$, we have that $\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota \geq \bigsqcap_{\gamma_\beta\leq\iota<\alpha} a_\iota$, we obtain that $\bigsqcap_{\beta\leq\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota \geq \bigsqcap_{\gamma_\beta\leq\iota<\alpha} a_\iota$.

On the other hand, since $(\gamma_\iota)_{\iota<\delta}$ is strictly monotone and $\bigsqcup_{\iota<\delta}\gamma_\iota=\alpha$, we find for each $\gamma_\beta\leq\gamma<\alpha$ some $\beta\leq\beta'<\delta$ such that $\gamma_{\beta'}\leq\gamma<\gamma_{\beta'+1}$ and, thus, $\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota \leq a_\gamma$. Therefore, we obtain that $\bigsqcap_{\beta\leq\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota \leq a_\gamma$ for all $\gamma_\beta\leq\gamma<\alpha$. Hence, we can conclude that $\bigsqcap_{\beta\leq\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\gamma_{\beta'+1}} a_\iota \leq \bigsqcap_{\gamma_\beta\leq\iota<\alpha} a_\iota$.

With the thus obtained equation $(*)$, it remains to be shown that

$$\bigsqcup_{\beta<\alpha}\bigsqcap_{\beta\leq\iota<\alpha} a_\iota = \bigsqcup_{\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota.$$

Again, we use the antisymmetry of $\leq$.

Since $\bigsqcap_{\iota<\delta}\gamma_\iota=\alpha$, we find for each $\beta<\alpha$ some $\beta'<\delta$ with $\gamma_{\beta'}\geq\beta$. Consequently, we have for each $\beta<\alpha$ some $\beta'<\delta$ with $\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \leq \bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$. Hence, we know that $\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \leq \bigsqcup_{\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$ for all $\beta<\alpha$, which means that $\bigsqcup_{\beta<\alpha}\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \leq \bigsqcup_{\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$.

On the other hand, since for each $\beta'<\delta$ there is a $\beta<\alpha$ (namely $\beta=\gamma_\beta$) with $\bigsqcap_{\beta\leq\iota<\alpha} a_\iota = \bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$, we also have $\bigsqcup_{\beta<\alpha}\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \geq \bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$ for all $\beta'<\delta$. Consequently, we have $\bigsqcup_{\beta<\alpha}\bigsqcap_{\beta\leq\iota<\alpha} a_\iota \geq \bigsqcup_{\beta'<\delta}\bigsqcap_{\gamma_{\beta'}\leq\iota<\alpha} a_\iota$. □