



# Proceedings of the 1st DIKU-IST Joint Workshop on Foundations of Software

Robert Glück and Zhenjiang Hu (Eds.)

Technical Report no. 05/07  
ISSN: 0107-8283

**HCØ Tryk**

University of Copenhagen • Universitetsparken 1  
DK-2100 Copenhagen • Denmark



# Preface

These proceedings contain the contributions presented at the *1st DIKU-IST Joint Workshop on Foundations of Software* held at Dragør Badehotel, Denmark, September 23-24, 2005. This workshop featured talks and discussions on the theory and practice of automatic software production, programming language technologies, and the safety and robustness of critical software components.

The workshop aimed to provide a forum for exchanging research ideas and promoting research collaboration on foundations of software between the *Department of Computer Science, University of Copenhagen* (DIKU), and the *Graduate School of Information Science and Technology, University of Tokyo* (IST). Until now there has been a long and continuing, but only intermittent, series of informal contacts between researchers at both institutions. Our objective is to create a closer collaboration between the two groups of researchers, postdocs, and graduate students. In 2004, IST at Tokyo University and the Faculty of Science at the University of Copenhagen entered a *Five-Year Academic and Student Exchange Agreement* during the State Visit of Queen Margrethe II to Japan.

Software has become a decisive factor for commercial success in many areas of modern technology and business development. Despite tremendous progress in hardware, the production of software is still manual, error prone, and costly. The exploding demand for software has led to a worldwide undersupply of skilled programmers, outsourcing, and to low software quality, which is evident to anyone today who uses software. The workshop invited contributions on all topics related to the foundations of software.

The workshop had about 40 participants from Japan and Denmark. The organizers would like to thank all speakers, participants, and the local organizers for making this meeting both a successful and enjoyable event. Special thanks to Jørgen Olsen, Pro-vice Chancellor of the University of Copenhagen, and to Miyoko Akashi, Minister Counsellor at the Embassy of Japan in Denmark, for opening the workshop, and to Stig Skelboe, Department Chairman of DIKU, and Masato Takeichi, Dean of IST, for presenting their institutions.

Copenhagen and Tokyo, November 2005

Robert Glück  
Zhenjiang Hu

# Organization

The *DIKU-IST Joint Workshop on Foundations of Software* was organized by the Department of Computer Science, University of Copenhagen, together with the Graduate School of Information Science and Technology, University of Tokyo.

## Meeting

Program committee: Robert Glück (co-chair)  
Masami Hagiya  
Zhenjiang Hu (co-chair)  
Neil Jones  
Masato Takeichi

Local arrangements: Jesper Andersen  
Morten Fjord-Larsen  
Karin Outzen  
Thomas Pecseli  
Naja Villien

## Sponsoring Institutions

The workshop was sponsored by The Danish Research Agency, Forskningsrådet for Natur og Univers (FNU), of the Danish Ministry of Science, Technology and Innovation. We gratefully acknowledge the support of the Department of Computer Science, University of Copenhagen.

# Table of Contents

## Introduction

Opening Address .....	1
<i>Miyoko Akashi</i>	
The University of Copenhagen: Presentation and Some Models for Cooperation .....	3
<i>Jørgen Olsen</i>	
Introduction to the Graduate School of Information Science and Technology, University of Tokyo .....	7
<i>Masato Takeichi</i>	
DIKU: Research and Teaching .....	11
<i>Stig Skelboe</i>	
Research at the TOPPS Group .....	14
<i>Neil D. Jones</i>	
Introduction to Hagiya Laboratory .....	18
<i>Masami Hagiya</i>	
Research at Takeichi Laboratory .....	24
<i>Zhenjiang Hu</i>	

## Software Construction

Parallel Tree Reduction and its Implementation in C++ .....	30
<i>Kiminori Matsuzaki</i>	
Coccinelle: A Language-based Approach to Managing the Evolution of Linux Device Drivers .....	36
<i>Julia Lawall, Gilles Muller</i>	

## Program Transformation

BiXJ: A Java Library for Bidirectional XML Transformation .....	38
<i>Dongxi Liu, Zhenjiang Hu, Masato Takeichi, Kazuhiko Kakehi, Hao Wang</i>	
Incrementalization of Axapta Report Programs .....	50
<i>Michael Nissen</i>	
Report of an Implementation of a Semi-inverter .....	54
<i>Torben Ægidius Mogensen</i>	

## Language Design and Processing

Streamlining Functional XML Processing .....	63
<i>Keisuke Nakano</i>	
Parametric Polymorphism for XML .....	79
<i>Haruo Hosoya, Alain Frisch, Giuseppe Castagna</i>	
Compositional Specification of Commercial Contracts.....	80
<i>Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, Christian Stefansen</i>	

## Termination

The Size-Change Termination Principle on Non-wellfounded Data Types .	95
<i>James Avery</i>	
Termination Analysis by the Size-Change Principle of Programs in the Untyped Lambda-Calculus with Arbitrary Input from a Well-defined Input-Set .....	109
<i>Nina Bohr</i>	

## Semantics and Rewriting

IO Swapping Leads you There and Back Again .....	111
<i>Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, Masato Takeichi</i>	
On the Relations between Monadic Semantics .....	118
<i>Andrzej Filinski</i>	
Infinitary Combinatory Reduction Systems .....	124
<i>Jeroen Ketema, Jakob Grue Simonsen</i>	

## Program Analysis

Multiset Discrimination for Acyclic Data.....	139
<i>Fritz Henglein</i>	
Natural Numbers Type in Call-by-Value Based on CPS Semantics.....	150
<i>Yoshihiko Kakutani</i>	
Verification of Liveness Properties.....	155
<i>Carl Christian Frederiksen</i>	

<b>Author Index</b> .....	165
---------------------------	-----



Dragør, Denmark (Sept.2005)





# Opening Address

Miyoko Akashi

Embassy of Japan in Denmark

Ladies and Gentlemen,

May I offer my congratulations to you on this special event which marks the first official joint step of academic cooperation between the two universities, the University of Copenhagen and the University of Tokyo? The cooperation is based on the academic exchange agreement signed in November last year on the occasion of the state visit of H.M. Queen Margrethe of Denmark to Japan.

Bilateral relations between our two countries have been very good throughout our history. This is very much due to the close ties between the Royal Family of Denmark and the Imperial Family of Japan. One could say that the relationship is almost too good so that we are in danger of taking the good situation for granted and becoming complacent about facing the challenges of the future.

However, in the last few years, our bilateral relations have become more proactive than ever before. In 2002, during the Danish presidency of the EU, Prime Minister Anders Fogh Rasmussen, together with his EU colleagues, had an important meeting with the Prime Minister of Japan, Mr. Jun-ichiro Koizumi and they signed a document which has given an enormous impulse to all levels of all-round exchanges between EU member states and Japan. And this year, in Denmark too, many cultural, political, economic and social grass-root events have been organized with great success.

In 2004, the state visit made by H.M. Queen Margrethe made a breakthrough into more solid relations in various fields and genres. The official Danish delegation which accompanied the Queen met business partners in Japan and also technological and academic cooperation was taken care of extensively. You must be proud to know that this workshop is one of the important fruits of the visit. The youth exchange program known as the Working Holiday Program has been agreed between our foreign ministers. The multilateral trade partnership in Asia seems to be getting started.

Moreover, in the fields of diplomacy and international politics, global problems and issues, more and more close contacts can be seen between our two governments, such as cooperation on UN reform, counter-terrorism, the eradication of poverty and the preservation of the environment etc. Actually, in many respects, our two countries share the same values and ideas. More can be done in the future.

Continuance is power. We should keep up this momentum and continue to challenge common problems together.

I sincerely hope that this workshop will be very successful and will be a basis for the continued development of the cooperation between our faculties and universities in the years to come.



## The University of Copenhagen

### Presentation and some models for cooperation

Joint Workshop on Foundations of Software  
23 September 2005

Jørgen Olsen  
Pro vice-chancellor at the University of Copenhagen



## University of Copenhagen

- The oldest and biggest research and education establishment in Denmark
- Founded as a Catholic university in 1479 and reformed as a Protestant university in 1536
- More than 32.000 students
- 6.700 employees
- Budget approx. 650 billion USD



## Organisational structure

### University Board:

11 members, external majority, 2 students, 2 academics and 1 administrative representative:  
Strategy, budget, appointment of rector, performance contract with the ministry

### Rector:

Day-to-day management of the University and appointment of deans

### Deans:

Day-to-day management of Faculties and appointment of heads of departments



## The faculties

### Traditional structure with 6 faculties:

- The Faculty of Theology (unitary faculty)
- The Faculty of Law (unitary faculty)
- The Faculty of Social Sciences (6 departments)
- The Faculty of Health Sciences (16 departments)
- The Faculty of Humanities (8 departments)
- The Faculty of Science (11 departments)



## The Mission of the University

### According to the University Act 2003:

- The University shall conduct research and offer reasearch-based education at the highest international level
- The University has freedom of research and shall safeguard this freedom and ensure the ethics of science
- The University shall collaborate with society and contribute to the development of international collaboration
- The University shall exchange knowledge and competences with society and encourage its employees to take part in the public debate
- The University shall contribute to ensuring that the most recent knowledge within relevant disciplines is made available to non-research oriented higher education



## Internationalisation

### International Students

More than 1200 international students every year

Student exchanges via:

*Institutional Agreements*  
*Socrates-Erasmus*  
*Nordplus*  
*State Bilateral Agreements*



## Crossroads Copenhagen

Crossroads Copenhagen is a professional network involving private and public enterprises with a commitment in *culture, media and/or communication technology*.



## Partners in Copenhagen Crossroads

- The IT University of Copenhagen
- Copenhagen Business School
- The Faculty of Humanities at the University of Copenhagen
- NOKIA
- CSC
- Hewlett-Packard
- TDC - Teleoperator
- Skanska
- DR - The Danish Broadcasting Corporation
- The Royal Library
- The Danish Consumer Information
- The Danish Business Daily Børsen
- The City of Copenhagen
- The Ministry of Research, Technology and Innovation



## Regional & local models for cooperation



ØRESUND UNIVERSITY

14 collaborating universities



## Facts and figures regarding the Øresund University

- 14 member universities
- 140.000 students
- 6.500 PhD students
- 10.000 researchers
- 4000 foreign students
- Networks with 800 universities world wide
- 5<sup>th</sup> in Europe on scientific output
- 8 Nobel Prizes



## The Alliance of International Research Universities (Preliminary name)

### Partner universities

- Australia National University
- National University of Singapore
- Tokyo University
- Peking University
- ETH Zürich
- University of Copenhagen
- Cambridge University
- Oxford University
- University of California Berkeley
- Yale University

### The Memorandum of Understanding includes:

Co-operation on: Research projects and protection of Intellectual Property, Joint degree programmes, Summer Schools, financing and sponsorship of activities, exchange of students and faculty, benchmarking, secretariat, web site and chat-room.

### Principles that underpin the Alliance

- (i) The Alliance will be strategic, drawing together a select group of research-intensive universities that share similar values, a global vision and a commitment to educating future world leaders. Central to these values is the importance of academic diversity and international collaboration



## Denmark - Home of Fairytales...





Introduction to  
Graduate School of  
Information Science and Technology  
University of Tokyo

www.i.u-tokyo.ac.jp

September 23, 2005

**Masato Takeichi**  
Dean IST, University of Tokyo



Outline of the University of Tokyo I



- Established in 1877 as the first National University
- Incorporated in 2004 as a National University Corporation
- 2,800 Professors, Assoc. Professors
- 29,000 Students
  - 3,250 Undergraduates graduate every year
  - 2,700 Masters and 1,000 Ph.D's every year
- 10 Undergraduate and 14 Graduate Schools



Information Science and Technology, University of Tokyo (September, 2005)

2

Outline of the University of Tokyo II



- Annual Expenditures
  - 180,315M Yen = US\$ 1,639M
    - Personnel 42.8% / Materials 27.9%
    - Educational&Research 42.7% / Hospital 20.6% / Administrative 7.4%
- Annual Revenues
  - Own revenue 48% (86,272M Yen = US\$ 784M)
  - From the 3rd Parties 16.4% (29,512M Yen = US\$ 268M)
- Scientific Research Grants
  - 22,918M Yen = US\$ 208M



Information Science and Technology, University of Tokyo (September, 2005)

3

## Outline of Graduate School of Information Science and Technology



- Established in 2001
  - Reorganizing 5 departments of School of Science and School of Engineering
- 60 Professors, Assoc. Professors [2.1%]
- 600 Students
  - 190 Masters and 60 Ph.D's every year [7.0%]
- Annual Revenues
  - Materials: 360M Yen = US\$ 3.3M
  - From the 3rd parties: 1,357M Yen = US\$ 12.3M [4.6%]
  - Scientific Research Grants: 709M Yen = US\$ 6.4M [3.1%]

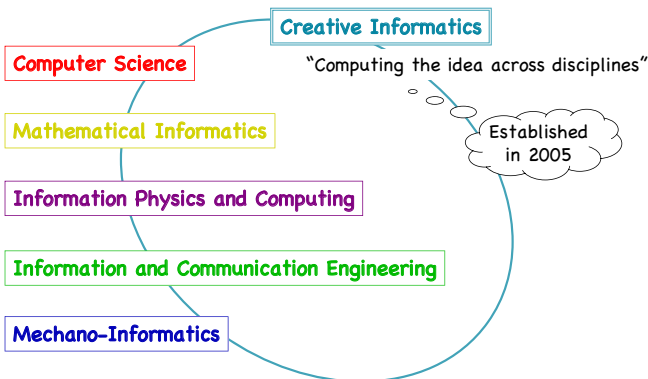
IST/UT ratio



Information Science and Technology, University of Tokyo (September, 2005)

4

## Graduate School of Information Science and Technology - Departments



Information Science and Technology, University of Tokyo (September, 2005)

5

## Graduate School of Information Science and Technology - Keywords I



### Computer Science

Logic, Computing Algorithm, Computer Languages, Operating Systems, Computer Architectures, Parallel and Distributed Computing, Security, Graphics, Numerical Computation, Natural Language Processing, Knowledge Discovery, User Interfaces, Genome Informatics, Computational Science

### Mathematical Informatics

Mathematical Informatics, Mathematical Modeling, Operational Research, Mathematical Programming, Probabilistic/Statistical Analysis, Numerical Analysis, Computational Mathematics, Computational Geometry, Information Theory, Mathematical Bases of Information Coding, Mathematical Bases of Complex Systems, Mathematical Bases of Bio-information, Mathematical Bases of Programming, Mathematical Finance

### Information Physics and Computing

Information Physics, Computing, Control Theory, Signal Processing, System Architecture, Physio- and Bio-cybernetics, Intelligent Sensors, Instrumentation and Sensory Systems, Integrated Intelligent Systems, Image and Speech Recognition and Synthesis, Adaptive Recognition and Control Systems, Virtual Reality, Tele-Robotics



Information Science and Technology, University of Tokyo (September, 2005)

6



## Graduate School of Information Science and Technology - Keywords II



### Information and Communication Engineering

Information & Electronics, Computer Architecture, Basic Software, Information System Design, Intelligent Information Processing, Database, Information Networks, Networked Information Environments, Signal Processing in Communication and Media, Communication Theory/Systems, Information Security, Media Technologies, Human Interface, Information Media Environments

### Mechano-Informatics

Mechatronics, Robotics, Micro-Nano Systems, Virtual Reality, Human Interfaces, Artificial Intelligence, Cognitive Informatics, Real World Informatics, Brain Informatics Machines, Bioinformatics Systems, Welfare Systems, Computer Aided Surgery

### Creative Informatics

Strategic System Creation, Strategic Network Software, Ubiquitous Network, Software Engineering, Software Verification, Real-time Distributed System, Human Media, Agent Technology, Intelligent Informatics, Natural Language Processing, Cognitive Action System, Real-world Robotics



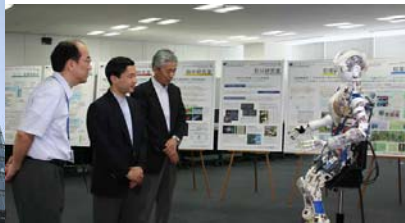
Information Science and Technology, University of Tokyo (September, 2005)

7

## IST "Satellite" Laboratory in Akihabara



Promoting Collaborative Research  
with Industries, Government, ...



Crown Prince Naruhito visits IST  
Akihabara Laboratory (August 18, 2005)



Information Science and Technology, University of Tokyo (September, 2005)

8

## Reported ..., and ...



- In "World's top 100 in Engineering and IT" (Times Higher Education Supplement, THES 12/2004) ranked University of Tokyo #8 in the world.
  - UC Berkeley, MIT, Stanford, Indian Institute of Technology, Imperial College London, California Institute of Technology, University of Tokyo, ...
  - Low score in internationalization; staffs, students

Improvement required for  
President's "The World's Tokyo University"



Information Science and Technology, University of Tokyo (September, 2005)

9

Academic Exchange Agreement Ceremony  
between Faculty of Science, University of Copenhagen and  
Graduate School of IST, University of Tokyo



November 15, 2004



Information Science and Technology, University of Tokyo (September, 2005)



## Research and Teaching

Stig Skelboe, Dept. Chairman



Department of Computer Science  
University of Copenhagen

## What characterizes DIKU?



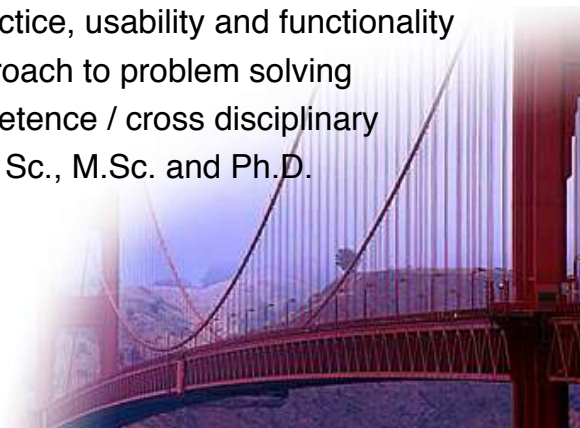
Founded 1970

Bridging practice, usability and functionality

Holistic approach to problem solving

Broad competence / cross disciplinary

Teaching B. Sc., M.Sc. and Ph.D.



## Which problems do we solve?



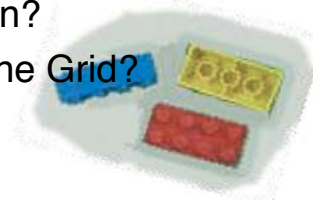
How do we pack the knapsack?

How do we simulate interacting objects?

How do we prevent your computer from  
breaking down?

How do we ease document reading on a  
pda or computer screen?

How do we construct the Grid?



## We have five research groups:



### Programming

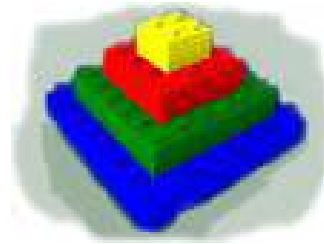
Distributed Systems & TOPPS

### Mathematics

Image Group & Algorithm and Optimization

### Systems

HCI and Systems Design



## Students



1000+ students enrolled

100 B.Sc. per year

40 M.Sc. per year

5 Ph.D. candidates per year

## Faculty of Natural Science



Founded 1850

9 departments

10 centers

3 museums

6000 students

# University of Copenhagen



Founded in 1479

6 faculties

Many locations in Copenhagen, Denmark  
and one in Greenland

32000 students

# Research at the TOPPS group (DIKU, University of Copenhagen)

Neil D. Jones

September, 2005

## What is TOPPS?

TOPPS is an acronym for “Teori Og Praksis i ProgrammeringsSprog”, which translates to “Theory and Practice of Programming Languages”. It is used as the name of a very active and international research group at DIKU, the Department of Computer Science at the University of Copenhagen, Denmark. The group has been doing innovative research in the field of *Programming Languages* since the early 1980s, as evidenced by external recognition, grants and involvement with leading international conferences. Students and researchers from TOPPS have been employed in Australia, Denmark, France, Germany, Japan, the UK and the US.

TOPPS performs both basic and applied research to advance programming language technologies in several directions, including:

1. *Domain-specific programming languages*, “tailor-made” languages for specific applications.
2. *Automatic program analysis*, e.g., to certify error-freedom, more generally *safety and liveness properties* of critical software components.
3. *Type-based memory management* to reduce memory consumption and prevent memory faults.
4. *Theoretical aspects of programming languages*: logic and computation, the limits of computer-solvable problems, computability and complexity from a programming perspective.
5. *Software production, generation and manipulation*: compilers and interpreters, bootstrapping, partial evaluation, program transformation, program inversion.

## Current members

Andrzej Filinski	Associate professor
Robert Glück	Associate professor
Klaus Grue	Associate professor
Fritz Henglein	Professor
Neil Jones	Professor
Julia Lawall	Associate professor
Torben Mogensen	Associate professor
Jakob Grue Simonsen	Research adjunct

## A brief profile of TOPPS

TOPPS programming language research is recognised worldwide, as indicated by the following.

**Research production.** The group’s members have been quite productive since the 1980s, with substantial numbers of scientific articles and several books. Most of them are described in the bibliography at <http://www.diku.dk/topps/Bibliography.html>, and the group’s web pages: <http://www.diku.dk/topps/> give some topical overviews.

**Citations.** *CiteSeer* is the main citation index for research in Computer Science. In particular, CiteSeer periodically compiles a list of the 10,000 most-cited researchers in Computer Science. A recent (2004) version of the list included 19 researchers based in Denmark. Among these, 7 were current or former TOPPS members, including the *two most-frequently cited* Danish researchers.

**EU/Esprit/Danish research grants.** The TOPPS group has been supported by a long series of research grants. This includes European Union support: Semantique I and II, Atlantique, Daedalus, ABILE, APPSEM; and Danish support: continuous support by SNF (Danish Natural Science Research Council) since the 1980s, support by CIT (Center for IT Research), and Ph.D. support by the University of Copenhagen, as well as MBS (Microsoft Business Solutions) and ATV (Academy for the Technical Sciences).

### Recent international meetings hosted by TOPPS in Copenhagen

- Summer school on *Program Analysis and Transformation*, June 2005.
- *First Joint Workshop on the Foundations of Software* in Denmark together with the University of Tokyo, September 2005.
- *17th Nordic Workshop on Programming Theory*, October 2005.
- The large conference cluster FLoC'02 (*Federated Logic Conference 2002*) consisted of 7 main conferences and more than 30 workshops, with 950 participants. FLoC'02 was hosted by the TOPPS group, whose members did most of the local organisation, and included the FLoC'02 Conference Chair.<sup>1</sup>

Other international conference series initiated by TOPPS members include PEPM (Partial Evaluation and Semantics-based Program Manipulation), continuous since 1991, and SPACE (Semantics, Program Analysis, and Computing Environments for memory management), held in 2001, 2004 and 2006.

The worldwide leading professional association for research in Computer Science is the US-based ACM, the *Association for Computing Machinery*. One of ACMs first conference series, and its flagship conference in the TOPPS research area, is POPL: *Principles of Programming Languages*.

TOPPS members have been involved in POPL since the 1980s, have published many scientific articles in its yearly proceedings, and have given several invited lectures. Further, TOPPS members were POPL Program Committee Chair (Paris 1997), General Chair (Paris 1997 and Venice 2004) and Chair of the Steering Committee (2004-2005).

**American research institutions.** Former TOPPS members are now employed at top-rate American Computer Science research institutions. Examples include: two researchers at Microsoft Research (Redmond Washington); one researcher at IBM Yorktown Heights; and three members of the programming language research group at Kansas State University.

### Educational achievements.

At least 19 Ph.D. degrees for research in programming languages have been earned within the TOPPS group, and numerous M.Sc. degrees.

Ph.D. degrees were earned by: Lars Ole Andersen, Anders Bondorf, Niels Christensen, Hans Dybkjær, Carsten Gomard, Jesper Jørgensen, Henning Makholm, Christian Mossin, Torben Mogenssen, Henning Niss, Jakob Rehof, Kristoffer Holm Rose, Jens Peter Secher, Peter Sestoft, Jakob Grue Simonsen, Sebastian Skalberg, Harald Søndergaard, Morten Heine Sørensen, Morten Welinder. Employers of these graduates included: Danish research institutions (5), Danish industry (8), foreign research institutions (5), and foreign industry (1).

---

<sup>1</sup>FLoC'06, the successor to FLoC'02, will be held in Seattle and is to be organised by Microsoft. Its Conference Chair is Jakob Rehof, who earned his Ph.D. degree in 1998 in the TOPPS group.

## Some TOPPS research breakthroughs.

**Partial evaluation.** Members of the TOPPS group [Jones, Sestoft, Søndergaard] created the world's first *self-applicable partial evaluator*<sup>2</sup>: a program whose construction settled an open problem that had stood unsolved for 13 years. This success showed for the first time (in practice as well as in theory) that partial evaluation could be used automatically to generate compilers from programming language interpreters. TOPPS members currently active in this area: Andrzej Filinski, Robert Glück, Julia Lawall, Torben Mogensen.

**Other well-recognised achievements** by TOPPS researchers:

- *Automatic program analysis*, e.g., termination analysis [Neil Jones, James Avery, Nina Bohr], and two much-cited pointer analyses to recognise possible variable aliasing in C language<sup>3</sup> [Lars Ole Andersen and Bjarne Steensgaard].
- *Type-based program analysis* was used to alleviate bad effects of the “year 2000” problem, and has been used for many applications since then [Fritz Henglein, Mads Tofte, et al.].
- *Region inference* allows implicit automatic memory management with order-of-magnitude reductions in storage requirements [Mads Tofte, Fritz Henglein, Henning Niss, Henning Makhholm, et al.].
- *Map theory* is an integration of computer science algorithms and reasoning (the  $\lambda$ -calculus) with classical mathematics [Klaus Grue].
- The programming language *Standard ML*: Earlier TOPPS members defined the first precise semantics for the language, co-authored two frequently-cited books, and developed a very widely-used implementation for teaching [Mads Tofte, Peter Sestoft, et al.].

**Current projects** include

- *Plan-X*: “If the network is the computer, how do we program it?” This aim is a software platform where program and data, running on one machine, can be distributed on a network and moved about, even while client programs execute and software and data migrate. The purpose is to allow application-focused programming, leaving the computer science machinery to the software platform. [Fritz Henglein, Thomas Ambus, Mads Pultz, et al]
- *Logiweb*: a system to construct a *world wide web of theorems and proofs*. It specifies protocols, data structures, and abstract machines to realize such a web of theorems and proofs. [Klaus Grue]
- *NEXT*: “Enterprise Resource Planning Systems” This project (joint with Microsoft Business Solutions IT University of Copenhagen), aims to develop software technology and methods to construct a next generation of systems to steer businesses. [Fritz Henglein, Christian Stefansen, Daniel Brixen, Ebbe Elsborg, et al].
- *Coccinelle*: A language-based approach to managing the evolution of Unix device drivers. Software systems frequently evolve to improve performance, fix bugs, and address new requirements. When evolutions affect heavily used interfaces, however, a multitude of *collateral evolutions* may be needed in dependent modules. An example is the frequently evolving Linux operating system. This project will design a transformation language dedicated to specifying the modifications needed for collateral evolution, and a transformation tool to apply them to device driver code. [Julia Lawall].

---

<sup>2</sup>The purpose of partial evaluation is to make programs run faster. A partial evaluator is given a program together with some, but not all, of its input data. From these it builds a new *specialised program* that, given any remaining input data, computes the same answer the original program would yield if given all its inputs. In practice partial evaluation sometimes yields order-of-magnitude program speedups.

<sup>3</sup>These analyses are at the base of some tools currently used by Microsoft.



## Theory in relation to applications: the TOPPS approach

The research areas above all focus on topics that lie near the border between theory and practice. Many of the theoretical results concern practical aspects of computing, and many of the practical parts of the work are carefully chosen to cast experimental light on questions of principle that arose in theoretical investigations.

**Context: Programming languages** are the raw material of the complex conceptual structures that make up all computer software. Formalized computer languages and their semantics play a central rôle in all phases of software development. This begins with a description of the desired abstract software entity, often expressed using a formal *specification language*. A next phase: the refinement of the specification into a prototype executable version expressed in an *implementation language*. Later and equally important steps: verifying that the constructed software will behave in *reliable and error-free* ways in safety-critical applications; and that software and hardware combinations can be guaranteed to run within *preassigned space and speed constraints*.

Since the 1950s, advances in programming languages have greatly aided effective computer usage. The phenomenal success of Java is an example of how advances in programming languages (clean semantics and well-planned implementation) can reach widespread acceptance. Thanks to Java, automatic memory management, type safety and several other features, well-known in research-related communities but not available in languages like C++ and Fortran, have finally entered mainstream computing.

**The rôle of research in programming languages**, as we see it, is to develop and test ideas that may gain widespread use ten years from now. The programming language research span is wide, encompassing: significant *experimental work* on language design, program transformation, etc.; *implementation technology*, which is necessary to establish the practical viability of new ideas; and purely *theoretical studies*, which are necessary to establish a foundation upon which all the other activities can be built.

**An approach to automation.** A central component of the group's world view is to use the computer itself for program optimisation, analysis, debugging, transformation and adaptation to new contexts.

The current practice of programming is (still) essentially a handcraft, analogous to one-at-a-time construction of artefacts by a blacksmith. In order better to automate programming, we need better computer tools to aid program creation, maintenance, updating, and other forms of program manipulation. These can be of higher or lower level, for example tools to analyze existing programs to locate possible errors; interpreters or compilers; and tools to aid in creating programs, given precise specifications of what the programs are intended to accomplish.

Our opinion, based on many years' experience: automatic program manipulation works much better when solidly based on the *semantics of programming languages*. This helps in systematically developing *correct and reliable* tools to manipulate existing programs and create new ones. Further, a good formal and informal understanding of semantics is essential to specify and implementing new languages (important: the number of existing poorly designed languages is legion).

**Problem-solving approaches.** The group is characterized by knowledge of and experience with a broad spectrum of programming languages: imperative, functional, object-oriented, logic and concurrent. Its emphases are and will continue to be on building semantically well-founded tools to aid humans in many ways when dealing with programs.

The group works with programming languages as an object of study and manipulation rather than just a tool for getting work done. In particular, we investigate methods of automatically analysing or transforming programs, often using the results of one to further the other. Some of the work done in the group centers on particular languages (e.g., ML and Scheme), while other work treats classes of languages that share common traits, like lazy functional languages or imperative languages.

# Introduction to Hagiya Lab. from Hagiya's home page

With background in formal logic, our laboratory is seeking for new computational models, developing new methods for analyzing and verifying such models, and implementing new tools for the analysis and verification.

While our main target is software and programming languages, we are also dealing with molecular and biological systems (DNA and molecular computing).

## Hagiya Lab.

- Students are doing what they want to do
  - Nishizawa (D3): category theory for abstract interpretation
  - Frederiksen (D2): termination and liveness analysis
  - Abe (D2): concurrency
  - Tanabe (D1): shape analysis
  - Kawamura (M2): computable analysis
  - Kakutani (Research Associate): categorical semantics
- Collaborations
  - Research Center for Verification and Semantics, AIST
  - NTT Communication Science Laboratories
  - Chiba University
  - IBM TRL
  - Other collaborations on DNA and molecular computing

## Satisfiability Checking of Two-way Modal $\mu$ -calculus and Its Applications

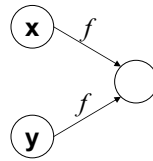
- Branching-time temporal logic can naturally be used to express properties of trees and graphs, where two-way logic is more useful because it can handle both forward and backward edges
- We have implemented a tableaux-based satisfiability checker for **alternation-free two-way modal  $\mu$ -calculus** using BDDs (TABLEAUX2005)
- and applied it to schema checking in XML (CIAA2003) and shape analysis (FLOPS2004)
- In particular, shape analysis can be considered as tableaux construction with suitable extensions to temporal logic (DSN2005): two-way, nominal, and global modality

## Two-way (an extension to temporal logic)

Each modality has its backward counterpart

Ex.: The variables  $x$  and  $y$   
point to the same node

$$x \wedge \langle f \rangle \langle \bar{f} \rangle y$$



Alternation-free two-way modal  $\mu$ -calculus

Prop: set of propositional variables

Mod: set of modalities

Backward modality:  $\bar{\phantom{a}}$ : Mod  $\rightarrow$  Mod,  $\overline{\overline{a}} = a$

Two-way CTL — a subsystem

## Satisfiability Checking...

- two-way (full)
  - Vardi 1998: full  $\mu$ -calculus (no efficient implementation?)
  - Sattler and Vardi 2001: hybrid full  $\mu$ -calculus (also handling nominals)
- using BDDs for tableaux
  - Pan, Sattler and Vardi 2002: for  $K$
  - Pan and Vardi 2003: also for  $K$
  - MONA: WS2S

## Our Tableaux Method

- Definition of  $\varphi_I$ -types (nodes of the tableau)
  - In order to avoid loops following forward and backward modalities, we introduce an irreflexive and transitive relation in each  $\varphi_I$ -type
- Consistency conditions of  $\varphi_I$ -types
  - Various consistency conditions are defined, including loop-freedom using the irreflexive and transitive relation
- Technical but crucial improvements
- Implementation by BDDs

## Application to XML

Tozawa and Hagiya proposed XML schema containment checking using BDDs (CIAA2003)

We have recently extended the method to handle schema checking of XML transducers

The satisfiability checker for alternation-free two-way  $\mu$ -calculus is used by restricting models to finite binary trees

Tozawa has reported that using the satisfiability checker is more efficient than using MONA

## Abstraction of Graphs (Shape Analysis)

A directed graph is a Kripke structure

Each node in a graph is characterized by a finite set  $F$  of modal formulas

- An abstract node  $C$  is a subset of  $F$ :  $C \subseteq F$

- A concrete node  $m$  is abstracted to

$$\{\varphi \in F \mid m \models \varphi\} \subseteq F$$

- An abstract graph  $G$  is a set of abstract nodes, i.e.,  $G \subseteq 2^F$

## Simple Analysis of Lists

- Mod =  $\{f, b\}, \bar{f} = b$
- Prop =  $\{\mathbf{x}, \mathbf{y}\}$ 
  - $\mathbf{x}$ : pointed to by the variable  $x$
  - $\mathbf{y}$ : pointed to by the variable  $y$
- $F = \{\mathbf{x}, \mathbf{y}, \langle f \rangle \text{true}, E_{\{f\}} \mathbf{F}\mathbf{x}, E_{\{f\}} \mathbf{F}\mathbf{y}, E_{\{b\}} \mathbf{F}\mathbf{x}, E_{\{b\}} \mathbf{F}\mathbf{y}\}$
- $C_1 = \{\mathbf{x}, \langle f \rangle \text{true}, E_{\{f\}} \mathbf{F}\mathbf{x}, E_{\{f\}} \mathbf{F}\mathbf{y}, E_{\{b\}} \mathbf{F}\mathbf{x}\}$
- $C_2 = \{\langle f \rangle \text{true}, E_{\{f\}} \mathbf{F}\mathbf{y}, E_{\{b\}} \mathbf{F}\mathbf{x}\}$
- $C_3 = \{\mathbf{y}, E_{\{f\}} \mathbf{F}\mathbf{y}, E_{\{b\}} \mathbf{F}\mathbf{x}, E_{\{b\}} \mathbf{F}\mathbf{y}\}$

## Global Modality (another extension to temporal logic)

A, E : global modalities

- $m \models \mathbf{A}\varphi \Leftrightarrow n \models \varphi$  holds for all  $n \in M$
- $m \models \mathbf{E}\varphi \Leftrightarrow n \models \varphi$  holds for some  $n \in M$

For the abstract graph  $\mathbf{G}$ , there exists a natural surjection from the tableau of the formula

$$\mathbf{A}(\bigvee\{\varphi_C \mid C \in \mathbf{G}\})$$

to the abstract graph, i.e., the tableau can be regarded as a refinement of the abstract graph

$$\varphi_C \equiv \bigwedge\{\varphi \mid \varphi \in C\} \wedge \bigwedge\{\neg\varphi \mid \varphi \in F-C\}$$

Shape analysis is tableaux

## Nominal (yet another extension to temporal logic)

In Sagiv et al.'s shape analysis, abstract nodes are classified into summary and non-summary nodes

A nominal is a kind of propositional constant, but it holds at exactly one node in a Kripke structure

(Modal logic with nominals is called hybrid)

Nominals correspond to non-summary nodes in the shape analysis

- $\mathbf{x}$  and  $\mathbf{y}$  in the simple analysis of lists should be nominals because they determine unique nodes

We have extended our BDD-based satisfiability checker to cope with nominals, though it is not complete

- A formula is unsatisfiable if tableaux construction fails

## TLAT (Temporal Logic Abstraction Tool)

Abstract graphs are not constructed explicitly but implicitly by the “Shape Analysis is Tableau” principle

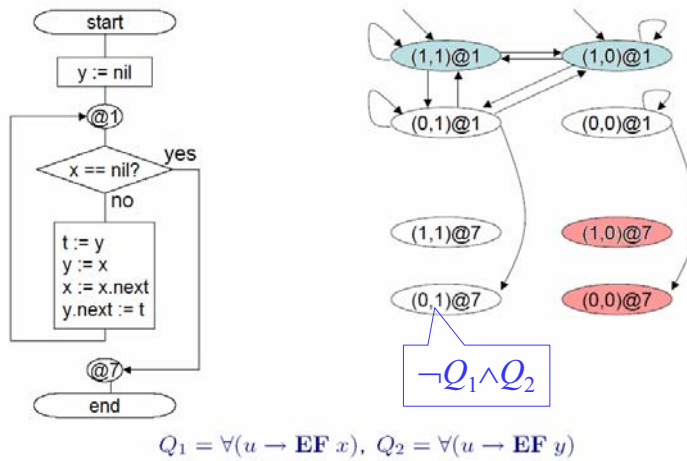
$Q_1, \dots, Q_n$  : formulas of the form  $\mathbf{A}\varphi$  (given by the user)

Abstract state:  $\varphi@i$

- $\varphi$ : conjunction of  $Q_k$  or  $\neg Q_k$
- $i$ : program counter

The weakest precondition of a program in PML (Pointer Manipulation Language) is computed

A transition between abstract states  $\varphi@i$  and  $\psi@j$  is allowed if  $\varphi \wedge \mathbf{wp}(\sigma_{ij}, \psi)$  is satisfiable, where  $\sigma_{ij}$  is the program statement from  $i$  to  $j$



## Current and Future Work

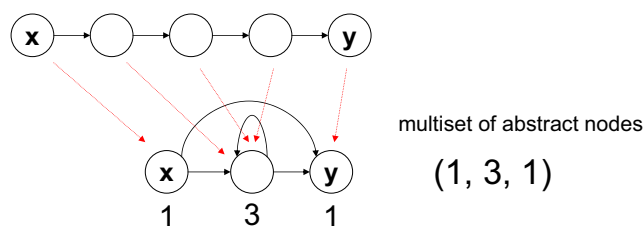
- Complete and efficient satisfiability checker with nominals
- Comparison with other (many) frameworks:
  - TVLA, MONA, Description logic, ...
- Spatio-temporal logic
- Cardinality analysis  $\rightarrow$  Termination analysis (cf. Frederiksen)
- Adding clocks or continuous parameters
  - Cardinality analysis  $\rightarrow$  Timed multiset rewriting (FTRTFT2002)

## Cardinality Analysis

Analyze changes in the number of nodes corresponding to each abstract node

Abstraction from graphs to multisets

- Less abstract than abstract graphs, but still infinite



## Cardinality Analysis by **wp**

- $C_i$  : abstract node
- $\varphi_i$  : the formula  $\varphi_{C_i}$  corresponding to  $C_i$
- $k_i$  : the cardinality of  $C_i$  before the transition  $\sigma$
- $k_i'$  : the cardinality of  $C_i$  after  $\sigma$

$$\models \varphi_{i_1} \vee \dots \vee \varphi_{i_m} \supset \mathbf{wp}(\sigma, \varphi_{j_1} \vee \dots \vee \varphi_{j_n}) \Rightarrow$$
$$k_{i_1} + \dots + k_{i_m} \leq k_{j_1}' + \dots + k_{j_n}'$$

$$\sum_{C_i \text{ contains } \mathbf{x}} k_i = 1, \quad \sum_{C_i \text{ contains } \mathbf{x}} k_i' = 1$$

## Research at Takeichi Laboratory

Zhenjiang Hu  
The University of Tokyo  
September 24, 2005

## Information Processing Laboratory



Department of Mathematical Informatics /  
Department of Creative Informatics  
Graduate School of Information Science and Technology  
The University of Tokyo

<http://www.ipl.t.u-tokyo.ac.jp>

## Members in IPL Lab.

- **Staffs**
  - ▶ **Masato Takeichi**, Professor, Head
  - ▶ **Zhenjiang Hu**, Associate Professor
  - ▶ **Kiminori Matsuzaki**, Research Associate
- **Visiting Researchers**
  - ▶ Xi Chen
- **Students**
  - ▶ Ph.D students (1)
  - ▶ Master students (9)
  - ▶ Research students (1) (*will be*)



## Programmable Structured Documents Laboratory



Department of Mathematical Informatics  
Graduate School of Information Science and Technology  
The University of Tokyo

<http://www.psdlab.org/>

## Staffs in PSD Lab.

- Masato Takeichi, Professor, Head
- Zhenjiang Hu, Associate Professor
- Kazuhiko Kakehi, Assistant Professor (Project)
- Yasushi Hayashi, Researcher
- Dongxi Liu, Researcher
- Shin-Cheng Mu, Researcher
- Keisuke Nakano, Researcher

## Current Research Topics

### The PSD Project



- A framework of *programmable structured documents*
- From *program* transformation to *document* transformation

### The SkeTo Project



- Systematic parallel programming with *skeletons*
- An environment supporting *sequential style* of parallel programming in C++

### The Yicho Project



- Development of calculation rules for *fold-free* program transformation
- A system for implementing calculation rules

## The PSD Project

### Programmable Structured Documents

Structure documents + Code

```
<addrbook>
  <names ref="contents", code="extractName" />
  <addresses>
    <number ref="contents", code="count" />
    <contents label="contents" >
      <person><name>Masato Takeichi</name> ...
      <person><name>Zhenjiang Hu</name> ... <
      <person><name>Shin-Cheng Mu</name> ...
    </contents>
  </addresses>
</addrbook>
```

**IPL Address Book**

**Names** 3

Masato TAKEICHI: Prof.  
Zhenjiang HU: Assoc. Prof.  
Shin-Cheng MU: Post Doc

**Addresses** 3

Masato Takeichi: takeichi@  
Zhenjiang Hu: hu@mist  
Shin-Cheng Mu: scm@ipl

## Challenges

- How to *create* PSD by people even with little knowledge of programming?
- How to *evaluate* PSD efficiently?
- How to *update* PSD?
- How to *manipulate* PSD?
- How to *check* that PSD is well-formed?

**IPL Address Book**

**Names** 3

Masato TAKEICHI: Prof.  
Zhenjiang HU: Assoc. Prof.  
Shin-Cheng MU: Post Doc

**Addresses** 3

Masato Takeichi: takeichi@  
Zhenjiang Hu: hu@mist  
Shin-Cheng Mu: scm@ipl

## Our Idea

- How to create PSD by people even with little knowledge of programming?  
⇒ *XEditor*
- How to evaluate PSD efficiently?  
⇒ *Lazy evaluation*
- How to update PSD?  
⇒ *Bidirectional Scripting Languages: X, BiXJ*
- How to manipulate PSD?  
⇒ *Program calculation*
- How to check that PSD is well-formed?  
⇒ *Make a type-check system*

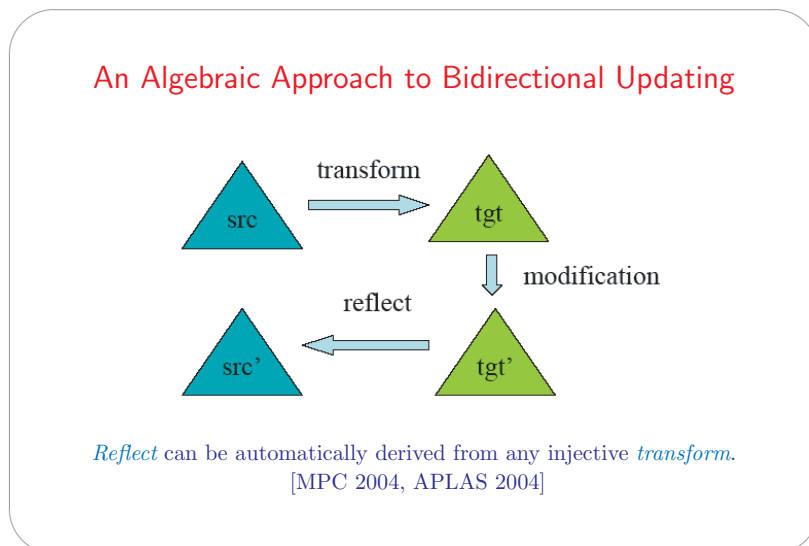
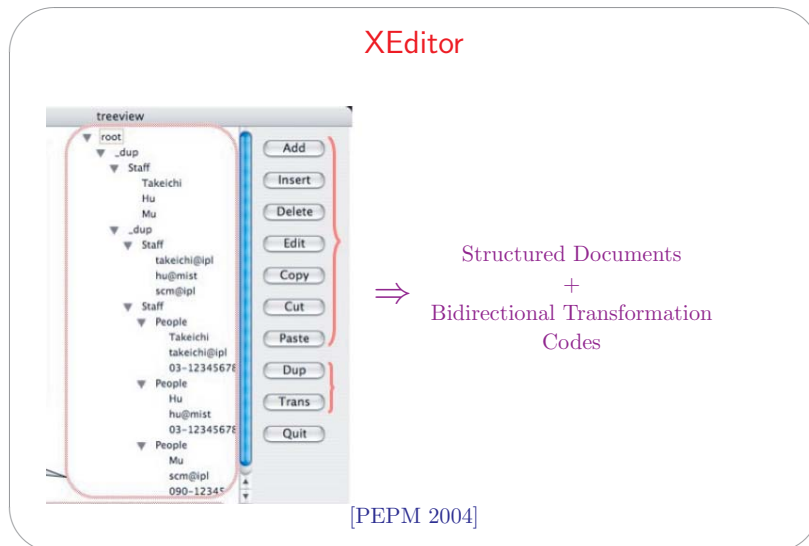
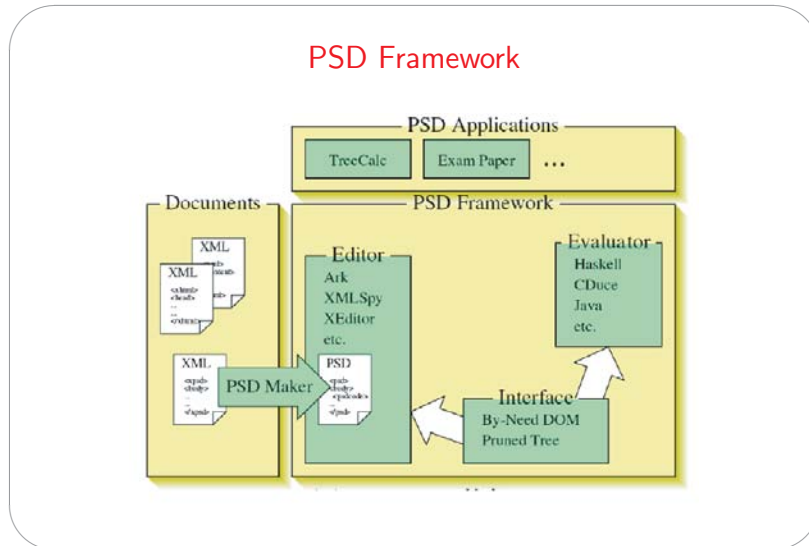
**IPL Address Book**

**Names** 3

Masato TAKEICHI: Prof.  
Zhenjiang HU: Assoc. Prof.  
Shin-Cheng MU: Post Doc

**Addresses** 3

Masato Takeichi: takeichi@  
Zhenjiang Hu: hu@mist  
Shin-Cheng Mu: scm@ipl



## A Bidirectional Language: X, BiXJ

- $X ::= B$  { primitives }
- |  $X ; X$  { sequencing }
- |  $X \otimes X$  { product }
- |  $\text{If } P \ X \ X$  { conditional branches }
- |  $\text{Map } X$  { apply to all children }
- |  $\text{Fold } X \ X$  { fold }
  
- $B ::= \text{GFun } (f, g)$  { Galoi function pairs }
- |  $\text{NFun } f$  { a simple function }
- |  $\text{Dup}$  { duplication }

[PEPM 2004, Liu's Talk]

## Applications: Tree Calculators

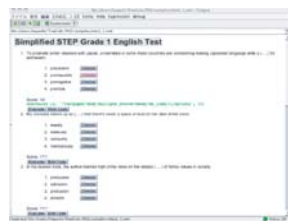


Figure 1: iExam

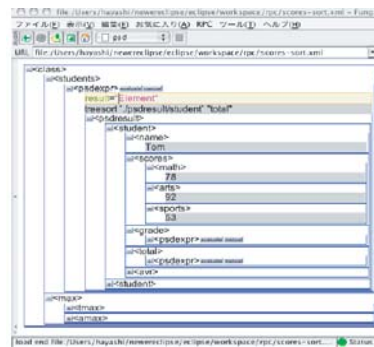


Figure 2: Student Management System  
[ACM DocEng 2005]

## Applications: iDocuments



Figure 3: iTextbook

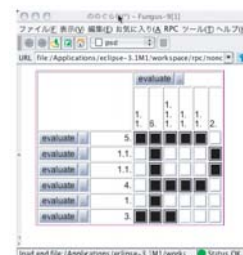


Figure 4: iPuzzle

## The Sketo Project

- How to program with parallel skeletons [TOPLAS:97, PEPM'99, ESOP'02]
  - ▶ What are basic skeletons?
  - ▶ How to combine basic skeletons to solve a problem?
- How to implement parallel skeletons efficiently in distributed-memory systems?
  - ▶ List skeletons [IJPP:04,EuroPar 2004]
  - ▶ Tree skeletons [EuroPar 2003, CMPP 2004]
  - ▶ Matrix skeletons [ongoing work]

<http://www.ipl.t.u-tokyo.ac.jp/sketo/>

## The Yicho Project

- Developing **calculation** rules for *fold-free transformations*:
  - ▶ Fusion [ICFP'96]
  - ▶ Tupling [ICFP'97]
  - ▶ Parallelization [POPL'98]
  - ▶ Accumulation [NGC:99]
  - ▶ Dynamic Programming (Maximum Marking Problems) [ICFP'00]
  - ▶ Iterative-free Program Analysis [ICFP'03]
  - ▶ IO-Swapping [ongoing work]
- Implementation:
  - ▶ *Deterministic* High-order Patterns [IPL:2004]
  - ▶ *Deterministic* High-order Matching Algorithms [TFP'05]

<http://www.ipl.t.u-tokyo.ac.jp/yicho/>

**Thank You!**

# Parallel Tree Reduction and its Implementation in C++ (Extended Abstract\*)

Kiminori Matsuzaki

Graduate School of Information Science and Technology  
University of Tokyo  
kmatsu@ipl.t.u-tokyo.ac.jp

## 1 Introduction

Trees are important datatypes that are often used in representing structured data such as XML. In recent years, the grow of computational power enables us to store huge data in forms of trees, which calls for methods of manipulating trees efficiently.

Though hardware environments for parallel computing are becoming widely available (e.g. PC clusters), parallel programming for trees is still known to be a hard task because of the ill-balanced and irregular structures of trees. For binary trees, there are parallel algorithms called tree contraction algorithms, which was first proposed by Miller and Reif [11] and studied by many researchers [1, 2, 6, 10, 12]. Though the tree contraction algorithm are efficient algorithms, it requires intuitions and/or experiences to design and implement the algorithms. For trees of arbitrary shapes there are only a few studies [5, 7] most of which are rather ad hoc.

In this paper, we formalize an important computational pattern called *reduction* or *homomorphism* on trees of arbitrary shapes. We then show this computational pattern is mapped to the reduction on binary trees, under certain conditions on the operators used in the reduction.

We have implemented several computational patterns on binary trees using C++ and MPI in the parallel skeleton library *SkeTo* [13]. We show furthermore how we can implement the reduction on general trees based on this ready-made implementation using the template mechanism and function objects in C++.

The rest of the paper is organized as follows. In Section 2, after briefly introducing the notational conventions, we review the tree contraction algorithms. In Section 3, we formalize the general tree reduction, and show how we can parallelize this computational pattern using the parallel tree contraction algorithms. In Section 4, we show an implementation of the general reduction on our parallel skeleton library. Finally in Section 5, we make conclusion remarks.

## 2 Preliminaries

### 2.1 Notations

In this paper, we use the notation of Haskell [3]. In the following, we briefly review important notations and define the datatypes of binary trees and general trees (called rose trees).

**Functions and Operators** Function application is denoted by a space and the argument may be written without brackets. Thus  $f a$  means  $f(a)$ . Functions are curried, and the function application associates to the left. Thus  $f a b$  means  $(f a) b$ . The function application

---

\* The detailed version of this paper was published as a technical report METR2005-30, Department of Mathematical Informatics, University of Tokyo [8].

binds stronger than any other operator, so  $f a \oplus b$  means  $(f a) \oplus b$ , but not  $f (a \oplus b)$ . Function composition is denoted by an infix operator  $\circ$ . By definition, we have  $(f \circ g) a = f (g a)$ . Function composition is associative and its unit is the identity function denoted by  $id$ .

Infix binary operators will be denoted by  $\oplus$ ,  $\otimes$ , etc, and their units are written as  $\iota_{\oplus}$ ,  $\iota_{\otimes}$ , respectively. These operators can be sectioned and be treated as functions, i.e.  $a \oplus b = (a \oplus) b = (\oplus b) a = (\oplus) a b$  holds.

In deriving parallel programs, algebraic rules on operators such as associativity or distributivity play important roles. We introduce an generalized rule of the distributivity in terms of a closure property of functions.

**Definition 1 (Extended Distributivity [7]).** Let  $\otimes$  be an associative operator. The operator  $\otimes$  is said to be *extended-distributive* over operator  $\oplus$ , if for any  $a, b, c, a', b'$ , and  $c'$ , the following equation holds.

$$(\lambda x. a \oplus (b \otimes x \otimes c)) \circ (\lambda x. a' \oplus (b' \otimes x \otimes c')) = \lambda x. A \oplus (B \otimes x \otimes C)$$

where  $A = p_1 (a, b, c, a', b', c')$ ,  $B = p_2 (a, b, c, a', b', c')$ , and  $C = p_3 (a, b, c, a', b', c')$ . The functions  $p_1$ ,  $p_2$ , and  $p_3$  are called characteristic functions.  $\square$

Many pairs of operators satisfy the extended distributivity. For example, let  $\oplus$  be an associative operator, then  $\oplus$  is also extended-distributive over  $\oplus$  itself. If operator  $\otimes$  distributes over operator  $\oplus$ , then of course the operator  $\otimes$  is extended-distributive over  $\oplus$ . There are many pairs of operators where the distributivity does not hold but the extended distributivity does. Thus, this property is useful for specifying the conditions for parallelization.

**Data Structures** Binary trees are trees whose internal nodes have exactly two children. In this paper we may assume the nodes in a binary tree have values of the same type.

**data**  $BTree \alpha = Leaf \alpha \mid Node \alpha (BTree \alpha) (BTree \alpha)$

Rose trees are trees whose internal nodes have an arbitrary number of children. In this paper we may assume the nodes in a rose tree have values of the same type.

**data**  $RTree \alpha = RNode \alpha [RTree \alpha]$

## 2.2 Parallel Tree Contraction Algorithms

The reduction on binary trees is a computational pattern that takes a binary tree and collapses the tree into a value by applying a function in a bottom-up manner.

$$\begin{aligned} \text{reduce}_b k (Leaf n) &= n \\ \text{reduce}_b k (Node n l r) &= k n (\text{reduce}_b k l) (\text{reduce}_b k r) \end{aligned}$$

The reduction is computed efficiently in parallel with the tree contraction algorithms. The parallel tree contraction algorithms were first proposed by Miller and Reif [11] and studied by many researchers [1, 2, 6, 10, 12]. The main idea of the tree contraction algorithms is to apply the local contractions independently on multiple nodes. One of the tree contraction algorithms applies *SHUNT* operation (Fig. 1), which removes a leaf and its parent node from a tree. The tree contraction algorithms collapses a binary tree of size  $n$  into a value in  $O(\log n)$  steps in regardless to the shape of trees.

To utilize the tree contraction algorithms, it may be required to modify the computation from the specification of reductions. For example, the computation of height of a tree can be

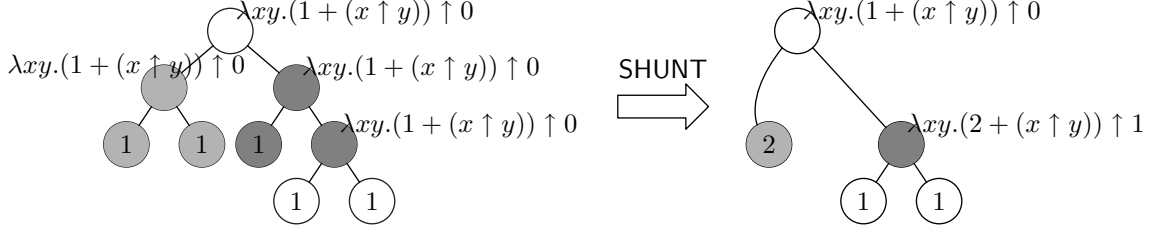


Fig. 1. The SHUNT operation and the computation for height of trees

defined by the reduction with function  $k$  defined as  $k\ n\ l'\ r' = 1 + (l' \uparrow r')$  where  $\uparrow$  return the larger of the two arguments. Unfortunately, it fails to compute the height of tree if we apply the function  $k$  above naively through the tree contraction. This is because the function  $k$  does not satisfy the closure property under function composition. To make the computation correct, it is required to modify the definition of  $k$  into a closed form of functions. In the case of computing the height, we can use a closed form  $\lambda\ l'\ r'. (a + (l' \uparrow r')) \uparrow b$ , where  $a$  and  $b$  are certain parameters. (Fig. 1).

As reviewed so far, tree contraction algorithms are efficient and important parallel algorithms for reductions on binary trees, but it is however necessary to find a closed form of functions to correctly compute with the algorithms.

### 3 General Tree Reduction

In this section, we first specify the reduction on rose trees, and then show how we can parallelize the reduction in terms of the reduction on binary trees.

#### 3.1 Specification

Since an internal node of rose trees may have an arbitrary number of children, we define the reduction on rose trees with two binary operators. The reduction on rose trees collapses the tree into a value in a bottom-up manner by applying  $\otimes$  to fold the siblings and  $\oplus$  to fold a parent and its children. By definition, the operator  $\otimes$  must be an associative operator.

$$\begin{aligned} \text{reduce}_r (\oplus) (\otimes) (RNode\ a\ []) &= a \oplus \iota_{\otimes} \\ \text{reduce}_r (\oplus) (\otimes) (RNode\ a\ [t_1, \dots, t_n]) \\ &= a \oplus ((\text{reduce}_r (\oplus) (\otimes) t_1) \otimes \dots \otimes (\text{reduce}_r (\oplus) (\otimes) t_n)) \end{aligned}$$

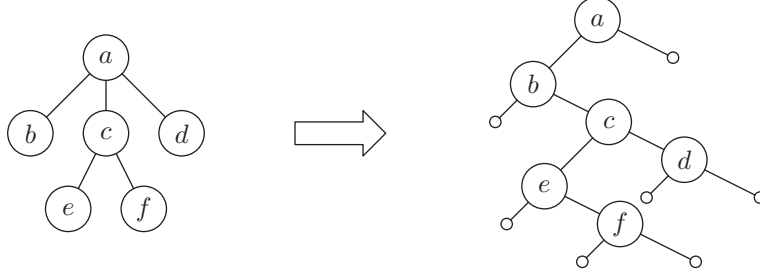
There are many applications that can be written in the form of reduction above. Examples of them are computing the summation of nodes, computing the height of trees, XML serialization and the maximum connected-set sum problem.

#### 3.2 Binary-Tree Representation of Rose Trees

Efficient implementations of the tree contraction algorithms have been known for binary trees, for example, on EREW PRAM [1] and on hypercubes [9]. To utilize these parallel implementations, we represent rose trees in shape of binary trees as shown in Fig. 2.

In this representation, every internal node comes from a node in the original rose tree, and all leaves are dummy nodes. The left child of a node in the binary tree is its left-most child in the original rose tree, and the right child of a node in the binary tree is its next sibling in the rose tree. Let  $n$  be the number of nodes of the original rose tree, then the number of nodes of the binary tree turns out to be  $2n + 1$ , which guarantees the asymptotic cost when we utilize the tree contraction algorithms.





**Fig. 2.** The binary-tree representation of rose trees.

### 3.3 Parallelizing General Tree Reduction with Binary Tree Reduction

In general, we can compute the reduction on rose trees by the reduction on binary trees with function  $k$  defined as  $k n l' r' = (n \oplus l') \otimes r'$ , after substituting  $\iota_{\otimes}$  for every leaf. As seen in the previous section, it is required to find a closed form of the functions to guarantee the correctness of the tree contraction algorithms. In the following of this section, we formalize the conditions for parallelizing the reductions and show a closed form of the functions for each condition.

First, we consider the most simple case where the operator  $\oplus$  is the same as the associative operator  $\otimes$ . In this case, we can use function form

$$\lambda l' r'. a \otimes l' \otimes r' \otimes b$$

as a closed form of the functions, where  $a$  and  $b$  are certain parameters.

Secondly, we consider the case where the two operators are different. It is well-known that the distributivity plays an important role in parallelizing programs, and the second condition is that the operator  $\oplus$  is associative and distributive over  $\otimes$  and  $\otimes$  is also associative. Under this condition, we can define a closed form of functions as

$$\lambda l' r'. (a \oplus l') \otimes (b \oplus r') \otimes e$$

where  $a$ ,  $b$ , and  $e$  are certain parameters.

Then we consider the case where the distributivity hold in the opposite pair of operators, in other words, the  $\otimes$  is distributive over  $\oplus$ . Unfortunately, there are still many reductions where  $\otimes$  is not distributive over  $\oplus$  because of the mismatch of the types. We apply the extended-distributive property [7] to generalize the condition to capture a wide range of reductions. The third condition is, therefore, the operator  $\otimes$  is both associative and extended-distributive over the operator  $\oplus$ . Under this condition, we can define a closed form of functions as

$$\lambda l' r'. a \oplus (b \otimes (c \oplus l') \otimes r' \otimes d)$$

where  $a$ ,  $b$ ,  $c$ , and  $d$  are certain parameters.

We summarize this section as the following theorem.

**Theorem 1.** The reduction on rose trees can be parallelized in terms of the reduction on binary trees, if either of the following conditions is fulfilled.

- The operator  $\oplus$  is the same as the associative operator  $\otimes$ .
- The operators  $\oplus$  and  $\otimes$  construct an algebraic semi-ring.
- The operator  $\otimes$  is both associative and extended-distributive over the operator  $\oplus$ .

*Proof:* We can show the correctness of the computation on binary trees by the induction on the structure of binary trees. We can prove the correctness of the tree contraction algorithms by calculating the composition of functions above and verifying the closure property.  $\square$

## 4 Implementing General Tree Reduction in C++

We have implemented a parallel skeleton library named SkeTo. The SkeTo library is a library based on the theories of Constructive Algorithmics [4] and now provides the parallel skeletons (ready-made components) for lists, matrices, and binary trees. In the SkeTo library, we can implement the reduction on rose trees as a wrapper functions of the skeletons on binary trees based on the discussion in Section 3.

The binary tree skeletons in the SkeTo library take function objects for their argument functions. The function objects enables us to generate new functions by function composition or partial binding of some arguments in conjunction with the template mechanism in C++. Therefore, we can generate suitable functions for the binary tree skeletons from the functions for rose trees given as function objects.

In the implementation, the data structure of rose trees is dealt as binary trees, and the class for the rose tree structure (`dist_rose_tree`) has a member of the class for the binary tree (`dist_tree`) as shown in the following segment of the code.

```
template<typename A> class dist_rose_tree {
    dist_tree<A>* btree;
    ...
}
```

We implement the function objects that will be passed to the reduction on binary trees, such as the argument function  $k$ , the closed form, its composition, and mapping from the closed form to the actual value. In the following we show the definitions of function  $k$  (`reduce_ring_k`) and the closed form for the tree contraction (`reduce_ring_form`), and the function that lifts function  $k$  to the closed form (`reduce_ring_phi`), for the case when the two operators construct an algebraic semi-ring.

```
template<typename A, typename OP, typename OT>
struct reduce_ring_k : public skeleton::ternary_function<A, A, A, A> {
    OP op; OT ot; reduce_ring_k(OP op_, OT ot_) : op(op_), ot(ot_) {}
    A operator()(const A& n, const A& l, const A& r) const {
        return ot(op(n, l), r);
    }
};
template<typename A>
struct reduce_ring_t {
    A a, b, c; reduce_ring_t(A a_, A b_, A c_) : a(a_), b(b_), c(c_) {}
};
template<typename A>
struct reduce_ring_phi : public skeleton::unary_function<A, reduce_ring_t<A>> {
    A e_op, e_ot; reduce_ring_phi(A e_op_, A e_ot_) : e_op(e_op_), e_ot(e_ot_) {}
    reduce_ring_t<A> operator()(const A& x) const {
        return reduce_ring_t(x, e_op, e_ot);
    }
};
```

Now we can implement the reduction on rose trees by using the function objects defined as above. First, we call the `map` skeleton to substitute the unit of  $\otimes$  for each leaf, and then call the `reduce` skeleton to obtain the result.

```
template<typename A, typename OP, typename OT>
A reduce_ring(OP op, A e_op, OT ot, A e_ot,
              const dist_rose_tree<A>* tree) {
    dist_tree<A>* bt1 = tree_skeleton::map(f_const(ot), f_id(), tree->btree);
    A result = tree_skeleton::reduce(reduce_ring_k(op, ot),
                                    reduce_ring_phi(e_op, e_ot), ..., bt1);
}
```

```

    if (bt1) delete bt1;
    return result;
}

```

As seen so far, we can write a new library function for the reduction on rose trees without considering the lower-level problems such as send-receive in MPI.

## 5 Conclusion

In this paper, we have shown the specification of the reduction on rose trees and the parallelization by the tree contraction algorithms. We have formalized three conditions for parallelizing the reduction, which capture a wide range of the reduction algorithms.

Actually, we have formalized seven general computational patterns on rose trees and their implementation in terms of the binary tree skeletons (component). We have implemented the rose tree skeletons in our parallel skeleton library *SkeTo*. Please refer our web-pages [13] for more details and download the current version of the library.

We are currently formalizing the derivation method of efficient parallel programs in terms of the general computational patterns on rose trees.

## References

1. K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.
2. D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In *9th International Conference on High Performance Computing (HiPC 2002)*, LNCS 2552, pages 63–75, Bangalore, India, Dec 2002.
3. R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, January 1998.
4. R. S. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.
5. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
6. X. He. Efficient parallel algorithms for solving some tree problems. In *24th Allerton Conference on Communication, Control and Computing*, pages 777–786, 1986.
7. K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. In S. Gorlatch, editor, *4th International Workshop on “Constructive Methods for Parallel Programming” (CMPP 2004)*, pages 109–123, July 2004.
8. K. Matsuzaki, Z. Hu, and M. Takeichi. Design and implementation of general tree skeletons. Technical Report METR2005-30, Department of Mathematical Engineering, University of Tokyo, 2005.
9. E. W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distributed Computing*, 26(2):181–192, 1995.
10. E. W. Mayr and R. Werchner. Optimal tree construction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, 1997.
11. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985. IEEE Computer Society Press.
12. G. L. Miller and S.-H. Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, 1997.
13. SkeTo Project. SkeTo project homepage. <http://www.ip1.t.u-tokyo.ac.jp/sketo/>, 2005.

# Coccinelle : a language-based approach to managing the evolution of Linux device drivers

Julia L. Lawall  
DIKU, University of Copenhagen  
2100 Copenhagen Ø, Denmark  
E-mail: julia@diku.dk

Gilles Muller  
Ecole des Mines de Nantes INRIA, LINA  
44307 Nantes cedex 3, France  
E-mail: Gilles.Muller@emn.fr

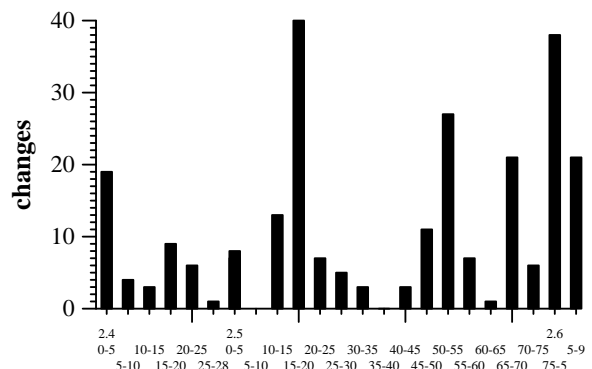
## 1 Introduction

The Linux operating system (OS) is becoming more and more widely used in government and industry at all levels. In areas ranging from servers, to desktop computing, to embedded systems, Linux is known for its configurability, low cost, and innovative applications. These advantages are due in part to the use of the open source model in developing Linux and accompanying applications. The key feature of this model is that users contribute to system development. Thus, bug fixes and new features are introduced as soon as there is a motivated user, and not when it becomes in the business interest of a company to do so. In practice, this model has led to the rapid evolution of Linux and accompanying applications.

Evolutions that affect the Application Program Interfaces (APIs) of the OS often trigger the need for *collateral evolutions* in OS services. Device drivers are particularly vulnerable to the need for such collateral evolutions, both because of their huge number and because of the multiplicity of their dependences on the rest of the OS. Furthermore, device drivers are typically developed by users who are more expert in the device than in the OS. This leads to a number of problems: The users who have sufficient expertise in a device to maintain a device driver often do not have sufficient expertise in the OS to infer the collateral evolution associated with a given API change. Even if the collateral evolution is understood, it may be complex and thus hard to apply correctly. Eventually, the user maintaining a driver may become unavailable, in which case, the driver can fall behind the rest of the OS. These issues contribute to the lack of continuity in support for devices across successive versions of Linux.

To illustrate the difficulty of keeping drivers up to date, we consider the case of the function `check_region`, used in driver initialization. In Linux 2.4.1, this function was called 322 times in 197 driver

files. Starting in Linux 2.4.2 (February 2001), the use of this function began to be eliminated, because changes in the driver initialization process implied that it could cause race conditions. Eliminating a call to `check_region` requires complex modifications that affect multiple parts of the code and require a non-trivial control-flow analysis. Indeed, discussion in Linux mailing lists indicates that the issues involved are not well understood. Accordingly, the elimination of `check_region` has proceeded very slowly, as shown by the graph below.



The evolution is not complete as of Linux 2.6.11 (March 2005), five years after the need for it first appeared, and bugs have been introduced along the way. This example highlights the critical need for a means of compensating for the diversity in expertise and commitment among Linux device driver developers. Specifically, an approach is required that provides (a) a mechanism to formally document collateral evolutions, and (b) a tool to aid developers in performing them.

## 2 Coccinelle

We are developing a system, Coccinelle, that provides a transformation language for precisely expressing

collateral evolutions and a transformation tool for applying these evolutions to device drivers. Coccinelle is being designed around the strategy of shifting the burden of collateral evolution from the driver maintainer to the OS developer who performs the original evolution of an OS API, and who thus understands this evolution best. In our vision, this OS developer uses the Coccinelle transformation language to write a semantic patch describing the transformations required to perform the collateral evolution in drivers. The developer then uses the Coccinelle transformation tool to apply the semantic patch to the drivers included in the Linux source distribution, to check and potentially refine the transformation.

To enable the OS developer to have confidence in the transformation process, we envision that the Coccinelle transformation tool will be interactive. Rather than transforming the code directly, Coccinelle will identify matching code patterns and propose the corresponding transformed code to the developer. The developer can either accept the change or modify the semantic patch to more accurately describe the required evolution. To further aid the developer, Coccinelle will identify partial matches, so that the developer can check for overlooked cases. When the OS developer has validated the semantic patch on the available drivers, he makes it publicly available for use by the maintainers of drivers outside the OS source distribution. These maintainers can also study the patch to understand the collateral evolution, and then use Coccinelle interactively to check that the assumptions made by the developer of the semantic patch hold for the given driver. Overall, Coccinelle will thus provide a means for formally documenting collateral evolutions and for easing the application of these evolutions to driver code.

Currently, we are carrying out a domain analysis, consisting of a thorough study of collateral evolution in device drivers in recent versions of Linux. This domain analysis will then form the basis of the design of the Coccinelle transformation language and transformation tool.

---

# BiXJ: A Java Library for Bidirectional XML Transformation

---

Dongxi Liu, Zhenjiang Hu, Masato Takeichi  
Kazuhiko Kakehi and Hao Wang

University of Tokyo

---

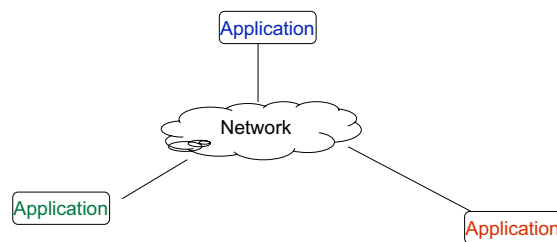
## Outline

- Motivation of this work.
- BiXJ in detail.
- Examples in BiXJ
  - Demos in <Oxygen />
- Translation of XQuery Core Expressions
- Related work and our contributions.
- Conclusions.

---

## XML Transformation

- XML -- the format for exchanging data.
- Heterogeneous applications require different format of data.
  - Transformation is pervasive for XML applications.

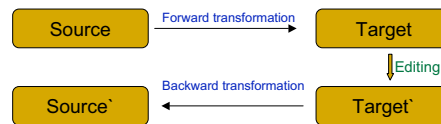


---

*Motivation of this work*

## Problem

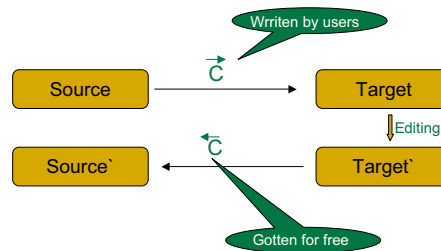
- The current transformation languages are only able to transform XML documents in **one direction**.
  - XQuery
  - XSLT
  - JAXB
  - ...
- In many cases, **bidirectional transformation** is desirable.
  - As illustrated by the following figure



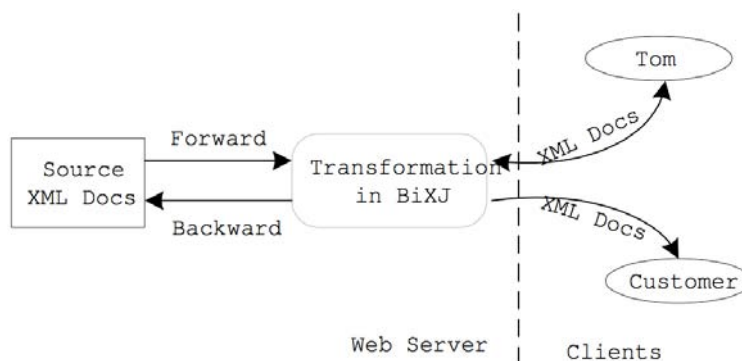
*Motivation of this work*

## Our approach

- BiXJ: a Java library for bidirectional XML transformation.
  - Writing forward transformation, users can get corresponding backward transformation for free.

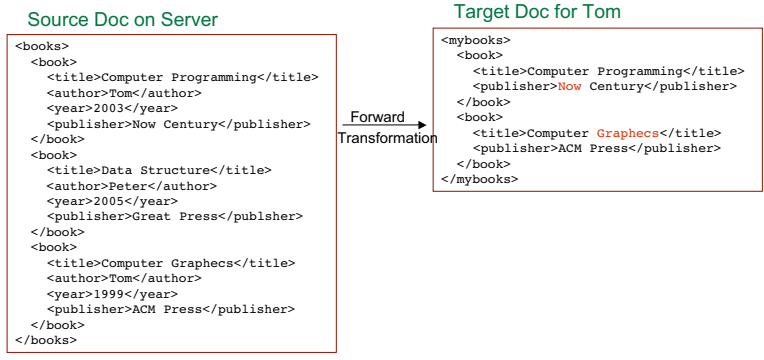


## A Sample Scenario of Using BiXJ (1)



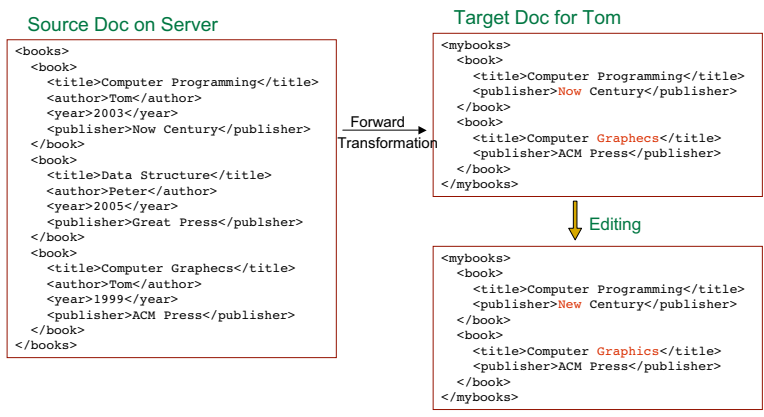
*Motivation of this work*

## A Sample Scenario of Using BiXJ (2)



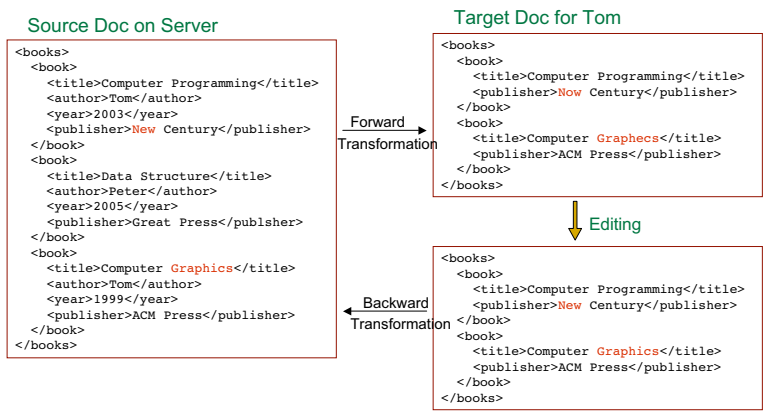
Motivation of this work

## A Sample Scenario of Using BiXJ (2)



Motivation of this work

## A Sample Scenario of Using BiXJ (2)

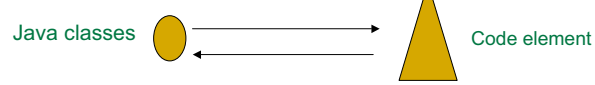


Motivation of this work



## Transformations in BiXJ

- Two ways to represent transformations in BiXJ
  - Java classes vs. Code element



- Syntax of Some Transformations in Code element

$X ::= PT \mid TC$

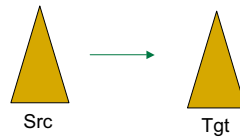
$PT ::= \langle xid \ / \rangle \mid \langle xconst \ elm \ / xconst \rangle \mid \langle xhide \ / \rangle$   
|  $\langle xmodifyname \ nm \ / xmodifyname \rangle$   
|  $\langle xnewroot \ nm \ / xnewroot \rangle \mid \langle xdistribute \ n \ / xdistribute \rangle$   
|  $\langle xchildren \ nm \ / xchildren \rangle \mid \langle xcollapse \ nm \ / xcollapse \rangle$

$TC ::= \langle xseq \ X_1 \ X_2 \ \dots \ X_n \ / xseq \rangle \mid \langle xmap \ X \ / xmap \rangle$   
|  $\langle xzip \ X_1 \ X_2 \ \dots \ X_n \ / xzip \rangle \mid \langle XIf \ pred \ X_1 \ X_2 \ / XIf \rangle$

*BiXJ in detail*

## XID

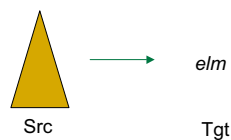
- $\langle xid \ / \rangle$



*BiXJ in detail*

## XConst

- $\langle xconst \ elm \ / xconst \rangle$

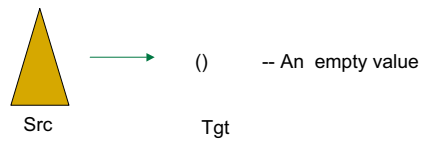


Note: the modification on *elm* will be reflected onto transformation.

*BiXJ in detail*

## XHide

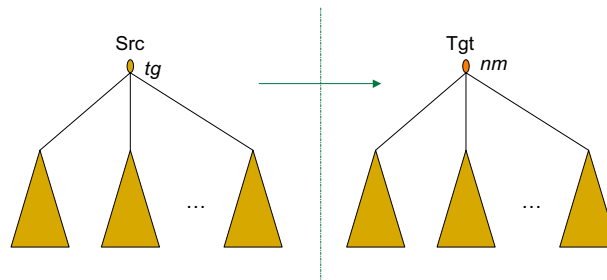
- `<xhide />`



*BiXJ in detail*

## XModifyName

- `<xmodifyname>nm</xmodifyname>`

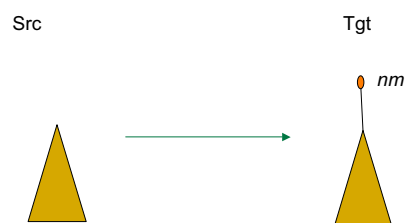


Note: the modification on tag *nm* will be reflected onto transformation.

*BiXJ in detail*

## XNewRoot

- `<xnewroot>nm</xnewroot>`

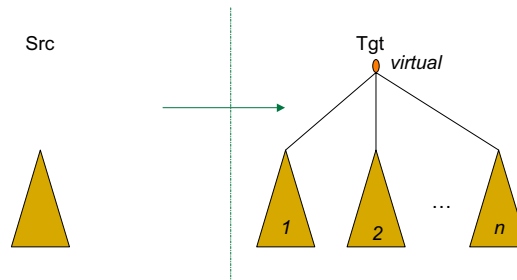


Note: the modification on tag *nm* will be reflected onto transformation.

*BiXJ in detail*

## XDistribute

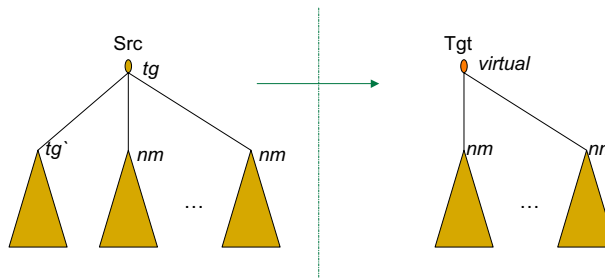
- `<xdistribute>n</xdistribute>`



*BiXJ in detail*

## XChildren

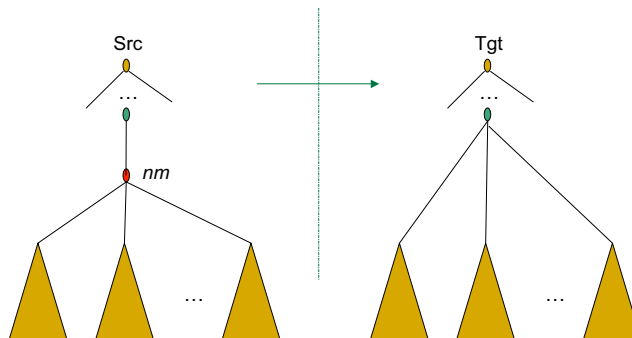
- `<xchildren>nm</xchildren>`



*BiXJ in detail*

## XCollapse

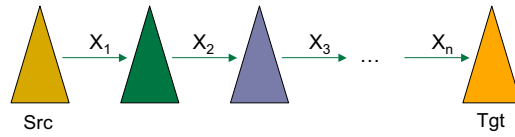
- `<xcollapse>nm</xcollapse>`



*BiXJ in detail*

## XSeq

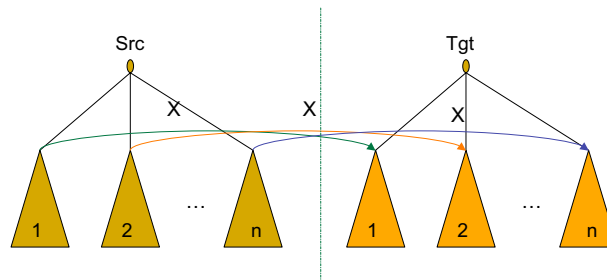
- `<xseq>X1 X2 ... Xn</xseq>`



*BiXJ in detail*

## XMap

- `<xmap>X</xmap>`

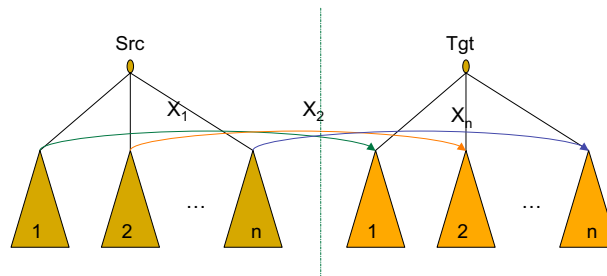


Note: Src and Tgt can have different numbers of child elements because  
1) Some child elements of Src are transformed into empty element ().  
2) New child elements are inserted in Tgt.

*BiXJ in detail*

## XZip

- `<xzip>X1 X2 ... Xn</xzip>`

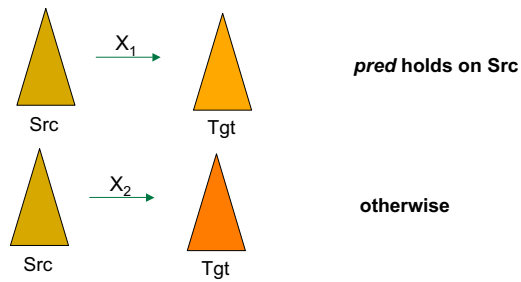


Note: Src and Tgt can have different numbers of child elements due to the same reason as XMap.

*BiXJ in detail*

## Xif

- `<xif>pred X1 X2</xif>`

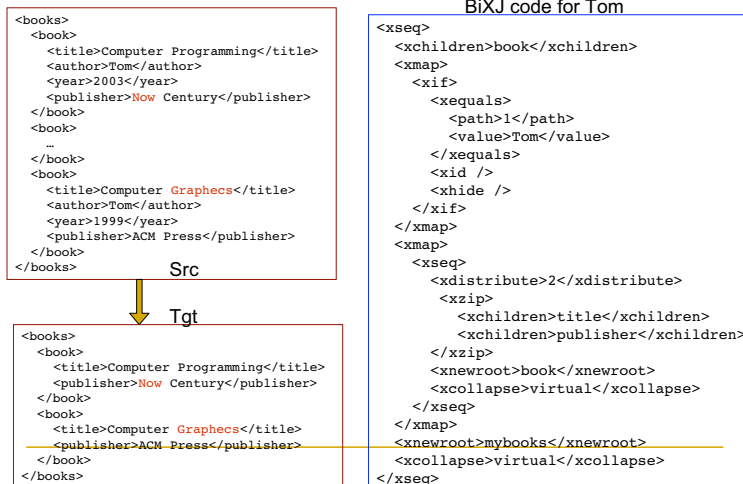


*BiXJ in detail*

## 3 Examples in BiXJ

- example in sample scenario
- example of bidirectionalizing XQuery
- example of bidirectionalizing XSLT

## Example #1 -- BiXJ Code for Scenario Example



## Example #2 -- Bidirectionalizing XQuery

- Typical XQuery expressions have a structure called “FLWR”.
  - For-Let-Where-Return
- The XQuery expression we have bidirectionalized as an example is:

```
<Books-of-Tom>
{
  for $l in doc("lib-src.xml")/lib return
  for $s in $l/shelf[category="Engineering"] return
  for $c in $s/cabinet return
  for $b in $c/book
  where $b/author = "Tom" and $b/price < 50 return
  <book>
  {
    $b/title,
    $b/price,
    <press>{$b/publisher/name/text()}</press>
  }
}</book>
}</Books-of-Tom>
```

*Find BiXJ code for this expression with Google in a same name paper!*

## Example #3 -- Bidirectionalizing XSLT

- A typical XSLT expression consists of transformation templates.
- The XSLT expression in the paper is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <title>Books-of-Tom</title>
      <body>
        <table>
          <tr>
            <th>title</th>
            <th>price</th>
            <th>press</th>
          </tr>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="/lib">
    <xsl:apply-templates select="shelf[category = 'Engineering']"/>
  </xsl:template>
```

*Find BiXJ code for this expression with Google in a same name paper!*

```
<xsl:template match="shelf">
  <xsl:apply-templates select="cabinet"/>
</xsl:template>
<xsl:template match="cabinet">
  <xsl:apply-templates select="book[author = 'Tom' and price < 50]"/>
</xsl:template>
<xsl:template match="book">
  <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="price"/></td>
    <td><xsl:value-of select="publisher/name"/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

## Demos in <Oxygen />

- We have implemented a plugin to combine BiXJ into <oxygen /> editor
  - helpful to develop BiXJ code in this editor.
  - helpful to experience Bidiretional XML transformation.

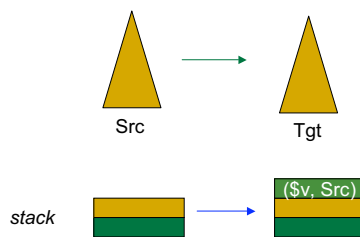
Take a look at demos in <Oxygen>!

## Translating XQuery Core Expressions

- Associate transformation with a context (stack)
- 3 operations to manage data in stack
  - `<xstore>$v</xstore>`
  - `<xload>$v</xload>`
  - `<xfree>$v</xfree>`

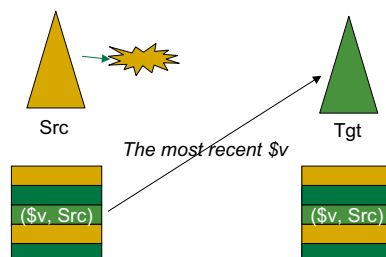
### XStore

- `<xstore>$v</xstore>`



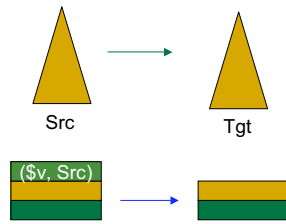
### XLoad

- `<xload>$v</xload>`



## XFree

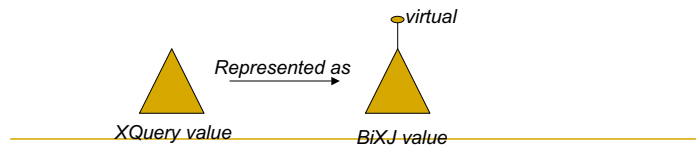
- `<xfree>$v</xfree>`



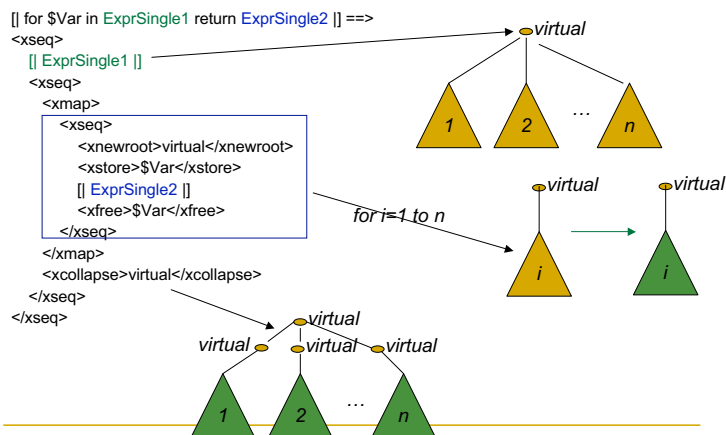
## Syntax of XQuery Core

```

Expr ::= ExprSingle (, ExprSingle)*
ExprSingle ::= for $Var in ExprSingle return ExprSingle
              | let $Var := ExprSingle return ExprSingle
              | if (Expr) then ExprSingle else ExprSingle
              | child::NCName | child::* | child::text()
              | descendant::NCName
              | ExprSingle InfixOp ExprSingle
              | Literal | $Var | (Expr?) | ElementConstructor
InfixOp ::= and | or | < | =
ElementConstructor ::= <NCName /> | <NCName> {Expr} </NCName>
Literal ::= NumericLiteral | StringLiteral
Var ::= NCName
    
```



## Transformation rules: one example





---

## Related Work and Our contribution

- J. Foster, etc. Combinators for Bi-Directional Tree Transformations. {ACM POPL 2005}
    - Based on un-ordered trees without repeated labels
    - Have not touched the expressiveness of their language (future work)
  - Z. Hu, etc. A Programmable Editor for Developing Structured Documents based on Bidirectional Transformation. {ACM PEPM 2004}
    - Not standard XML destructing operations, e.g., using integer list to represent path
    - Have not touched the expressiveness of their language, either.
  - Contributions in BiXJ w.r.t the existing work:
    - Support variable binding
    - Expressiveness is studied
      - Algorithm of translating expressions of XQuery Core to BiXJ.
    - Accepts the native data model.
    - Support standard XML processing operations.
    - ...
- 

---

## Conclusion and Future Work

- BiXJ
  - A Java Library for Bidirectional XML Transformation
  - Writing forward transformations, users can get backward transformation for free
- Future Work
  - Translating XSLT to BiXJ code automatically.
  - Implement insertion and deletion operations.
    - Some progress has been made-- deletion propagation.
  - Combining this work into a real XQuery Engine.

Thanks!

---



# Incrementalization of Axapta Report Programs

Presented by Michael Nissen



## Introduction

- Overview of the programming language FunSETL
- Overview of incrementalization
- Overview of the Axapta report "Financial Statement"
- Implementation of the Axapta report "Financial statement" in FunSETL
- Automatic incrementalization of the "Financial statement" in FunSETL
- Demo of the Financial statement
- Explanation of the computational differences.
- Advantages & Future work (summing up).



## FunSETL

- A simple functional language
- No recursion allowed, only iteration
- Made for the purpose of illustrating, that it is possible to make a system that automatically incrementalizes programs.

## FunSETL Example

```
fun sum5 (a : int, b : int) =  
  if 5 <= a then a + b else b  
  
fun summarize (xs : mset(int)) =  
  fold (fn (a, b) => sum5(a, b), 0, xs)
```

```
sumarize({1,5,7,3} as mset(int))
```

Computes :

```
sum5(3, sum5(7, sum5(5, sum5(1, 0))))
```

## FunSETL Language

$$t ::= t_{id} \mid \text{bool} \mid \text{int} \mid \text{real} \mid \text{date} \mid \text{string} \mid t_1 + t_2 \mid$$
$$\{lab : t_1, \dots, lab : t_n\} \mid \text{map}(t_1, t_2) \mid \text{mset}(t)$$
$$p ::= \text{fundef}_1, \dots, \text{fundef}_n$$
$$\text{fundef} := f(v_1 : t_1, \dots, v_n : t_n) = e$$

## FunSETL language overview

$$e ::= v \mid c \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid$$
$$\text{let } v = e_1 \text{ in } e_2 \mid \text{fold}(a, b \Rightarrow e_1) e_2 e_3 \mid$$
$$f(e_1, \dots, e_n) \mid \{lab_1 := e_1, \dots, lab_n := e_n\} \mid$$
$$\#lab(e) \mid e_1 \text{ with } e_2 \mid [e_1 \rightarrow e_1', \dots, e_n \rightarrow e_n'] \text{ as } t \mid$$
$$e_1[e_2] \mid e[e_1 \rightarrow e_1', \dots, e_n \rightarrow e_n']$$

The *fold* operation calculates the following :

$$f(e_n', f(e_{n-1}', \dots, f(e_2', f(e_1', e_2)) \dots)), \text{ where}$$
$$e_3 = \{e_1', e_2', \dots, e_n'\}$$
$$f = \lambda(a, b). e_1$$



## Definition and goal

Definition : Given a program  $f$  and an operation  $\oplus$ , then a program  $f'$  is called an incremental version of  $f$  under  $\oplus$ , if  $f'$  computes  $f(x \oplus y)$  by making use of  $f(x)$ , and the intermediate results of  $f(x)$ .

Goal : To make a system, that automatically make incremental versions of programs.



## Overview of incrementalization

We wish to incrementalize  $f$  with respect to  $\oplus$ .

1.  $f$  is transformed into  $\hat{f}$ , where  $\hat{f}$  returns the original return value from  $f$ , and the intermediate results of  $f$ .
2. The expression  $\hat{f}(x)$  is unfolded, and the return value of  $\hat{f}(x)$  is used as a cache  $C$  (the return value of  $\hat{f}(x)$  contains the return value of  $f(x)$ , and its intermediate results).
3. The expression  $\hat{f}(x \oplus y)$  is unfolded, and then there are made *simplifications* followed by *substituting values from the cache  $C$*  where possible. Calls to other functions is also incrementalized.
4. A new function  $\hat{f}'$  is defined, with parameters  $x, y$  and  $r$ , such that

$$r = \hat{f}(x) \Rightarrow \hat{f}'(x, y, r) = \hat{f}(x \oplus y)$$



## Simplifications

The only form of repetition in FunSETL is when using the fold expression. The following rewrite preserves the semantics of the expression :

$$\text{fold } (a, b \Rightarrow e_1) e_2 (S \text{ with } e) \rightarrow e_1[e/a, \text{fold } (a, b \Rightarrow e_1) e_2 S / b]$$

If  $r = \text{fold } (a, b \Rightarrow e_1) e_2 S$  is cached, we can simplify again to  $e_1[e/a, r/b]$



## Advantages & Future work

### Advantages:

- To get a program efficiency improvement
- To get simpler looking programs (source code)
- To reduce the errors induced by humans
- To gain a reduction in programming time

### Future work:

- Automatic feedback on result of incrementalization (have we gained speedups)



## Literature

- Daniel Brixen. *Inkrementelle metoder til REA-baseret rapportering*. DIKU, January 2005.
- Yanhong Annie Liu. *Incremental computation: A semantics-based systematic transformational approach*. Technical Report TR95-1551, Cornell University, Computer Science, October 31, 1995.
- Robert Paige. *Formal Differentiation: A Program Synthesis Technique*. UMI Research Press, 1981.

# Report of an implementation of a Semi-Inverter

Torben Ægidius Mogensen  
torbenm@diku.dk

DIKU-IST workshop 2005

## Abstract

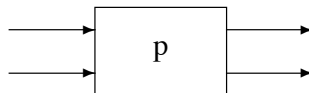
Semi-inversion is a generalisation of inversion: A semi-inverse of a program takes some of the inputs and outputs of the original program and returns the remaining inputs and outputs.

We report on an implementation of the semi-inversion method described in a paper [2] that will be presented at GPCE 2005. We will show some examples of semi-inversions made by the implementation and discuss limitations and possible extensions.

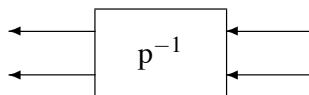
## 1 Summary of semi-inversion

### 1.1 Inversion vs. semi-inversion

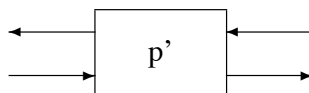
Original program:



Inversion is reversing all arrows:

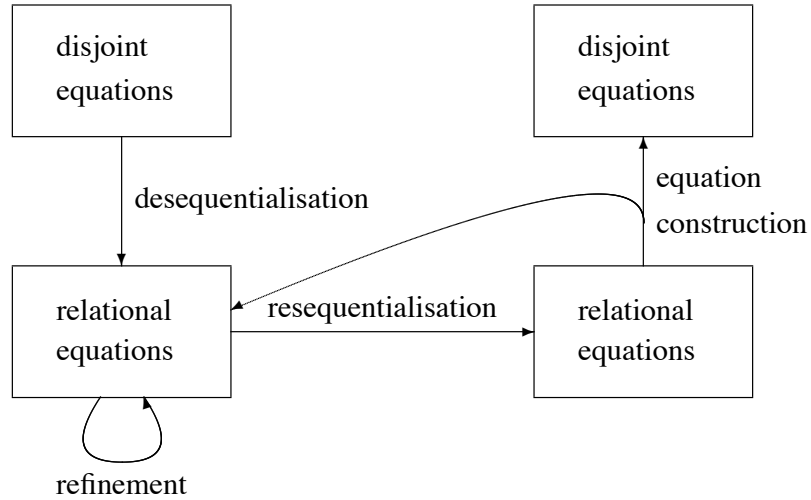


Semi-inversion is reversing *some* arrows:



### 1.2 The semi-inversion transformation

The steps of the semi-inversion transformation is shown in the following diagram:



The following sections will elaborate on each of these steps.

### 1.2.1 Desequentialisation

Make patterns and expressions into unordered set of relations.

$$\begin{aligned}
 f P \mid G &= E \\
 &\Downarrow \\
 f (P_1, P_2) &= R_1 \cup R_2 \cup R_3 \\
 \text{where} \\
 (P_1, R_1) &= I_p(P) \\
 (P_2, R_2) &= I_e(E) \\
 R_3 &= I_g(G)
 \end{aligned}$$

$$\begin{aligned}
 I_p &: \text{pattern} \rightarrow \text{var}^+ \times \text{relationset} \\
 I_e &: \text{expression} \rightarrow \text{var}^+ \times \text{relationset} \\
 I_g &: \text{guard} \rightarrow \text{relationset}
 \end{aligned}$$

Symmetric with respect to construction and deconstruction.

### 1.3 Refinement

- Combine operators:

$$z = \text{div}(x, y), v = \text{mod}(x, y) \text{ is replaced by } (z, v) = \text{divmod}(x, y)$$

- Split tuples:

$$(x, y) = (z, v) \text{ is replaced by } x = z, y = v$$

## 1.4 Resequentialisation

- Use dependency analysis to list relations in possible evaluation order.
- Operators use list of possible semi-inverses:

$$z = x + y \Rightarrow x = z - y, y = z - x$$

- Assume any subset of input/output to functions can define rest, backtrack if wrong.
- May fail.

## 1.5 Constructing equations

- Construct patterns from structural relations for new inputs.
- Make guards from non-structural relations for new inputs.
- Make expression from remaining relations.
- Equations for same function must be visibly disjoint by patterns and guards. If not, backtrack.

## 2 Detailed example: Permutation index

Original program: Takes  $n$  and  $i$  and returns list of  $1 \dots n$  permuted by permutation number  $i$ .

```
i2p (0, 0) = [];  
i2p (n, i) | 0 < n = insert (i2p (n-1, i/n), n, i%n);  
  
insert (xs, n, 0) = n : xs;  
insert (x : xs, n, i) | 0 < i && x /= n  
  = x : insert (xs, n, i-1);
```

Want: Program  $p2i$  that takes  $n$  and permuted list and returns the index  $i$ . Semi-inverter is given division:

```
i2p (1, 0, 1) = p2i;  
insert (0, 1, 0, 1) = find;
```

which indicates that the first argument and the result of  $i2p$  are known and that the semi-inverse is to be called  $p2i$ . The (optional) division for  $insert$  indicates that the second argument and the second half of the result are known and that the semi-inverse is to be called  $find$ .



## 2.1 Desequentialisation of i2p

```
i2p (x1,x2,x3) where
  {x1 = 0, x2 = 0, x3 = []}
i2p (n,i,x1) where
  {0<n, insert (x2,n,x3, x1), i2p (x4,x5, x2), x4 = -(n,1),
   x5 = /(i,n), x3 = %(i,n)}

insert (xs, n, x1, x2) where
  {x1 = 0, x2 = : (n,xs)}
insert (x1, n, i, x2) where
  {x1 = : (x,xs), x2 = : (x,x3), 0<i, x/=n,
   insert (xs,n,x4, x3), x4 = -(i,1)}
```

## 2.2 Refinement

We can combine / and % into /%:

```
i2p (x1,x2,x3) where
  {x1 = 0, x2 = 0, x3 = []}
i2p (n,i,x1) where
  {0<n, insert (x2,n,x3, x1), i2p (x4,x5, x2), x4 = -(n,1),
   x5 = /%(i,n)}

insert (xs, n, x1, x2) where
  {x1 = 0, x2 = : (n,xs)}
insert (x1, n, i, x2) where
  {x1 = : (x,xs), x2 = : (x,x3), 0<i, x/=n,
   insert (xs,n,x4, x3), x4 = -(i,1)}

(x5,x3) = /%(i,n) can semi-invert to i = m5*n+x3.
```

## 2.3 Resequentialisation

Order by data-dependence:

```
i2p (x1,x2,x3) where
  {x1 = 0, x3 = [], x2 = 0}
i2p (n,i,x1) where
  {0<n, x4 = -(n,1), insert (x2,n,x3, x1), i2p (x4,x5, x2),
   x5 = /%(i,n)}

insert (xs, n, x1, x2) where
  {x1 = 0, x2 = : (n,xs)}
insert (x1, n, i, x2) where
  {x2 = : (x,x3), x/=n, insert (xs,n,x4, x3),
   x1 = : (x,xs), x4 = -(i,1), 0<i}
```

## 2.4 Construction of equations

Create patterns, guards and expressions:

```

p2i (0, []) = 0;
p2i (n, x1) | 0 < n =
  let x4 = n-1 in
    let (x2, x3) = find (n, x1) in
      let x5 = p2i (x4, x2) in
        let i = x5*n+x3 in i;

find (n, n:xs) = (xs, 0);
find (n, x:x3) | x/=n =
  let (xs, x4) = find (n, x3) in
    let x1 = (x:xs) in
      let i = x4+1 in
        let True () = 0 < i in (x1, i);

```

## 2.5 Inlining some let-expressions

Inline let-binding if used once or constant:

```

p2i (0, []) = 0;
p2i (n, x1) | 0 < n =
  let (x2, x3) = find (n, x1) in p2i (n-1, x2)*n+x3;

find (n, n : xs) = (xs, 0);
find (n, x : x3_) | x/=n =
  let (xs, x4) = find (n, x3) in
    let i = x4+1 in
      let True () = 0 < i in (x : xs, i);

```

This is the final version as output by the semi-inverter (bar renaming of variables and removal of redundant parentheses).

## 3 Design details

The GPCE article leaves some details unspecified. Here are a few details of how the implementation handles these.

### 3.1 Refinement

Multiplication and division are combined, as shown above. Additionally,  $p = x+y$ ,  $q = x-y$  are combined to  $(p, q) = x \pm y$ , which when  $p$  and  $q$  are known can be inverted to  $(x, y) = ((p+q)/2, (p-q)/2)$ .

Equalities between tuples are split, and if a variable is equated with a tuple, the variable is replaced by the tuple where the variable is a result of or argument to an operator or function call.

### 3.2 Resequentialisation

When there are several possible relations that can be selected during resequentialisation, the following priorities are used:

1. Tests with all parameters known.
2. Primitive operators with sufficient instantiation for semi-inversion.
3. Calls to user-defined functions with instantiation corresponding to a desired semi-inverse.
4. Other calls to user-defined functions.

### 3.3 Backtracking

If semi-inverter fails to semi-invert a desired semi-inverse, it prints a message, marks the semi-inverse invalid and starts over. Sometimes, it may find other semi-inverses that can be used instead, otherwise the message may help the user rewrite the program to get better results.

### 3.4 Determining disjointness of equations

First, the variables in the two equations are renamed to make them use disjoint variables. Then, the patterns are unified. If this fails, the equations are disjoint, otherwise the unifying substitution is applied to the guards. If the conjunction of the guards can never be true, the equations are disjoint.

To test the guard, intervals of variables are maintained and if one becomes empty, the guard can't become true. A few additional cases of unsatisfiable constraints such as  $e/ = e$  are also considered.

There is room for improvement, as a conjunction like  $x < y \ \&\& \ y < x$  is not recognised as unsatisfiable.

## 4 A more ambitious example: Multiplication of binary numbers

If the result of a multiplication and one of its arguments are known, the other argument can be found by dividing the result by the known argument. For integers, this semi-inversion of multiplication is built into the semi-inverter, but if we represent numbers as lists of bits and multiplication as a recursive function over such lists, can the semi-inverter derive division from multiplication?

The usual binary multiplication algorithm can be described by the following recursive equations:

$$\begin{aligned}
 1 & \times y = y \\
 2x & \times y = 2(x \times y) \\
 (2x + 1) & \times y = 2(x \times y) + y
 \end{aligned}$$

The last equation uses addition, so the first step is to semi-invert addition of binary numbers. Since we require unique answers, we assume that numbers do not have leading zeroes (so zero is represented by an empty list of bits). It turns out that we, additionally, have to require that the second argument of the addition has at least as many bits as the

first in order to avoid overlapping equations in the semi-inverse.<sup>1</sup> Hence, addition (for little-endian lists of bits) looks like:

```

add(m,n) = adc(m,n,0);

adc([],bs,0) = bs;
adc([],bs,1) = inc(bs);
adc(a:as,b:bs,c) =
  let (s,c1) = add3(a,b,c) in
    s:adc(as,bs,c1);

add3(0,0,0) = (0,0);
add3(0,0,1) = (1,0);
add3(0,1,0) = (1,0);
add3(0,1,1) = (0,1);
add3(1,0,0) = (1,0);
add3(1,0,1) = (0,1);
add3(1,1,0) = (0,1);
add3(1,1,1) = (1,1);

inc [] = [1];
inc (0:bs) | bs/=[] = 1:bs;
inc (1:bs) = 0 : inc bs;

```

Note that absence of leading zeroes is explicitly tested in `inc`. With the division:

```
add(1,0,1) = sub;
```

we get the following semi-inverse:

```

sub (m,e_23_) = (adc_1011 (m,0,e_23_));

adc_1011 ([],0,bs) = bs;
adc_1011 ([],1,e_26_) = (inc_01 e_26_);
adc_1011 (a : as,c,s : e_30_) =
  let (b,c1) = (add3_10110 (a,c,s)) in b : (adc_1011 (as,c1,e_30_));

add3_10110 (0,0,0) = (0,0);
add3_10110 (0,1,1) = (0,0);
add3_10110 (0,0,1) = (1,0);
add3_10110 (0,1,0) = (1,1);
add3_10110 (1,0,1) = (0,0);
add3_10110 (1,1,0) = (0,1);
add3_10110 (1,0,0) = (1,1);
add3_10110 (1,1,1) = (1,1);

inc_01 [1] = [];
inc_01 1 : bs | bs/=[] = 0 : bs;
inc_01 0 : e_39_ = 1 : (inc_01 e_39_);

```

---

<sup>1</sup>An alternative is to require zero-extension to a fixed number of bits. This also works, but complicates the multiplication.

Note that the semi-inverter automatically finds the required semi-inverses of `adc`, `add3` and `inc`, and that the test for absence of leading zeroes in `inc` is now a guard that ensures disjointedness. `sub` is, in fact, reverse subtraction, as the first argument is subtracted from the second.

Getting multiplication to semi-invert is a bit more tricky. After a few tries, we come up with:

```
mul(0:as,bs) = 0:mul(as,bs);
mul([1],bs) = bs;
mul(1:as,0:bs) | as/=[] = 0:mul(1:as,bs);
mul(1:as,[1]) | as/=[] = 1:as;
mul(1:as,1:bs) | as/=[] && bs/=[] =
  1:add(as,mul(1:as,bs))
```

Note that we have done case-analysis of the second argument when the first is of the form  $2x + 1$ . Even so, the two last equations give overlapping patterns in the semi-inverse, so we have to add an assertion in the form of a superfluous test:

```
mul(0:as,bs) = 0:mul(as,bs);
mul([1],bs) = bs;
mul(1:as,0:bs) | as/=[] = 0:mul(1:as,bs);
mul(1:as,[1]) | as/=[] = 1:as;
mul(1:as,1:bs) | as/=[] && bs/=[] =
  let p = add(as,mul(1:as,bs)) in
  let True () = p/=as in 1:p;
```

With this test in place, we can successfully semi-invert with the division

```
mul(1,0,1) = div;
```

to

```
div(0 : as,0 : e_66_) = (div (as,e_66_));
div([1],bs) = bs;
div(1 : as,0 : e_74_) | as/=[] = 0 : (div (1 : as,e_74_));
div(1 : as,1 : as) | as/=[] = [1];
div(1 : as,1 : e_87_) | as/=[] && e_87_/=as =
  let bs = (div (1 : as,(add_101 (as,e_87_))) in
  let (True ()) = (bs/=[] in 1 : bs;
```

where `add_101` is equivalent to `sub`. As with `sub`, the arguments to `div` are reversed: The second argument is divided by the first. The nonlinear pattern in the penultimate equation and the comparison in the last equation together correspond to the test that in the traditional algorithm for binary division stops the repeated doubling of the divisor. Since the above requires the divisor to divide evenly into the other argument (otherwise, it wouldn't be a true semi-inverse), the test is here for equality rather than less-than-or-equal.

## 5 Conclusion

Other examples that work successfully include a variant of the permutation function, where it is used as a simple encryption function that uses the key to permute the list of characters. The semi-inverse is the corresponding decrypter. But some simple programs, such as list reversal using an accumulating parameter, defy semi-inversion with this method, so stronger methods like those in Glück and Kawabe's LR-parsing inspired inversion [1] need to be investigated.

As the multiplication example shows, it is sometimes necessary to rewrite programs and add assertions to get successful semi-inversion, similarly to how you sometimes need to rewrite programs to get good results with partial evaluation.

## References

- [1] Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Yuki Yoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming. Proceedings*, LNCS 2998, pages 291–306. Springer-Verlag, 2004.
- [2] Torben Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE'05*, Lecture Notes in Computer Science 3676, pages 189–204. Springer-Verlag, 2005.

# Streamlining Functional XML Processing\*

Keisuke Nakano

Department of Mathematical Informatics, University of Tokyo  
Bunkyo-ku, Tokyo, 113-8656, Japan  
ksk@mist.i.u-tokyo.ac.jp

## Abstract

Since an XML document has tree structure, XML transformations are ordinarily defined as recursive functions over the tree. Their direct implementation often causes inefficient memory usage because the input XML tree needs to be completely stored in memory. In contrast, XML stream processing can minimize the memory usage and execution time since it begins to output the transformation result before reading the whole input. However, it is much harder to write the XML transformation program in stream processing style than in functional style because stream processing requires stateful programming. In this paper, we propose a method for automatic derivation of XML stream processor from XML tree transformation written in functional style. We use an extension of macro forest transducers as a model of functional XML processing. Since an XML parser is represented by (infinitary) top-down tree transducer, the automatic derivation of XML stream processor is based on the composition of the top-down tree transducer and the extension of macro forest transducers.

## 1 Introduction

Since an XML document has tree structure, it is natural to define XML transformations as recursive functions over the tree. Such a style will be called *functional XML processing*. Several XML transformation languages [8, 2, 22] have been presented in this style, in which programs are recursive functions over *forests* that are sequences of labeled trees. This is mainly because each node in an XML tree can have an arbitrary number of children.

Forests are defined by

$$f ::= \sigma[f] f \mid \%[str] f \mid ()$$

where we write  $\sigma[f_1] f_2$  for a sequence whose head is a  $\sigma$ -labeled tree with a child forest  $f_1$  and tail is a sibling forest  $f_2$ ,  $\%[str] f$  for a text node of strings  $s$  which has a sibling forest  $f$ , and  $()$  for the empty sequence. For simplicity, we ignore attributes and allow the root to have sibling labeled trees. Text nodes are represented by a labeled tree with no child. For instance, an XML fragment

```
<p> XML is <em>forest</em>. </p>
```

are represented by

```
p[ %[ XML is ] em[ %[forest] ] ]
```

Perst and Seidl [17] recently presented *Macro forest transducers* (mft) which can be regarded as programs based on recursive XML Transformation. Roughly speaking, mft's can deal with recursive programs over forests<sup>1</sup>. A mft specifies a forest transformation by defining mutual recursive functions over forests with accumulating parameters. For example, we consider a simple XML transformation which creates a corresponding XHTML code and adds an index including all **key** elements before a postscript paragraph at the end of the input article. The transformation program is defined by a mft shown in Figure 1. The main function **Main** transforms an XML

---

\*This work is partially supported by the *Comprehensive Development of e-Society Foundation Software* program of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

<sup>1</sup>This paper deals with mft's extended for text nodes.

```

Main(article[$x1]$x2) = html[ head[ Title($x1) body[ InArticle($x1, () ) ] ] ] ()

Title(title[$x1]$x2) = title[$x1]

InArticle(title[$x1]$x2, $y1) = h1[$x1] InArticle($x2, $y1)
InArticle(para[$x1]$x2, $y1) = p[Key2Em($x1)] InArticle($x2, $y1 AllKeys($x1))
InArticle(postscript[$x1]$x2, $y1) = h2[%[Index]] ul[$y1] h2[%[Postscript]] $x1

Key2Em(key[$x1]$x2) = em[$x1] Key2Em($x2)
Key2Em(%[$s] x) = %[$s] Key2Em($x)
Key2Em(()) = ()

AllKeys(key[$x1]$x2) = li[$x1] AllKeys($x2)
AllKeys(%[$s] $x) = AllKeys($x)
AllKeys(()) = ()

```

Figure 1: Example of a mft-style XML transformation program

```

<article>
  <title>MFT</title>
  <para> XML is <key>forest</key>. </para>
  <para> <key>MFT</key> transforms forests. </para>
  <para> MFT transforms XML. </para>
  <postscript> MFT is quite expressive. </postscript>
</article>

```

into an XML

```

<html>
  <head><title>MFT</title></head>
  <body>
    <h1>MFT</h1>
    <p> XML is <em>forest</em>. </p>
    <p> <em>MFT</em> transforms forests. </p>
    <p> MFT transforms XML. </p>
    <h2>Index</h2>
    <ul> <li>forest</li> <li>MFT</li> </ul>
    <h2>Postscript</h2>
    <p> MFT is quite expressive. </p>
  </body>
</html>

```

using two auxiliary functions `AddKeys` and `AllKeys`. The function `Main` matches the argument with a pattern `article[x1]x2` and call the function `AddKeys` with the sub-forest *x*<sub>1</sub> and an extra argument `()`. The function `AddKeys` collects all keys in `para` elements and add them as an `index` element before a `copyright` element. The function `AllKeys` collects all keys occurred in a given forest. The pattern `_` matches any values. When the first argument of `AddKeys` matches with a pattern `para[x1]x2`, the function `AllKeys` is called with *x*<sub>1</sub> and the result is accumulated in the second argument of `AddKeys`.

In a mft-style XML transformation, only the first argument of every function is matched with several patterns as a forest. The rest of arguments can be used for accumulating parameters such as `y1` in the definition of `AddKeys` in the above example. Though some functions may be partial, they can be extended to total functions just by adding a rule `F (_, ...) = ()` with a wild-card pattern `_`.

Though a recursive XML processing is a handy style for XML transformations over forests, it frequently causes inefficiency of memory usage and execution time because the entire XML stream has to be read and passed to construct the complete input tree before the computation takes place. It is quite harmful in particular when the input is extremely long.

XML stream processing improves the efficiency such as [21]. It minimizes memory usage and execution time by not storing trees in memory. An XML stream processor begins to output the transformation result before reading the whole input. A program written in stream processing style simply consists of initial buffered value *v*<sub>0</sub> and event-associated function  $\mathcal{P}$  which takes the current buffered value and an input event and returns a value to be buffered and a part of output where an input event is  $\langle\sigma\rangle$ ,  $\langle/\sigma\rangle$ , *str* or the end-of-file event EOF.



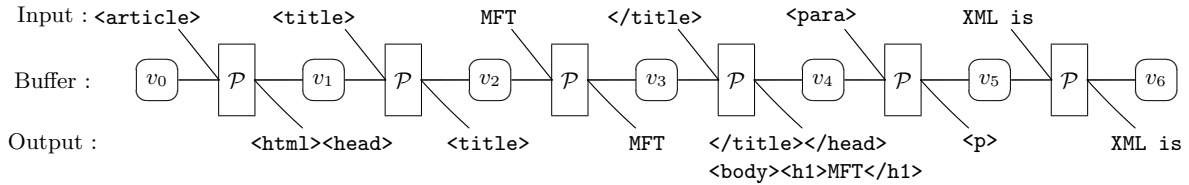


Figure 2: Example of stream processing flow

XML processing proceeds with updating a buffer as shown in Figure 2. First, the buffer has the initial value  $v_0$ . After that it processes the input stream depending on the event and the current buffered value for each event read. Now the current event is the begin tag `<article>` and the current buffered value is  $v_0$ . Then the function  $\mathcal{P}$  is called with the arguments the event `<article>` and the buffered value  $v_0$ . In the result, a part of the transformation result `<html><head>` is output and the buffered value is updated to  $v_1$ . When the next event `<title>` is read the function  $\mathcal{P}$  is called in a similar way. Note that enough information should be stored in the buffer since a stream processor cannot backtrack on the input stream in general. For example,  $v_3$  should comprehend all child nodes of `title` element so that the stream processor can output an `h1` element with these nodes.

While stream processing saves memory usage and execution time, it is much harder to write a program in stream processing style than in functional XML processing style because complicated stateful programming is required.

This paper presents a method to automatically derive the XML stream processor from an XML transformation program written as recursive XML processing. To be more precise we give a method to obtain an XML stream processing program from a mft. The method is based on the composition of a top-down tree transducer (tdtt) and a mft. The tdtt represents an XML parser which transforms XML streams into XML forests. Though we need a stack in order to parse XMLs by tdtt [15, 14], it can be simulated as an infinitary tdtt, that is a tdtt with infinite number of states. The composition is done in a way similar to that of a (finitary) tdtt and a mtt presented by Engelfriet and Vogler [5]. Though it has not been proved that an infinitary tdtt and a mft can be composed by their method, we directly prove that the XML stream processor obtained by our method behaves equivalently to the original mft in this paper.

## Related Work

Several researchers have discussed the automatic derivation of XML stream processors from declarative programs. Most of them, however, deals with only query languages including XPath [1, 4, 6, 7] and a subset of XQuery [11]. These querying languages are not expressive enough to specify XML transformation. For example, they could not define the structure-preserved transformation, e.g., renaming the label `a` to `b`. In recursive functional style, we can easily deal with this kind of transformation.

The key idea of our framework was presented in the author's preceding work [15, 16]. The previous work is based on the composition of (stack-)attributed tree transducers [14]. The author has released the XML transformation language XTISP [15, 13]. All programs definable in XTISP can be translated into attributed tree transducers. It is well known that attributed tree transducers are less expressive than macro tree transducers [5], i.e., our result in this paper is more powerful than before. Moreover, since the previous framework [15] does not give the formal model of stream processors, some part of the implementation of XTISP is ad-hoc and that contains inefficient evaluation.

Kiselyov [9] gave an XML parser with a general folding function `foldts` over rose trees. They define an XML transformation by applying three actions `fup`, `fdown` and `fhere` to `foldts`. These actions specify how to accumulate the seed value. This programming style is not user-friendly and many function closures are stored during the processing. Furthermore, his framework does not mention whether the processor can output a part of the result when reading a single XML event, e.g., a begin tag `<a>`.

STX [3] is a template-based XML transformation language that operates on stream of SAX [21] events. While the programmers can define the XML transformation program as well as XSLT [22], they have to explicitly write when and how to store the temporary information like stream processing style.

TransformX presented by Scherzinger and Kemper [18] gives the framework for syntax-directed transformations of XML streams. We can obtain XML stream processors by defining a kind of attribute grammar on the regular tree of the type schema for inputs. Even in their framework, however, we must still keep in mind which

information should be buffered before and after reading each subtree in the input.

Kodama, Suenaga, Kobayashi and Yonezawa [10, 19] propose a translation method from tree processing programs to XML stream processors where the programmer does not have to consider which information should be buffered. Their tree processing language only deals with binary trees which can be easily parsed without end tags, that is rather far from practical XML transformation languages.

## Outline

In Section 2, we introduce a simplified model of XML documents and macro forest transducers. In Section 3 we give the formal model of XML stream processors and its derivation from our transducers. We discuss an extension of our framework for applying the existing functional XML transformation languages in Section 4. Finally Section 5 concludes the paper.

## 2 Model of XML and its Transformation

This section formalizes a model of XML documents. For simplicity, we deal with a simplified model of XML documents. Firstly, we deal with only element nodes. Our framework is easily extended for other kinds of nodes, such as text nodes and attributes. Secondly, we assume that the input XML is *well-formed*, i.e., all begin/end tags are balanced. Therefore, in the input and output, we ignore the names of end tags. The names can be recovered by keeping a stack of names whose size coincides with the depth of the XML tree.

### 2.1 Trees, Forests and XML Streams

Let  $\Sigma$  be an alphabet. Then  $\Sigma$ -trees and  $\Sigma$ -forests are defined by the following syntax:

$$t ::= \sigma[f] \mid \%[\sigma] \qquad f ::= () \mid tf$$

where  $\sigma \in \Sigma$  and  $()$  denotes the empty forest.  $\Sigma$ -forest is also called  $\Sigma$ -hedge [12] and seen as a list of rose trees. Every tree  $t \in \mathcal{T}_\Sigma$  can be seen as a forest  $t() \in \mathcal{F}_\Sigma$  even if it is written as  $t$ . We denote  $\mathcal{T}_\Sigma$  and  $\mathcal{F}_\Sigma$  for sets of  $\Sigma$ -trees and  $\Sigma$ -forests, respectively. A  $\Sigma$ -tree  $\mathbf{a}[\%[\mathbf{bar}] \mathbf{b}[\%[\mathbf{foo}] ()] ()]$  with certain  $\Sigma$  corresponds to an XML fragment  $\langle \mathbf{a} \rangle \mathbf{bar} \langle \mathbf{b} \rangle \mathbf{foo} \langle / \mathbf{b} \rangle \langle / \mathbf{a} \rangle$ . For two forests  $f_1, f_2 \in \mathcal{F}_\Sigma$ , we write  $f_1 f_2$  for a  $\Sigma$ -forest  $t_1 \dots t_n u_1 \dots u_m ()$  where  $f_1 = t_1 \dots t_n ()$  and  $f_2 = u_1 \dots u_m ()$ .

An XML stream is modeled by a sequence of named begin/end tags and texts. The model of XML stream is defined by a sequence of  $\Sigma$ -events which is an alphabet  $\{\langle \sigma \rangle \mid \sigma \in \Sigma\} \cup \{\langle / \sigma \rangle \mid \sigma \in \Sigma\} \cup \Sigma$  denoted by  $\Sigma_{\langle / \rangle}$ , provided that the sequence is well-formed, i.e., every begin tag has a corresponding end tag and vice versa. For instance, an XML fragment  $\langle \mathbf{a} \rangle \mathbf{bar} \langle \mathbf{b} \rangle \mathbf{foo} \langle / \mathbf{b} \rangle \langle / \mathbf{a} \rangle$  is represented in our model by the sequence of six symbols,  $\langle \mathbf{a} \rangle$ ,  $\mathbf{bar}$ ,  $\langle \mathbf{b} \rangle$ ,  $\mathbf{foo}$ ,  $\langle / \mathbf{b} \rangle$  and  $\langle / \mathbf{a} \rangle$ . We denote by  $\Sigma_{\langle / \rangle}^*$  a set of well-formed sequences of  $\Sigma$ -events and denote by  $\varepsilon$  the empty sequence. The set  $\Sigma_{\langle / \rangle}^*$  is a subset of  $\mathcal{F}_{\Sigma_{\langle / \rangle}}$  where every tree has no child. The symbol EOF denotes the end of an XML stream, which is also regarded as an event. We write  $\Sigma_{\langle / \rangle \text{EOF}}$  for  $\Sigma_{\langle / \rangle} \cup \{\text{EOF}\}$ .

Let  $\Sigma$  be an alphabet. The *streaming* of a forest is the function  $[\_ ] : \mathcal{F}_\Sigma \rightarrow \Sigma_{\langle / \rangle}$  defined by

$$[\sigma[f_1]f_2] = \langle \sigma \rangle [f_1] \langle / \sigma \rangle [f_2] \qquad [\%[\sigma]f] = \sigma [f] \qquad [()] = \varepsilon.$$

For example,  $[\mathbf{a}[\%[\mathbf{bar}] \mathbf{b}[\%[\mathbf{foo}] ()] ()]] = \langle \mathbf{a} \rangle \mathbf{bar} \langle \mathbf{b} \rangle \mathbf{foo} \langle / \mathbf{b} \rangle \langle / \mathbf{a} \rangle$ .

### 2.2 Macro Forest Transducers

*Macro forest transducers* (for short, mft) were proposed by Perst and Seidl [17] to define transformations from forests to (sets of) forests. They extend *macro tree transducer* (for short, mtt) [5] with the concatenation operator for forests as a primitive. Let us write  $\mathbb{N}$  for the set of non-negative integers including 0.

**Definition 2.1** A macro forest transducer (mft) is a tuple  $M = (Q, \Sigma, \Delta, in, R)$ , where

- $Q$  is a finite set of ranked states whose ranks are obtained by  $rank : Q \rightarrow \mathbb{N} \setminus \{0\}$ ,
- $\Sigma$  and  $\Delta$  are alphabets with  $Q \cap (\Sigma \cup \Delta) = \emptyset$ , called the input alphabet and the output alphabet, respectively,
- $in \in Q$  is the initial ranked state,

- $R$  is a set of rules such that  $R = \bigcup_{q \in Q} R_q$  with sets  $R_q$  of  $q$ -rules of the form  $q(pat, y_1, \dots, y_n) \rightarrow rhs$  with  $rank(q) = n + 1$  and variables  $y_i$  where
  - $pat$  is either  $()$  or  $\sigma[x_1]x_2$  with  $\sigma \in \Sigma$ ,
  - $rhs$  ranges over expressions defined by

$$rhs ::= q'(x_i, rhs, \dots, rhs) \mid y_j \mid () \mid \delta[rhs] \mid \%[\delta] \mid rhs \ rhs$$

with  $q' \in Q$ ,  $\delta \in \Delta$ ,  $i = 1, 2$  and  $j = 1, \dots, rank(q') - 1$ . Additionally, no variable  $x_i$  occurs in  $rhs$  when  $pat = ()$ .

Next we define the semantics of mft's such that every state is translated into a function with accumulating parameters following [17].

**Definition 2.2** Let  $M = (Q, \Sigma, \Delta, in, R)$  be a mft and  $f \in \mathcal{F}_\Sigma$ . The semantics of states  $q \in Q$  with  $n = rank(q)$  is given by the function  $\llbracket q \rrbracket : \mathcal{F}_\Sigma \times (\mathcal{F}_\Delta)^n \rightarrow \mathcal{F}_\Delta$ . The functions are inductively defined by  $q$ -rules in  $M$  as follows:

- $\llbracket q \rrbracket ((), \varphi_1, \dots, \varphi_n) = \llbracket rhs \rrbracket_\rho$  where  $(q((), y_1, \dots, y_n) \rightarrow rhs) \in R$  and  $\rho(y_j) = \varphi_j$  for  $j = 1, \dots, n$ ,
- $\llbracket q \rrbracket (\sigma[\omega_1]\omega_2, \varphi_1, \dots, \varphi_n) = \llbracket rhs \rrbracket_\rho$  where  $(q(\sigma[x_1]x_2, y_1, \dots, y_n) \rightarrow rhs) \in R$ ,  $\rho(x_i) = \omega_i$  for  $i = 1, 2$  and  $\rho(y_j) = \varphi_j$  for  $j = 1, \dots, n$ ,
- $\llbracket q \rrbracket (\%[\sigma]\omega, \varphi_1, \dots, \varphi_n) = \llbracket rhs \rrbracket_\rho$  where  $(q(\%[\sigma]x, y_1, \dots, y_n) \rightarrow rhs) \in R$ ,  $\rho(x) = \omega$  for  $i = 1, 2$  and  $\rho(y_j) = \varphi_j$  for  $j = 1, \dots, n$ ,

where  $\llbracket \_ \rrbracket_\rho$  denotes the evaluation of a right-hand side expression for states with respect to the binding  $\rho$  of the formal parameters  $x_i$  and  $y_j$  that is defined by

$$\begin{aligned} \llbracket q'(x_i, rhs_1, \dots, rhs_m) \rrbracket_\rho &= \llbracket q' \rrbracket (\rho(x_i), \llbracket rhs_1 \rrbracket_\rho, \dots, \llbracket rhs_m \rrbracket_\rho) \\ \llbracket y_j \rrbracket_\rho &= \rho(y_j) & \llbracket () \rrbracket_\rho &= () \\ \llbracket \delta[rhs] \rrbracket_\rho &= \delta[\llbracket rhs \rrbracket_\rho] & \llbracket \%[\delta] \rrbracket_\rho &= \%[\delta] & \llbracket rhs \ rhs' \rrbracket_\rho &= \llbracket rhs \rrbracket_\rho \llbracket rhs' \rrbracket_\rho. \end{aligned}$$

Our definition is different from [17] in that we deal with text nodes explicitly and consider only deterministic mft's, i.e., every semantics of states ranges over a set of forests instead of a power set of them. Note that the semantics of a state is a partial function when there is no rule for the state and a certain pattern. For such uncovered state and pattern, we assume that the mft implicitly has rules whose right-hand side is a leaf. Then we can claim that the semantics is total. In the rest of paper, we deal with only total mft's though we may omit these additional rules.

The transformation of a forest  $f$  by an mft is defined by applying the semantics of the initial state to  $f$  and an adequate number of leaves.

**Definition 2.3** The transformation induced by a mft  $M = (Q, \Sigma, \Delta, in, R)$  is the function  $\tau_M : \mathcal{F}_\Sigma \rightarrow \mathcal{F}_\Delta$  defined by

$$\tau_M(f) = \llbracket in \rrbracket (f, (), \dots, ()).$$

We show two examples of XML transformation written in mft's. First example  $M_{htm}$  is an XML transformation shown in Section 1.

**Example 2.4** Let the mft  $M_{htm} = (Q, \Sigma, \Delta, Main, R)$  be defined by

$$\begin{aligned} Q &= \{Main, Title, InArticle, Key2Em, AllKeys, Copy\}, \\ \Sigma &= \Delta = (proper \ alphabet), \\ R &= \{ Main(\mathbf{article}[x_1]x_2) \rightarrow \mathbf{html}[\mathbf{head}[Title(x_1)]\mathbf{body}[InArticle(x_1, ())]()], \\ &\quad Title(\mathbf{title}[x_1]x_2) \rightarrow \mathbf{title}[Copy(x_1)], \\ &\quad InArticle(\mathbf{title}[x_1]x_2, y_1) \rightarrow \mathbf{h1}[Copy(x_1)]InArticle(x_2, y_1), \\ &\quad InArticle(\mathbf{para}[x_1]x_2, y_1) \rightarrow \mathbf{p}[Key2Em(x_1)]InArticle(x_2, y_1 AllKeys(x_1)), \\ &\quad InArticle(\mathbf{postscript}[x_1]x_2, y_1) \rightarrow \mathbf{h2}[\%[Index]] \mathbf{ul}[y_1] \mathbf{h2}[\%[Postscript]] Copy(x_1), \\ &\quad Key2Em(\mathbf{key}[x_1]x_2) \rightarrow \mathbf{em}[Copy(x_1)] Key2Em(x_2), \\ &\quad Key2Em(\%[\sigma]x) \rightarrow \%[\sigma]Key2Em(x) \quad (\sigma \in \Sigma), \quad Key2Em() \rightarrow (), \\ &\quad AllKeys(\mathbf{key}[x_1]x_2) \rightarrow \mathbf{li}[Copy(x_1)]AllKeys(x_2), \quad AllKeys(\%[\sigma]x) \rightarrow AllKeys(x) \quad (\sigma \in \Sigma), \\ &\quad AllKeys() \rightarrow (), \end{aligned}$$

$$Copy(\sigma[x_1]x_2) \rightarrow \sigma[Copy(x_1)]Copy(x_2) \quad (\sigma \in \Sigma), \quad Copy(\epsilon) \rightarrow \epsilon \}$$

Almost rules of  $M_{htm}$  are the same as the function definition in Figure 1. However, the variables matched with a pattern cannot occur in the right-hand side of rules except for the case where they are used as the first argument of the states according to the definition of *rhs*. For example, the right-hand side of the definition of `Title` is `title[$x1]` in Figure 1. The definition of mft's does not allow the expression `title[x1]` in a right-hand side of rules. The mft  $M_{htm}$  solves the problem by using a state *Copy* whose semantics is an identity function, i.e., we can use `title[Copy(x1)]` instead of `title[x1]`.

Second example of a mft represents an XML transformation which reverses all descendants of `rev` node in the input. For instance, when the input XML fragment is

```
<a>
  <rev><b><c></c><d></d></b><e></e></rev>
  <f><rev><g></g><h></h></rev></f>
</a>
```

the transformation returns

```
<a>
  <rev><e></e><b><d></d><c></c></b></rev>
  <f><rev><h></h><g></g></rev></f>
</a>
```

The transformation can be given by an mft with only two states.

**Example 2.5** Let the mft  $M_{mir} = (Q, \Sigma, \Delta, Main, R)$  be defined by

$$\begin{aligned}
Q &= \{Main, Rev\}, & \Sigma &= \Delta = (\text{proper alphabet}), \\
R &= \{ Main(\mathbf{rev}[x_1]x_2) \rightarrow \mathbf{rev}[Rev(x_1, \epsilon)]Main(x_2), \\
& Main(\sigma[x_1]x_2) \rightarrow \sigma[Main(x_1)]Main(x_2) \quad (\sigma \neq \mathbf{rev}), \quad Main(\%[\sigma]x) \rightarrow \%[\sigma]Main(x) \quad (\sigma \neq \mathbf{rev}), \\
& Main(\epsilon) \rightarrow \epsilon, \\
& Rev(\sigma[x_1]x_2, y_1) \rightarrow Rev(x_2, \sigma[Rev(x_1, \epsilon)]y_1) & Rev(\%[\sigma]x, y_1) \rightarrow Rev(x, \%[\sigma]y_1) \quad (\sigma \in \Sigma), \\
& Rev(\epsilon, y_1) \rightarrow y_1 \}.
\end{aligned}$$

### 3 XML Stream Processors and Its Derivation

This section presents a formal model of XML stream processors and its derivation method based on the composition of tree transducers. Since the set  $\Sigma_{\langle / \rangle}^*$  is a subset of  $\mathcal{F}_{\Sigma_{\langle / \rangle}}$ , XML stream processor (for short, xsp) can be defined in a way similar to the definition of tree transducers such as mft's.

#### 3.1 XML Stream Processors

An XML stream processor proceeds an XML transformation by updating the buffered value. In our framework, we consider a partially-evaluated result, called *temporary expression*, as the buffered value. The value will be the transformation result itself after all input events are read. Additionally, the stream processor can output a part of the transformation result by squeezing some decided output events at the head of the temporary expression before completing reading all input events.

Our XML stream processor consists of rules which specifies how to update the temporary expression

**Definition 3.1** An XML stream processor (*xsp*) is a tuple  $S = (Q, \Sigma, \Delta, in, R)$ , where

- $Q$  is a set of ranked states, which may be countably infinite and the rank for each state is obtained by  $rank : Q \rightarrow \mathbb{N}$ ,
- $\Sigma$  and  $\Delta$  are (finite) alphabets with  $Q \cap (\Sigma \cup \Delta) = \emptyset$ , called the input alphabet and the output alphabet, respectively,
- $in \in Q$  is the initial state,

- $R$  is a set of rules such that  $R = \{r_{(q,\chi)} \mid q \in Q, \chi \in \Sigma_{</>\text{EOF}}\}$  with  $(q, \chi)$ -rules  $r_{(q,\chi)}$  of the form

$$q(y_1, \dots, y_n) \xrightarrow{\chi} rhs$$

with variables  $y_j$  where  $n = \text{rank}(q)$  and  $rhs$  ranges over expressions defined by

$$rhs ::= q'(rhs, \dots, rhs) \mid y_j \mid \varepsilon \mid \langle \delta \rangle rhs \langle / \delta \rangle \mid \delta \mid rhs \ rhs$$

where  $q' \in Q$ ,  $\delta \in \Delta$  and  $j = 1, \dots, n$ . Additionally, the pattern  $q'(\_)$  does not occur in  $rhs$  for any  $q' \in Q$  when  $\chi = \text{EOF}$ .

### 3.2 Semantics of XML Stream Processors

The definition of semantics of states in an xsp is different from that of states in a mft because a rule of an xsp specifies how to update the temporary expression for each input event. Temporary expressions range over output XML streams with a number of unknown parts given by using states with arguments.

**Definition 3.2** Let  $S = (Q, \Sigma, \Delta, in, R)$  be an xsp. A temporary expression  $E$  for  $S$  is defined by the following syntax:

$$E ::= q(E, \dots, E) \mid \varepsilon \mid \langle \delta \rangle E \langle / \delta \rangle \mid \delta \mid E E$$

where  $q \in Q$ ,  $\delta \in \Delta$ . We denote the set of temporary expressions by  $\text{Tmp}_S$ .

The semantics of an xsp is defined by translating every rule of the xsp into a transition for temporary expressions.

**Definition 3.3** Let  $S = (Q, \Sigma, \Delta, in, R)$  be an xsp and  $s \in \Sigma_{</>}^*$ . The transition over  $\text{Tmp}_S$  for an input  $\Sigma$ -event is a function  $\langle \_ \rangle : \text{Tmp}_S \times \Sigma_{</>\text{EOF}} \rightarrow \text{Tmp}_S$ . The function is defined with another transition over  $\text{Tmp}_S$  which is a function  $\langle \_ \rangle : \text{Tmp}_S \times \Sigma_{</>\text{EOF}} \rightarrow \text{Tmp}_S$ . The definition use the evaluation function  $\llbracket \_ \rrbracket_\rho : rhs \rightarrow \text{Tmp}_S$  for right-hand side expressions with respect to the binding  $\rho$  of the formal parameters in the left-hand side. We give the definition as follows:

- $\langle \_ \rangle$  are defined by
  - $\langle q(E_1, \dots, E_n), \chi \rangle = \llbracket rhs \rrbracket_\rho$  where  $(q(y_1, \dots, y_n) \xrightarrow{\chi} rhs) \in R$  with  $q \in Q$ ,  $\chi \in \Sigma_{</>\text{EOF}}$ ,  $\rho(y_j) = \langle E_j, \chi \rangle$  for  $j = 1, \dots, n$ ,
  - $\langle \varepsilon, \chi \rangle = \varepsilon$ ,  $\langle \langle \delta \rangle E \langle / \delta \rangle, \chi \rangle = \langle \delta \rangle \langle E, \chi \rangle \langle / \delta \rangle$ , and  $\langle \delta, \chi \rangle = \delta$  where  $\delta \in \Delta$ ,
  - $\langle E E', \chi \rangle = \langle E, \chi \rangle \langle E', \chi \rangle$ ,
- $\llbracket \_ \rrbracket_\rho$  is defined by

$$\begin{aligned} \llbracket q'(rhs_1, \dots, rhs_m) \rrbracket_\rho &= q'(\llbracket rhs_1 \rrbracket_\rho, \dots, \llbracket rhs_m \rrbracket_\rho) \\ \llbracket y_j \rrbracket_\rho &= \rho(y_j) & \llbracket \varepsilon \rrbracket_\rho &= \varepsilon \\ \llbracket \langle \delta \rangle rhs \langle / \delta \rangle \rrbracket_\rho &= \langle \delta \rangle \llbracket rhs \rrbracket_\rho \langle / \delta \rangle & \llbracket \delta \rrbracket_\rho &= \delta & \llbracket rhs \ rhs' \rrbracket_\rho &= \llbracket rhs \rrbracket_\rho \llbracket rhs' \rrbracket_\rho. \end{aligned}$$

XML processing reads the input events one by one. For each reading step, the processor computes something with stored information and store a new information for the next step. The transition  $\langle \_ \rangle$  defines how the processor computes the next information for each input event. In our framework, the information is represented by a temporary expression. Let  $S = (Q, \Sigma, \Delta, in, R)$  be an xsp and  $\chi_1 \chi_2 \dots \chi_k$  be an XML stream with  $\chi_j \in \Sigma_{</>}$  for  $j = 1, 2, \dots, k$ . The initial information is represented by  $in(\varepsilon, \dots, \varepsilon)$ . When finding the end of the input XML stream, the transition for EOF is applied to the current information. The final information is the transformation result itself. Thus we obtain the transformation result by

$$\langle \langle \dots \langle \langle in(\varepsilon, \dots, \varepsilon), \chi_1 \rangle, \chi_2 \rangle, \dots, \chi_k \rangle, \text{EOF} \rangle. \quad (1)$$

This transformation is not what we require as XML processing, however, because the XML stream processor should output part of the result if possible before reading the whole input.

We give two definitions of transformation induced by an xsp. One is called *non-squeezing*. The definition is simply given as represented in (1). Another is called *squeezing*. The squeezing transformation achieve the

best result possible, that is, the output written so far always the largest that can be determined from the input read so far. Stream processing with squeezing is a desirable behavior of real XML stream processors which can start to output a part of the result for each input event. Since squeezing will collapse the syntax of temporary expressions, we define *collapsed temporary expressions*  $Tmp_S^\times$  with an xsp  $S$  by

$$E ::= q(E, \dots, E) \mid \varepsilon \mid \langle \delta \rangle E \mid \langle / \delta \rangle E \mid \delta E$$

where  $q$  is a state of  $S$  and  $\delta$  is an output symbol of  $S$ . We can easily confirm that  $Tmp_S \subset Tmp_S^\times$ .

**Definition 3.4** 1. The non-squeezing transformation induced by an xsp  $S = (Q, \Sigma, \Delta, in, R)$  is the function  $\tau_S : \Sigma_{\langle / \rangle}^* \rightarrow \Delta_{\langle / \rangle}^*$  defined by  $\tau_S(s) = \theta_S(in(\varepsilon, \dots, \varepsilon), s\mathbf{EOF})$  where

$$\theta_S(e, \varepsilon) = e \qquad \theta_S(e, \chi s) = \theta_S(\langle e, \chi \rangle, s)$$

for  $e \in Tmp_S$ .

2. The squeezing transformation induced by an xsp  $S = (Q, \Sigma, \Delta, in, R)$  is the function  $\tau_S : \Sigma_{\langle / \rangle}^* \rightarrow \Delta_{\langle / \rangle}^*$  defined by  $\tau_S(s) = \eta_S(in(\varepsilon, \dots, \varepsilon), s\mathbf{EOF}, \varepsilon)$  where for  $e \in Tmp_S$

$$\eta_S(e, \varepsilon, b) = be \qquad \eta_S(e, \chi s, b) = \eta_S(e', s, bs')$$

with  $(e', s') = sqz(\langle e, \chi \rangle)$  and a squeeze function  $sqz : Tmp_S^\times \rightarrow Tmp_S^\times \times \Delta_{\langle / \rangle}^*$  is defined by

$$\begin{aligned} sqz(q(e_1, \dots, e_n)) &= (q(e_1, \dots, e_n), \varepsilon) & sqz(\varepsilon) &= (\varepsilon, \varepsilon) & sqz(\langle \delta \rangle e_1) &= (e'_1, \langle \delta \rangle s'_1) \\ sqz(\langle / \delta \rangle e_1) &= (e'_1, \langle / \delta \rangle s'_1) & sqz(\delta e_1) &= (e'_1, \delta s'_1) & sqz(e_1 e_2) &= \begin{cases} (e'_2, s'_1 s'_2) & \text{if } e'_1 = \varepsilon \\ (e'_1 e_2, s'_1) & \text{otherwise} \end{cases} \end{aligned}$$

where  $(e'_1, s'_1) = sqz(e_1)$  and  $(e'_2, s'_2) = sqz(e_2)$ .

The non-squeezing transformation uses the auxiliary function  $\theta$  which takes two arguments, the current information as a temporary expression and the rest of the stream, and returns the next information. On the other hand, the squeezing transformation uses the auxiliary function  $\eta$  which takes three arguments adding one extra argument to those of  $\theta$ . The extra argument is used for output buffer which does not change during the computation except for adding some events to the tail of the original buffer. In the output buffer, the second element of the result of the squeeze function  $sqz$  is added as a possibly-known part at the head of the result. It is easy to show that

$$s'e' = e \quad \text{if} \quad (e', s') = sqz(e) \tag{2}$$

for  $e \in Tmp_S$  by induction on the structure of  $e$ .

The following lemma shows that the non-squeezing transformation and the squeezing transformation are equivalent. In the rest of the paper, we employ the non-squeezing transformation instead of the squeezing one to compare the behavior of a mft and an xsp since it is simpler than the squeezing transformation, although the implementation of XML stream processor does employ the squeezing transformation.

**Lemma 3.5** Let  $S = (Q, \Sigma, \Delta, in, R)$  be an xsp and  $s \in \Sigma_{\langle / \rangle}^*$ . Then we have

$$\theta_S(in(\varepsilon, \dots, \varepsilon), s) = \eta_S(in(\varepsilon, \dots, \varepsilon), s, \varepsilon) \tag{3}$$

where  $\theta_S$  and  $\eta_S$  are as given in DEFINITION 3.4.

PROOF. We prove the equation

$$\theta_S(be, s) = \eta_S(e, s, b) \tag{4}$$

for  $b \in \Delta_{\langle / \rangle}^*$  and  $e \in Tmp_S$ , which is more general than (3). Equation (3) is the special case of (4) in which  $b = \varepsilon$  and  $e = in(\varepsilon, \dots, \varepsilon)$ . We show at the same time

$$\eta_S(e, s, b) = \eta_S(be, s, \varepsilon) \tag{5}$$

for  $s, b \in \Delta_{\langle / \rangle}^*$  and  $e \in Tmp_S$ .

Equations (4) and (5) are proved by induction on the length  $\#s$  of  $s$ . If  $\#s = 0$ , then both (4) and (5) are the same, that is  $be$ .

If  $\#s > 0$ , then suppose that  $s = \chi s'$  with  $\chi \in \Sigma_{</>}$  and  $s' \in \Sigma_{</>}^*$  and that  $b = \xi_1 \dots \xi_n$  ( $n \geq 0$ ) with  $\xi_j \in \Delta_{</>}$  for  $j = 1, \dots, n$ . The left-hand side of (4) is

$$\begin{aligned} \theta_S(\xi_1 \dots \xi_n e, \chi s') &= \theta_S(\langle \xi_1 \dots \xi_n e, \chi \rangle, s') \\ &= \theta_S(\xi_1 \dots \xi_n \langle e, \chi \rangle, s') \\ &= \eta_S(\langle e, \chi \rangle, s', \xi_1 \dots \xi_n) \\ &= \eta_S(\xi_1 \dots \xi_n \langle e, \chi \rangle, s', \varepsilon) \end{aligned}$$

from the definitions of  $\theta$  and  $\langle \_ \rangle$  and the induction hypotheses of (4) and (5). When  $(e'', s'') = sqz(\langle e, \chi \rangle)$ , the right-hand side of (4) is

$$\begin{aligned} \eta_S(e, \chi s', \xi_1 \dots \xi_n) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s', \varepsilon) \end{aligned}$$

from the definitions of  $\eta_S$  and the induction hypothesis of (5). Both sides of (4) are the same since we have  $s'' e'' = \langle e, \chi \rangle$  by (2). Hence (4) holds.

From the definition of  $\eta_S$ , the induction hypothesis of (5) and  $(e'', s'') = sqz(\langle e, \chi \rangle)$ , the left-hand side of (5) is

$$\begin{aligned} \eta_S(e, \chi s', \xi_1 \dots \xi_n) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s', \varepsilon). \end{aligned}$$

Since we have  $sqz(\langle \xi_1 \dots \xi_n e, \chi \rangle) = sqz(\xi_1 \dots \xi_n \langle e, \chi \rangle) = (e'', \xi_1 \dots \xi_n s'')$  from the definitions of  $\langle \_ \rangle$  and  $sqz$ , the right-hand side of (5) is

$$\begin{aligned} \eta_S(\xi_1 \dots \xi_n e, \chi s', \varepsilon) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s', \varepsilon) \end{aligned}$$

from the definition of  $\eta_S$  and the induction hypothesis of (5). Hence (5) holds.  $\blacksquare$

### 3.3 Derivation of XML Stream Processors

The derivation of an xsp from a given mft is achieved in a similar way to the existing method by Engelfriet and Vogler [5] to synthesize two tree transducers, a top-down tree transducer (for short, tdt) and a macro tree transducer (for short, mtt). That is because an XML parser which transforms XML streams to forests (binary labeled trees) can be represented by an infinitary tdt and a mft is a simple extension of a mtt. Therefore we can give a derivation method of an xsp just as a straightforward extension of the existing method. Correctness of our method will be shown in the next subsection.

**Definition 3.6** Let  $M = (Q, \Sigma, \Delta, in, R)$  be a mft. We define an xsp  $\mathcal{SP}(M) = (Q', \Sigma, \Delta, in', R')$  where

- $Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\}$  where  $rank(q[i]) = rank(q) - 1$  for every  $q \in Q$  and  $i \in \mathbb{N}$ ,
- $in' = in[0] \in Q$ ,
- $R'$  contains the following rules:

– For every  $q \in Q$ ,  $\sigma \in \Sigma$  and  $(q(\sigma[x_1]x_2, y_1, \dots, y_n) \rightarrow rhs) \in R$ , the  $(q[0], \langle \sigma \rangle)$ -rule in  $R'$  is

$$q[0](y_1, \dots, y_n) \xrightarrow{\langle \sigma \rangle} \mathcal{A}(rhs),$$

– For every  $q \in Q$ ,  $\sigma \in \Sigma$  and  $(q(\%[\sigma]x, y_1, \dots, y_n) \rightarrow rhs) \in R$ , the  $(q[0], \sigma)$ -rule in  $R'$  is

$$q[0](y_1, \dots, y_n) \xrightarrow{\sigma} \mathcal{A}(rhs[x_1/x])$$

where  $rhs[x_1/x]$  is obtained by replacing  $x$  by  $x_1$  in  $rhs$ ,

- For every  $q \in Q$ ,  $\sigma \in \Sigma$ ,  $i \in \mathbb{N}$  and  $q(\cdot), y_1, \dots, y_n \rightarrow rhs \in R$ , the  $(q[0], \langle / \sigma \rangle)$ -rule and  $(q[i], \text{EOF})$ -rule in  $R'$  are

$$q[0](y_1, \dots, y_n) \xrightarrow{\langle / \sigma \rangle} \mathcal{A}(rhs), \quad q[i](y_1, \dots, y_n) \xrightarrow{\text{EOF}} \mathcal{A}(rhs),$$

respectively,

- For every  $q \in Q$ ,  $\sigma \in \Sigma$  and  $i \geq 1$ , the  $(q[i], \langle \sigma \rangle)$ -rule and  $(q[i], \langle / \sigma \rangle)$ -rule in  $R'$  are

$$q[i](y_1, \dots, y_n) \xrightarrow{\langle \sigma \rangle} q[i+1](y_1, \dots, y_n), \quad q[i](y_1, \dots, y_n) \xrightarrow{\langle / \sigma \rangle} q[i-1](y_1, \dots, y_n),$$

respectively,

where  $\mathcal{A}$  is defined over right-hand side expressions of rules in mft's as follows:

$$\begin{aligned} \mathcal{A}(q'(x_1, rhs_1, \dots, rhs_m)) &= q'[0](\mathcal{A}(rhs_1), \dots, \mathcal{A}(rhs_m)) \\ \mathcal{A}(q'(x_2, rhs_1, \dots, rhs_m)) &= q'[1](\mathcal{A}(rhs_1), \dots, \mathcal{A}(rhs_m)) \\ \mathcal{A}(y_j) &= y_j & \mathcal{A}(\cdot) &= \varepsilon \\ \mathcal{A}(\delta[rhs]) &= \langle \delta \rangle \mathcal{A}(rhs) \langle / \delta \rangle & \mathcal{A}(\%[\delta]) &= \delta & \mathcal{A}(rhs \ rhs') &= \mathcal{A}(rhs) \ \mathcal{A}(rhs') \end{aligned}$$

Now we show two examples of derivation of xsp's from a mft  $M_{htm}$  of EXAMPLE 2.4 and a mft  $M_{mir}$  of EXAMPLE 2.5. Additionally we illustrate how the obtained xsp  $\mathcal{SP}(M_{htm})$  works for a certain input XML stream. In these examples, we omit some of rules whose right hand side is  $\varepsilon$  they are derived from omitted rules whose right-hand side is a leaf in the original mft.

**Example 3.7** The derivation method gives an xsp  $\mathcal{SP}(M_{htm}) = (Q', \Sigma, \Delta, Main[0], R')$  from the mft  $M_{htm} = (Q, \Sigma, \Delta, Main, R)$  in EXAMPLE 2.4 where

$$Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\},$$

$$\begin{aligned} R' = \{ & Main[0]() \xrightarrow{\langle \text{article} \rangle} \langle \text{html} \rangle \langle \text{head} \rangle Title[0]() \langle / \text{head} \rangle \langle \text{body} \rangle InArticle[0](\varepsilon) \langle / \text{body} \rangle \langle / \text{html} \rangle, \\ & Title[0]() \xrightarrow{\langle \text{title} \rangle} \langle \text{title} \rangle Copy[0]() \langle / \text{title} \rangle, \\ & InArticle[0](y_1) \xrightarrow{\langle \text{title} \rangle} \langle \text{h1} \rangle Copy[0]() \langle / \text{h1} \rangle InArticle[1](y_1), \\ & InArticle[0](y_1) \xrightarrow{\langle \text{para} \rangle} \langle \text{p} \rangle Key2Em[0]() \langle / \text{p} \rangle InArticle[1](y_1 \ AllKeys[0]), \\ & InArticle[0](y_1) \xrightarrow{\langle \text{postscript} \rangle} \langle \text{h2} \rangle Index \langle / \text{h2} \rangle \langle \text{ul} \rangle y_1 \langle / \text{ul} \rangle \langle \text{h2} \rangle Postscript \langle / \text{h2} \rangle Copy[0](), \\ & Key2Em[0]() \xrightarrow{\langle \text{key} \rangle} \langle \text{em} \rangle Copy[0]() \langle / \text{em} \rangle Key2Em[1](), \quad Key2Em[0]() \xrightarrow{\sigma} \sigma Key2Em[1] \quad (\sigma \in \Sigma), \\ & AllKeys[0]() \xrightarrow{\langle \text{key} \rangle} \langle \text{li} \rangle Copy[0]() \langle / \text{li} \rangle AllKeys[1](), \quad AllKeys[0]() \xrightarrow{\sigma} \sigma AllKeys[1] \quad (\sigma \in \Sigma), \\ & Copy[0]() \xrightarrow{\langle \sigma \rangle} \langle \sigma \rangle Copy[0]() \langle / \sigma \rangle Copy[1] \quad (\sigma \in \Sigma), \quad Copy[0]() \xrightarrow{\sigma} \sigma Copy[0] \quad (\sigma \in \Sigma), \\ & q[i]() \xrightarrow{\langle \sigma \rangle} q[i+1]() \quad (\sigma \in \Sigma, i \geq 1, q \neq InArticle), \\ & q[i]() \xrightarrow{\sigma} q[i] \quad (\sigma \in \Sigma, i \geq 1, q \neq InArticle) \\ & q[i]() \xrightarrow{\langle / \sigma \rangle} q[i-1]() \quad (\sigma \in \Sigma, i \geq 1, q \neq InArticle) \\ & q[i]() \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_0, q \neq InArticle), \\ & InArticle[i](y_1) \xrightarrow{\langle \sigma \rangle} InArticle[i+1](y_1) \quad (\sigma \in \Sigma, i \geq 1), \\ & InArticle[i](y_1) \xrightarrow{\sigma} InArticle[i](y_1) \quad (\sigma \in \Sigma, i \geq 1), \\ & InArticle[i](y_1) \xrightarrow{\langle / \sigma \rangle} InArticle[i-1](y_1) \quad (\sigma \in \Sigma, i \geq 1), \\ & InArticle[i](y_1) \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_0) \}, \end{aligned}$$

where  $\Sigma_0 = \{(\langle / \sigma \rangle, 0) \mid \sigma \in \Sigma\} \cup \{(\text{EOF}, i) \mid i \in \mathbb{N}\}$ .

Let an input XML stream for  $\mathcal{SP}(M_{htm})$  be

`<article> <title> MFT </title> <para> XML is ...`



```

Main[0]()
 $\xrightarrow{\langle \text{article} \rangle}$  <html> <head> Title[0]() </head> <body> InArticle[0]( $\varepsilon$ ) </body> </html>
 $\xrightarrow{\langle \text{title} \rangle}$  <html> <head> <title> Copy[0]() </title> </head>
  <body> <h1> Copy[0]() </h1> InArticle[1]( $\varepsilon$ ) </body> </html>
 $\xrightarrow{\text{MFT}}$  <html> <head> <title> MFT Copy[0]() </title> </head>
  <body> <h1> MFT Copy[0]() </h1> InArticle[1]( $\varepsilon$ ) </body> </html>
 $\xrightarrow{\langle \text{title} \rangle}$  <html> <head> <title> MFT </title> </head> <body> <h1> MFT </h1> InArticle[0]( $\varepsilon$ ) </body> </html>
 $\xrightarrow{\langle \text{para} \rangle}$  <html> <head> <title> MFT </title> </head>
  <body> <h1> MFT </h1> <p> Key2Em[0]() </p> InArticle[1](AllKeys[0]) </body> </html>
 $\xrightarrow{\text{XML is}}$  <html> <head> <title> MFT </title> </head>
  <body> <h1> MFT </h1> <p> XML is Key2Em[0]() </p> InArticle[1](AllKeys[0]) </body> </html>
 $\Rightarrow \dots$ 

```

Figure 3: Stream processing induced by  $\mathcal{SP}(M_{htm})$

Then an xsp proceeds as shown in Figure 3 where  $\xrightarrow{\chi}$  stands for buffer updating when an input event  $\chi$  is read. The processing is as expected in Figure 2. For each step, the stream processor outputs the head-determined part by the squeeze function. The remainder is stored in a buffer.

**Example 3.8** *The derivation method gives an xsp  $\mathcal{SP}(M_{mir}) = (Q', \Sigma, \Delta, \text{Main}[0], R')$  from the mft  $M_{mir} = (Q, \Sigma, \Delta, \text{Main}, R)$  in EXAMPLE 2.5 where*

$$Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\},$$

$$\begin{aligned}
R' = \{ & \text{Main}[0]() \xrightarrow{\langle \text{rev} \rangle} \langle \text{rev} \rangle \text{Rev}[0](\varepsilon) \langle / \text{rev} \rangle \text{Main}[1](), \\
& \text{Main}[0]() \xrightarrow{\langle \sigma \rangle} \langle \sigma \rangle \text{Main}[0] \langle / \sigma \rangle \text{Main}[1] \quad (\sigma \neq \text{rev}), \quad \text{Main}[0]() \xrightarrow{\sigma} \sigma \text{Main}[0] \quad (\sigma \in \Sigma), \\
& \text{Main}[i]() \xrightarrow{\langle \sigma \rangle} \text{Main}[i+1]() \quad (\sigma \in \Sigma, i \geq 1), \quad \text{Main}[i]() \xrightarrow{\sigma} \text{Main}[i]() \quad (\sigma \in \Sigma, i \geq 1), \\
& \text{Main}[i]() \xrightarrow{\langle / \sigma \rangle} \text{Main}[i-1] \quad (\sigma \in \Sigma, i \geq 1) \quad \text{Main}[i]() \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_0), \\
& \text{Rev}[0](y_1) \xrightarrow{\langle \sigma \rangle} \text{Rev}[1](\langle \sigma \rangle \text{Rev}[0] \langle / \sigma \rangle y_1) \quad (\sigma \in \Sigma), \quad \text{Rev}[0](y_1) \xrightarrow{\sigma} \text{Rev}[0](\sigma y_1) \quad (\sigma \in \Sigma), \\
& \text{Rev}[i](y_1) \xrightarrow{\langle \sigma \rangle} \text{Rev}[i+1](y_1) \quad (\sigma \in \Sigma, i \geq 1), \quad \text{Rev}[i](y_1) \xrightarrow{\sigma} \text{Rev}[i](y_1) \quad (\sigma \in \Sigma, i \geq 1), \\
& \text{Rev}[i](y_1) \xrightarrow{\langle / \sigma \rangle} \text{Rev}[i-1](y_1) \quad (\sigma \in \Sigma, i \geq 1), \quad \text{Rev}[i](y_1) \xrightarrow{\chi} y_1 \quad ((\chi, i) \in \Sigma_0) \},
\end{aligned}$$

where  $\Sigma_0 = \{(\langle / \sigma \rangle, 0) \mid \sigma \in \Sigma\} \cup \{(\text{EOF}, i) \mid i \in \mathbb{N}\}$ .

### 3.4 Correctness of derivation

The correctness of our derivation of XML stream processors is that, for every mft and every input forest, the XML stream corresponding to the transformation result of the mft for the forest is equal to the transformation result of the xsp obtained by our derivation from the mft. In the rest of section we do not deal with text nodes in forests. The proof can be easily extended for text nodes. Additionally we ignore the names of end tags since we deal with only well-formed XML. We write  $\langle / \square \rangle$  to denote a proper end tag. The name of the end tag can be recovered from the context.

Correctness is stated by the following theorem.

**Theorem 3.9** *Let  $M = (Q, \Sigma, \Delta, in, R)$  be a mft. Then*

$$\tau_{\mathcal{SP}(M)}(\llbracket f \rrbracket) = \llbracket \tau_M(f) \rrbracket$$

for every  $f \in \mathcal{F}_\Sigma$ .

To prove this theorem, we show several lemmas with respect to properties of an extension of  $\theta_S$ . Before the definition of the extension, we introduce the *following forests representation* (for short, FFR) for a forest  $f$  that is a list of sub-forests of  $f$  whose syntax is

$$L ::= [] \mid f' :: L$$

where  $f'$  is a sub-forest of  $f$ . We use  $FF_f$  to denote a set of FFR's for a forest  $f$ . The  $i$ -th element of  $L \in FF_f$  is accessed by  $L.i$  where  $(f' :: L).0 = f'$  and  $(f' :: L).i = L.(i - 1)$ . The FFR can output one by one the next XML event from the XML stream corresponding to  $f$  by updating the representation. The initial FFR for a forest  $f$  is a singleton list of  $f$ , i.e.,  $f :: []$ . The updating function  $ud$  takes the current FFR and returns the next XML event and the next FFR as follows:

$$ud(\sigma[f_1]f_2 :: L) = \langle \sigma \rangle, f_1 :: f_2 :: L \quad ud(\langle \rangle :: f :: L) = \langle / \square \rangle, f :: L \quad ud(\langle \rangle :: []) = (\text{EOF}, []).$$

It outputs one by one the next XML event by updating the FFR by  $ud$ , which is confirmed by the following lemma. We do not define  $ud([])$  because the computation is not required in the rest of the paper.

**Lemma 3.10** *Let  $g$  be the function defined over FFR's by*

$$g(l) = \begin{cases} \varepsilon & \text{if } l = [] \\ \chi g(l') & \text{otherwise} \end{cases}$$

where  $(\chi, l') = ud(l)$ . Then we have

$$g(f :: []) = \lfloor f \rfloor \text{EOF}. \quad (6)$$

PROOF. First we show the following equation

$$g(f :: f' :: L) = \lfloor f \rfloor \langle / \square \rangle g(f' :: L) \quad (7)$$

for forests  $f', f$  and a FFR  $l$  with some  $\sigma'$  by induction on the structure of  $f$ . If  $f = \langle \rangle$ , then (7) holds by the definitions of  $g$ ,  $ud$  and  $\lfloor \_ \rfloor$ . If  $f = \sigma[f_1]f_2$ , then we have

$$\begin{aligned} g(\sigma[f_1]f_2 :: f' :: L) &= \langle \sigma \rangle g(f_1 :: f_2 :: f' :: L) \\ &= \langle \sigma \rangle \lfloor f_1 \rfloor \langle / \square \rangle g(f_2 :: f' :: L) \\ &= \langle \sigma \rangle \lfloor f_1 \rfloor \langle / \square \rangle \lfloor f_2 \rfloor \langle / \square \rangle g(f' :: L) \\ &= \lfloor \sigma[f_1]f_2 \rfloor \langle / \square \rangle g(f' :: L) \end{aligned}$$

from the definition of  $g$ ,  $ud$  and  $\lfloor \_ \rfloor$  and the induction hypothesis. Hence (7) holds for every forest  $f$ .

Now we prove the equation (6) by induction on the structure of  $f$ . If  $f = \langle \rangle$ , then (6) holds by the definitions of  $g$ ,  $ud$  and  $\lfloor \_ \rfloor$ . If  $f = \sigma[f_1]f_2$ , then we have

$$\begin{aligned} g(\sigma[f_1]f_2 :: []) &= \langle \sigma \rangle g(f_1 :: f_2 :: []) \\ &= \langle \sigma \rangle \lfloor f_1 \rfloor \langle / \square \rangle g(f_2 :: []) \\ &= \langle \sigma \rangle \lfloor f_1 \rfloor \langle / \square \rangle \lfloor f_2 \rfloor \text{EOF} \\ &= \lfloor \sigma[f_1]f_2 \rfloor \text{EOF} \end{aligned}$$

from the definition  $g$ ,  $ud$  and  $\lfloor \_ \rfloor$ , the equation (7) and the induction hypothesis. Therefore (6) holds for every forest  $f$ . ■

Now we define an extension of  $\theta_S$  with FFR. Let  $S$  be an xsp and  $f$  be a forest such that  $\lfloor f \rfloor$  is an input stream for  $S$ . We define the function  $\Theta_S$  as well as  $\theta_S$  by

$$\Theta_S(E, []) = E \quad \Theta_S(E, L) = \Theta(\langle E, \chi \rangle, L')$$

for  $E \in \text{Tmp}_S$  and  $L \in FF_f$  where  $(\chi, L') = ud(L)$ . The following lemma shows that the function  $\Theta_S$  can simulate  $\tau_S$  for an input stream  $\lfloor f \rfloor$ .

**Lemma 3.11** *Let  $S = (Q, \Sigma, \Delta, in, R)$  be an xsp. Then*

$$\tau_S(\lfloor f \rfloor) = \Theta_S(in(\varepsilon, \dots, \varepsilon), f :: []) \quad (8)$$

for every  $f \in \mathcal{F}_\Sigma$ .

PROOF. From the definition of  $\tau_S$ , what we have to show is

$$\theta_S(\text{in}(\varepsilon, \dots, \varepsilon), \lfloor f \rfloor \text{ EOF}) = \Theta_S(\text{in}(\varepsilon, \dots, \varepsilon), f :: []). \quad (9)$$

Let  $g$  be the function as given in LEMMA 3.10. and  $G(L)$  be a set of FFR's occurring as an argument of  $g$  in the computation of  $g(L)$ , i.e., if  $L = []$  then  $G(L) = \{L\}$  and otherwise  $G(L) = \{L\} \cup G(L')$  with  $(\chi, L') = \text{ud}(L)$ . The set  $G(f :: [])$  is finite because  $g(f :: [])$  always terminates as shown in the proof of LEMMA 3.10. We show the more general equation

$$\theta_S(E, g(L)) = \Theta_S(E, L) \quad (10)$$

for  $E \in \text{Tmp}_S$  and  $L \in G(f :: [])$ . From LEMMA 3.10 we can claim that the equation (9) is the special case of (10) in which  $E = \text{in}(\varepsilon, \dots, \varepsilon)$  and  $L = f :: []$ . We prove the equation (10) by induction on the cardinality  $\sharp G(L)$  of  $G(L)$ . If  $\sharp G(L) = 1$ , i.e.,  $L = []$ , then the both sides are the same, that is  $e$ . If  $\sharp G(L) > 1$ , then  $g(L) = \chi g(L')$  with  $(\chi, L') = \text{ud}(L)$ . We have  $\sharp G(L') = \sharp G(L) - 1$  since  $L \notin G(L')$  from the definition of  $G$ . Therefore we obtain

$$\begin{aligned} \theta_S(E, g(L)) &= \theta_S(E, \chi g(L')) \\ &= \theta_S(\langle \! \langle E, \chi \! \rangle \! \rangle, g(L')) \\ &= \Theta_S(\langle \! \langle E, \chi \! \rangle \! \rangle, L') \\ &= \Theta_S(E, L) \end{aligned}$$

from the induction hypothesis, the definition of  $\Theta_S$  and  $(\chi, L') = \text{ud}(L)$ . Hence the equation (10) holds for  $e \in \text{Tmp}_S$  and  $L \in G(f :: [])$ . ■

Next we define the function  $\mathcal{I}$  that translates temporary expressions into output forests. The function  $\mathcal{I}$  has a good property that  $\mathcal{I}(E, L) = \mathcal{I}(E', L')$  if  $\Theta_S(E, L)$  is computed by  $\Theta_S(E', L')$ , which will be shown as LEMMA 3.12.

$$\begin{aligned} \mathcal{I}(q[i](E_1, \dots, E_n), L) &= \llbracket q \rrbracket(L.i, \mathcal{I}(E_1, L), \dots, \mathcal{I}(E_n, L)) & \mathcal{I}(\varepsilon, L) &= () \\ \mathcal{I}(\langle \! \langle \delta \! \rangle \! \rangle E \langle \! \langle \square \! \rangle \! \rangle, L) &= \delta[\mathcal{I}(E, L)] & \mathcal{I}(E \ E', L) &= \mathcal{I}(E, L) \ \mathcal{I}(E', L) \end{aligned}$$

**Lemma 3.12** *Let  $M = (Q, \Sigma, \Delta, \text{in}, R)$  be a mft,  $\mathcal{SP}(M) = (Q, \Sigma, \Delta, \text{in}, R')$  be an xsp,  $f \in \mathcal{F}_\Sigma$  be a forest and  $L \in G(f :: [])$  be a FFR where  $G$  is a function as given in the proof of LEMMA 3.11. Then*

$$\mathcal{I}(E, L) = \mathcal{I}(\langle \! \langle E, \chi \! \rangle \! \rangle, L') \quad (11)$$

where  $(\chi, L') = \text{ud}(L)$ .

PROOF. We prove the statements by induction on the structure of  $E$ . Here we show only the case of  $E = q[0](E_1, \dots, E_n)$  with  $q \in Q$  that is the most complicated one in the induction. The other cases can be shown in a similar way, which are omitted.

If  $E = q[0](E_1, \dots, E_n)$  with  $q \in Q$ , then the left-hand side of (11) is equal to  $\llbracket q \rrbracket(L.0, \mathcal{I}(E_1), \dots, \mathcal{I}(E_n))$  by the definition of  $\mathcal{I}$ . When  $L = \sigma[f_1]f_2 :: L'$ ,  $\text{ud}(L) = (\langle \! \langle \sigma \! \rangle \! \rangle, L')$  with  $L' = f_1 :: f_2 :: L$ . Then the left-hand side of (11) is

$$\begin{aligned} \llbracket q \rrbracket(L.0, \mathcal{I}(E_1, L), \dots, \mathcal{I}(E_n, L)) &= \llbracket q \rrbracket(\sigma[f_1]f_2, \mathcal{I}(E_1, L), \dots, \mathcal{I}(E_n, L)) \\ &= \llbracket q \rrbracket(\sigma[f_1]f_2, \mathcal{I}(\langle \! \langle E_1, \chi \! \rangle \! \rangle, L'), \dots, \mathcal{I}(\langle \! \langle E_n, \chi \! \rangle \! \rangle, L')) \\ &= \llbracket rhs^{q, \sigma} \rrbracket_\rho \end{aligned}$$

from the definition of  $\llbracket \_ \rrbracket$  and  $\text{ud}$  and the induction hypothesis for (11), where  $\rho(x_i) = f_i$  for  $i = 1, 2$  and  $\rho(y_j) = \mathcal{I}(\langle \! \langle E_j, \chi \! \rangle \! \rangle, L')$  for  $j = 1, \dots, n$ . The right-hand side of (11) is

$$\mathcal{I}(\langle \! \langle q[0](E_1, \dots, E_n), \langle \! \langle \sigma \! \rangle \! \rangle \! \rangle, L') = \mathcal{I}(\llbracket \mathcal{A}(rhs^{q, \sigma}) \rrbracket_{\rho'}, L')$$

where  $\rho'(y_j) = \langle \! \langle E_j, \langle \! \langle \sigma \! \rangle \! \rangle \! \rangle$  for  $j = 1, \dots, n$ . It is shown by induction on the structure of  $rhs^{q, \sigma}$  that

$$\llbracket rhs^{q, \sigma} \rrbracket_\rho = \mathcal{I}(\llbracket \mathcal{A}(rhs^{q, \sigma}) \rrbracket_{\rho'}, L')$$

using  $L'.0 = f_1$ ,  $L'.1 = f_2$  and the definition of  $\mathcal{I}$  and  $\mathcal{A}$ . When  $L = () :: f :: L''$  and  $L = () :: []$ , we can show the equation (11) in a way similar to the case of  $L = \sigma[f_1]f_2 :: L''$ . ■

Now we prove THEOREM 3.9. Let  $M$  be a mft and  $S = \mathcal{SP}(M)$  be an xsp. The statement of LEMMA 3.12 shows that, for the computation of

$$\Theta_S(E_0, L_0) = \Theta_S(E_1, L_1) = \dots = \Theta_S(E_n, L_n) = E_n \quad (12)$$

with  $L_n = []$ , all  $\mathcal{I}(E_k, L_k)$  with  $k = 0, \dots, n$  are the same. Since  $E_n = \langle E_{n-1}, \mathbf{EOF} \rangle$  and the right-hand side of every rule in  $S$  with respect to  $\mathbf{EOF}$  contains no occurrence of  $q(\dots)$  with a state  $q$ ,  $E_n$  is just an XML stream. Therefore  $[\mathcal{I}(E_n, L_n)] = E_n$  holds by the definition of  $\mathcal{I}$  and  $[\_]$ . From the relation of  $\mathcal{I}$  and  $\Theta$  shown in LEMMA 3.12 and the equation (12), we obtain

$$[\mathcal{I}(E_0, L_0)] = E_n = \Theta_S(E_0, L_0)$$

When  $E_0 = in[0](\varepsilon, \dots, \varepsilon)$  and  $L_0 = f :: []$  with an input forest  $f$ ,

$$\begin{aligned} [\tau_M(f)] &= [[in](f, (), \dots, ())] = [\mathcal{I}(E_0, L_0)] \\ \tau_S([f]) &= \Theta_S(E_0, L_0). \end{aligned}$$

Therefore THEOREM 3.9 has been proved.

## 4 Discussion

We have shown how to derive an xsp from an arbitrary mft. Thus whether existing languages can be implemented as a program in stream processing style is whether the language can be translated into an mft. In order to make the translation easy, we discuss the extension of mft. In the formalization of mft, we cannot use even primitive functions over booleans, integers, strings, etc. In this section, we discuss how to extend our framework for such additional features and a few idea of translation for existing languages. Additionally, we add limitations of XML stream processors and show a benchmark result comparing with the existing processor.

### 4.1 Booleans and Conditional Branches

We consider a simple extension of mft with booleans and their operator, conditional branches. Let us extend the right-hand expression of mft with them as follows.

$$rhs ::= \dots \mid true \mid false \mid if(rhs, rhs, rhs)$$

where *true* and *false* are boolean values and *if*( $e_1, e_2, e_3$ ) stands for a conditional branch with a test  $e_1$ , a true-branch  $e_2$  and a false-branch  $e_3$ .

If we regard *true*, *false* and *if* as output symbols of the mft, our algorithm derives an xsp from the mft though these symbols are left in right-hand side of rules in the obtained xsp. Hence we add the following special rules for them in a similar way to [15]:

$$if(true, e_1, e_2) \rightarrow e_1 \qquad if(false, e_1, e_2) \rightarrow e_2$$

By applying these rules in each squeezing phase, we achieve an XML stream processing for the extended mft.

### 4.2 Pattern-based Languages

Most of existing XML transformation languages support pattern-based recursion (iteration). For instance, XSLT [22] and XQuery [20] are based on pattern matching by XPath expressions. It is easy to encode simple forward XPath expressions in mft style. Predicates in an XPath expression can be encoded into mft-style programs using boolean values in the extended mft.

On the other hand, XDuce [8] and CDuce [2] are base on pattern matching by regular expressions. Though a regular pattern may contain the simbol  $*$  for Kleene-closure, both languages use the definition

$$\text{type } T = () \mid E T$$

for a regular expression type  $E*$ . Hence a program in these languages are written in recursive style which is quite similar to a mft-style program.

### 4.3 Limitation

There is a class of *inherently memory inefficient* transformations [16] such as  $M_{mir}$  in EXAMPLE 2.5, which reverses the order of markups at every nesting level for all descendants of **rev** nodes. Suppose the root node of an input XML is labeled with **rev**. Though our framework can deal with such a transformation, the obtained XML stream processor is not efficient because it cannot output any result until reading the end of the input stream. This problem is not specific to our framework. Every SAX-like stream processing program has the same problem: this kind of transformation is not suitable for stream processing.

## 5 Conclusion

We have presented a method to automatically derive an XML stream processor from a program in functional XML processing style, where we write XML transformations as recursive functions over the input XML tree. We adopt macro forest transducers (mft) as a model of functional XML processing and have shown that we can obtain an XML stream processor for every mft by our method. The framework presented in this paper will be applied to the next release of XTISP [13]. The extension of our method will be applied to existing languages [8, 2, 22] in which programs are given a set of recursive functions over XML trees (forests).

## Acknowledgment

The author is grateful to Giuseppe Castagna and Shin-Cheng Mu for their kind help and advice on the manuscript.

## References

- [1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. *International Journal on Very Large Data Bases*, pages 53–64, 2000.
- [2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th International Conference of Functional Programming*, pages 51–63, 2003.
- [3] P. Cimprich, O. Becker, C. Nentwich, M. K. H. Jiroušek, P. Brown, M. Batsis, T. Kaiser, P. Hlavnička, N. Matsakis, C. Dolph, and N. Wiechmann. Streaming transformations for XML (STX) version 1.0. <http://stx.sourceforge.net/documents/>.
- [4] Y. Diao and M. J. Franklin. High-performance XML filtering: An overview of YFilter. In *IEEE Data Engineering Bulletin*, volume 26(1), pages 41–48, 2003.
- [5] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.
- [6] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.
- [7] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2003.
- [8] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
- [9] O. Kiselyov. A better XML parser through functional programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 209–224, 2002.
- [10] K. Kodama, K. Suenaga, N. Kobayashi, and A. Yonezawa. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In *The 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 41–56, 2004.

- [11] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 227–238, 2002.
- [12] M. Murata. Extended path expressions of XML. In *Proceedings of the 20th ACM Symp. on Principles of Database Systems*, pages 153–166, 2001.
- [13] K. Nakano. XTISP: XML transformation language intended for stream processing. <http://xtisp.psdlab.org/>.
- [14] K. Nakano. Composing stack-attributed transducers. Technical Report METR-2004-01, Department of Mathematical Informatics, University of Tokyo, 2004.
- [15] K. Nakano. An implementation scheme for XML transformation languages through derivation of stream processors. In *The 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 74–90, 2004.
- [16] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54:257–290, 2005.
- [17] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89:141–149, 2004.
- [18] S. Scherzinger and A. Kemper. Syntax-directed transformations of XML streams. In *The workshop on Programming Language Technologies for XML*, pages 75–86, 2005.
- [19] K. Suenaga, N. Kobayashi, and A. Yonezawa. Extension of type-based approach to generation of stream processing programs by automatic insertion of buffering primitives. In *International workshop on Logic-based Program Synthesis and Transformation*, 2005. To appear.
- [20] XQuery 1.0: An XML query language. <http://www.w3.org/TR/xquery/>.
- [21] SAX: the simple api for XML. <http://www.saxproject.org/>.
- [22] XSL transformations (XSLT). <http://www.w3c.org/TR/xslt/>.

# Parametric Polymorphism for XML

Haruo Hosoya

The University of Tokyo

`hahosoya@is.s.u-tokyo.ac.jp`

Alain Frisch

INRIA

`Alain.Frisch@inria.fr`

Giuseppe Castagna

École Normale Supérieure de Paris

`Giuseppe.Castagna@ens.fr`

## Abstract

Despite the extensiveness of recent investigations on static typing for XML, parametric polymorphism has rarely been treated. This well-established typing discipline can also be useful in XML processing in particular for programs involving “parametric schemas,” i.e., schemas parameterized over other schemas (e.g., SOAP). The difficulty in treating polymorphism for XML lies in how to extend the “semantic” approach used in the mainstream (monomorphic) XML type systems. A naive extension would be “semantic” quantification over all substitutions for type variables. However, this approach reduces to an NEXPTIME-complete problem for which no practical algorithm is known. In this paper, we propose a different method that smoothly extends the semantic approach yet is algorithmically easier. In this, we devise a novel and simple *marking* technique, where we interpret a polymorphic type as a set of values with annotations of which subparts are parameterized. We exploit this interpretation in every ingredient of our polymorphic type system such as subtyping, inference of type arguments, and so on. As a result, we achieve a sensible system that directly represents a usual expected behavior of polymorphic type systems—“values of variable types are never reconstructed”—in a reminiscence of Reynold’s parametricity theory. Also, we obtain a set of practical algorithms for typechecking by local modifications to existing ones for a monomorphic system.

This paper has been presented at ACM Symposium on Principles of Programming Languages (POPL’05).

# Compositional Specification of Commercial Contracts

Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen

Department of Computer Science, University of Copenhagen (DIKU)  
Universitetsparken 1, DK-2100 Copenhagen Ø  
Denmark

**Abstract.** We present a declarative language for compositional specification of contracts governing the exchange of resources. It extends Eber and Peyton Jones’s declarative language for specifying financial contracts to the exchange of money, goods and services amongst multiple parties and complements McCarthy’s Resources/Events/Agents (REA) accounting model with a view-independent formal contract model that supports definition of user-defined contracts, automatic monitoring under execution, and user-definable analysis of their state before, during and after execution. We provide several realistic examples of commercial contracts and their analysis. A variety of (real) contracts can be expressed in such a fashion as to support their integration, management and analysis in an operational environment that registers events.

## 1 Introduction

When entrepreneurs enter contractual relationships with a large number of other parties, each with possible variations on standard contracts, they are confronted with the interconnected problems of *specifying* contracts, *monitoring* their execution for performance<sup>1</sup>, *analyzing* their ramifications for planning, pricing and other purposes prior to and during execution, and *integrating* this information with accounting, workflow management, supply chain management, production planning, tax reporting, decision support *etc.*

### 1.1 Problems with Informal Contract Management

Typical problems that can arise in connection with informal modeling and representation of contracts and their execution include: (i) disagreement on what a contract actually requires; (ii) agreement on contract, but disagreement on what events have actually happened (event history); (iii) agreement on contract and event history, but disagreement on remaining contractual obligations; (iv) breach or malexecution of contract; (v) entering bad or undesirable contracts/missed opportunities; (vi) bad coordination of contractual obligations with production planning and supply chain management; (vii) impossibility, slowness or costliness in evaluating state of company affairs.

Anecdotal evidence suggests that costs associated with these problems can be considerable. Eber estimates that a major French investment bank has costs of about 50 mio. Euro per year attributable to (i) and (iv) above, with about half due to legal costs in connection with contract disputes and the other half due to malexecution of financial contracts [Ebe02].

In summary, capturing contractual obligations precisely and managing them conscientiously is important for a company’s planning, evaluation, and reporting to management, shareholders, tax authorities, regulatory bodies, potential buyers, and others.

We argue that a declarative *domain-specific (specification) language (DSL)* for compositional specification of commercial contracts (defining contracts by combining subcontracts in various, well-defined ways) with an associated precise *operational semantics* is ideally suited to alleviating the above problems.

<sup>1</sup> *Performance* in contract lingo refers to *compliance* with the *promises* (contractual commitments) stipulated in a contract; nonperformance is also termed *breach of contract*.



## 1.2 Contributions

We (i) extend the contract language of Peyton-Jones, Eber and Seward for two-party financial contracts in a view-independent fashion to multi-party commercial contracts with iteration and first-order recursion. They involve explicit agents and transfers of arbitrary resources (money, goods and services, or even pieces of information), not only currencies. Our contract language is stratified into a pluggable base language for atomic contracts (commitments) and a combinator language for composing commitments into structured contracts. In addition, we (ii) provide a natural contract semantics based on an inductive definition for when a trace—a finite sequence of events—constitutes a successful (“performing”) completion of a contract. This induces a denotational semantics, which compositionally maps contracts to trace sets as in Hoare’s Communicating Sequential Processes (CSP). We (iii) systematically develop three operational semantics in a stepwise fashion, starting from the denotational semantics: A reduction semantics with deferred matching of events to specific commitments in a contract; an eager matching semantics in which events are matched nondeterministically against commitments; and finally an eager matching semantics where an event is equipped with explicit control information that *routes* it deterministically to a particular commitment. Finally, we (iv) validate applicability of our language by encoding a variety of existing contracts in it, and illustrate analyzability of contracts by providing examples of compositional analysis.

Our work builds on a previous language design by Andersen and Elsborg [AE03] and is inspired by Peyton Jones and Eber’s compositional specification of financial contracts, the REA accounting model and CSP-like process algebras. See Section 7 for a comparison with that work.

## 2 Modeling Commercial Contracts

A *contract* is an agreement between two or more parties which creates obligations to do or not do the specific things that are the subject of that agreement. A *commercial contract* is a contract whose subject is the exchange of scarce *resources* (money, goods, and services). Examples of commercial contracts are sales orders, service agreements, and rental agreements. Adopting terminology from the REA accounting model [McC82] we shall also call obligations *commitments* and parties *agents*.

### 2.1 Contract Patterns

In its simplest form a contract commits two contract parties to an exchange of resources such as goods for money or services for money; that is to a pair of *transfers* of resources from one party to the other, where one transfer is in *consideration* of the other.

The sales order *template* in Figure 1 commits the two parties (*seller*, *buyer*) to a pair of transfers, of goods from *seller* to *buyer* and of money from *buyer* to *seller*. Many commercial contracts are of this simple *quid-pro-quo* kind, but far from all. Consider the legal services agreement template in Figure 2. Here commitments for rendering of a monthly legal service are *repeated*, and each monthly service consists of a standard service part and an *optional* service part. More generally, a contract may allow for *alternative* executions, any one of which satisfies the given contract.

We can discern the following basic *contract patterns* for composing commercial contracts from subcontracts (a subcontract is a contract used as part of another contract):

- a *commitment* stipulates the transfer of a resource or set of resources between two parties; it constitutes an *atomic contract*;
- a contract may require *sequential* execution of subcontracts;
- a contract may require *concurrent* execution of subcontracts, that is execution of all subcontracts, where individual commitments may be interleaved in arbitrary order;

- a contract may require execution of one of a number of *alternative* subcontracts;
- a contract may require *repeated* execution of a subcontract.

In the remainder of this paper we shall explore a declarative contract specification language based on these contract patterns.

---

**Fig. 1 Agreement to Sell Goods**

**Section 1.** (Sale of goods) Seller shall sell and deliver to buyer (description of goods) no later than (date).

**Section 2.** (Consideration) In consideration hereof, buyer shall pay (amount in dollars) in cash on delivery at the place where the goods are received by buyer.

**Section 3.** (Right of inspection) Buyer shall have the right to inspect the goods on arrival and, within (days) business days after delivery, buyer must give notice (detailed-claim) to seller of any claim for damages on goods.

---



---

**Fig. 2 Agreement to Provide Legal Services**

**Section 1.** The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.

**Section 2.** In consideration hereof, the company shall pay a monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receipt of an invoice.

**Section 3.** This contract is valid 1/1-12/31, 2004.

---

### 3 Compositional Contract Language

In this section we present a core contract specification language that reflects the contract composition patterns of Section 2.1. This is a cursory presentation, with no proofs given. See the technical report [AEH<sup>+</sup>04] for a full presentation.

#### 3.1 Syntax

Our contract language  $\mathcal{C}^P$  is defined inductively by the inference system for deriving judgements of the forms  $\Gamma; \Delta \vdash c : \text{Contract}$  and  $\Delta \vdash D : \Gamma$ . Here  $\Gamma$  and  $\Delta$  range over maps from identifiers to *contract template types* and to *base types*, respectively. The  $\oplus$ -operator on maps is defined as follows:

$$(m \oplus m')(x) = \begin{cases} m'(x) & \text{if } x \in \text{domain}(m') \\ m(x) & \text{otherwise} \end{cases}$$

The language is built on top of a typed *base language*  $P$  defined by  $\Delta \vdash a : \tau$  that defines expressions denoting *agents*, *resources*, *time*, other basic types and predicates (Boolean expressions) over those.  $P$  provides the possibility of referring to *observables* [JES00,JE03]. The language is parametric in  $P$ , and we shall introduce suitable base language expressions on an *ad hoc* basis in our examples for illustrative purposes.

The language  $\mathcal{C}^P$  is defined by the inference system in Figure 3. If judgement  $\Gamma; \Delta \vdash c : \text{Contract}$  is derivable, we say that  $c$  is a well-defined contract given type assumptions  $\Gamma$  and  $\Delta$ . Success denotes the *trivial* or (*successfully*) *completed* contract: it carries no obligations on

**Fig. 3** Syntax for contract specifications

$$\begin{array}{c}
\Gamma; \Delta \vdash \text{Success} : \text{Contract} \quad \Gamma; \Delta \vdash \text{Failure} : \text{Contract} \\
\\
\frac{\Gamma(f) = \tau \rightarrow \text{Contract} \quad \Delta \vdash a : \tau}{\Gamma; \Delta \vdash f(a) : \text{Contract}} \quad \frac{\Delta' = \Delta \oplus \{A_1 : \text{Agent}, A_2 : \text{Agent}, R : \text{Resource}, T : \text{Time}\} \quad \Gamma; \Delta' \vdash c : \text{Contract} \quad \Delta' \vdash P : \text{Boolean}}{\Gamma; \Delta \vdash \text{transmit}(A_1, A_2, R, T \mid P). c : \text{Contract}} \\
\\
\frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1 + c_2 : \text{Contract}} \quad \frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1 \parallel c_2 : \text{Contract}} \\
\\
\frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1; c_2 : \text{Contract}} \quad \frac{\Gamma = \{f_i \mapsto \tau_{i1} \times \dots \times \tau_{in_i} \rightarrow \text{Contract}\}_{i=1}^m \quad \Gamma; \Delta \oplus \{X_{i1} : \tau_{i1}, \dots, X_{in_i} : \tau_{in_i}\} \vdash c_i : \text{Contract}}{\Delta \vdash \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m : \Gamma} \\
\\
\frac{\Delta \vdash \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m : \Gamma \quad \Gamma; \Delta \vdash c : \text{Contract}}{\Delta \vdash \text{letrec } \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m \text{ in } c : \text{Contract}}
\end{array}$$

anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. The environment  $D = \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m$  contains named *contract templates*. A contract template needs to be instantiated with actual arguments from the base language. The contract expression  $\text{transmit}(A_1, A_2, R, T \mid P). c$  represents a contract where the *commitment*  $\text{transmit}(A_1, A_2, R, T \mid P)$  must be satisfied first. Note that  $A_1, A_2, R, T$  are binding variable occurrences whose scope is  $P$  and  $c$ . The commitment must be *matched* by a (*transfer*) event  $e = \text{transmit}(a_1, a_2, r, t)$  of resource  $r$  from agent  $a_1$  to agent  $a_2$  at time  $t$  where  $P(a_1, a_2, r, t)$  holds. After matching, the residual contract is  $c$  in which  $A_1, A_2, R, T$  are bound to  $a_1, a_2, r, t$ , respectively. In this fashion, the subsequent contractual obligations expressed by  $c$  may depend on the actual values in event  $e$ . The *contract combinators*  $\cdot + \cdot$ ,  $\cdot \parallel \cdot$  and  $\cdot ; \cdot$  compose subcontracts according to the contract patterns we have discerned: by alternation, concurrently, and sequentially, respectively. A contract consists of a finite set of named contract templates and a contract body. Note that contract templates may be (mutually) recursive, which, in particular, lets us capture repetition of subcontracts. In the following we shall adopt the convention that  $A_1, A_2, R, T$  must not be bound in environment  $\Delta$ . If a variable from  $\Delta$  or any expression  $a$  only involving variables bound in  $\Delta$  occurs as an argument of a transmit, we interpret this as an abbreviation; e.g.,  $\text{transmit}((a, A_2, R, T \mid P). c)$  abbreviates  $\text{transmit}((A_1, A_2, R, T \mid P \wedge A_1 = a). c)$  where  $A_1$  is a new (agent-typed) variable not bound in  $\Delta$  and different from  $A_2, R$  and  $T$ . We abbreviate  $\text{transmit}(A_1, A_2, R, T \mid P)$ . Success to  $\text{transmit}(A_1, A_2, R, T \mid P)$ . Examples encoding the contracts from Figures 1 and 2 are presented in Section 4.

### 3.2 Event Traces and Contract Satisfaction

A contract specifies a set of alternative performing event sequences (contract executions), each of which satisfies the obligations expressed in the contract and concludes it. In this section we make these notions precise for our language.

A *base structure* is a tuple  $(\mathcal{R}, \mathcal{T}, \mathcal{A})$  of sets of resources  $\mathcal{R}$ , agents  $\mathcal{A}$  and a totally ordered set  $(\mathcal{T}, \leq_{\mathcal{T}})$  of *dates* (or *time points*), plus other sets for other types, as needed. A (*transfer*) event  $e$  is a term  $\text{transmit}(a_1, a_2, r, t)$ , where  $a_1, a_2 \in \mathcal{A}, r \in \mathcal{R}$  and  $t \in \mathcal{T}$ . An (*event*) trace  $s$  is a finite sequence of events that is chronologically ordered; that is, for  $s = e_1 \dots e_n$  the time points in  $e_1 \dots e_n$  occur in ascending order. We adopt the following notation:  $\langle \rangle$  denotes the empty sequence; a trace consisting of a single event  $e$  is denoted by  $e$  itself; concatenation of traces

$s_1$  and  $s_2$  is denoted by juxtaposition:  $s_1 s_2$ ; we write  $(s_1, s_2) \rightsquigarrow s$  if  $s$  is an interleaving of the events in traces  $s_1$  and  $s_2$ ; we write  $\mathbf{X}$  for the vector  $X_1, \dots, X_k$  with  $k \geq 0$  and where  $k$  can be deduced from the context; we write  $P[a_1/A_1, a_2/A_2, r/R, t/T]$  and  $c[a_1/A_1, a_2/A_2, r/R, t/T]$  for *substitution* of expressions  $a_1, a_2, r, t$  for free variables  $A_1, A_2, R, T$  in Boolean expression  $P$  and contract expression  $c$ , respectively.<sup>2</sup> We are now ready to specify when a trace *satisfies* a contract, i.e. gives rise to a performing execution of the contract. This is done inductively by the inference system for judgements  $s \vdash_D^\delta c$  in Figure 4, where  $D = \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m$  is a finite set of named *contract templates* and  $\delta$  is a finite set of bindings of variables to elements of the given base structure. A derivable judgement  $s \vdash_D^\delta c$  expresses that event sequence  $s$  satisfies—successfully executes and concludes—contract  $c$  in an environment where contract templates are defined as in  $D$  and  $\delta$  specifies to which values the base variables in  $c$  and  $D$  are bound. Conversely, if  $s \vdash_D^\delta c$  is not derivable then  $s$  does not satisfy  $c$ . The premise  $\delta \models P[a_1/A_1, a_2/A_2, r/R, t/T]$  in the 3d rule stipulates that  $P[a_1/A_1, a_2/A_2, r/R, t/T]$ , with free variables bound as in  $\delta$ , must be true for an event to match the corresponding commitment.

**Fig. 4** Contract satisfaction

$$\begin{array}{c}
\langle \rangle \vdash_D^\delta \text{Success} \quad \frac{s \vdash_D^\delta c[\mathbf{a}/\mathbf{X}] \quad (f[\mathbf{X}] = c) \in D}{s \vdash_D^\delta f(\mathbf{a})} \\
\\
\frac{\delta \models P[a_1/A_1, a_2/A_2, r/R, t/T] \quad s \vdash_D^\delta c[a_1/A_1, a_2/A_2, r/R, t/T]}{\text{transmit}(a_1, a_2, r, t) s \vdash_D^\delta \text{transmit}((A_1, A_2, R, T|P)).c} \\
\\
\frac{s_1 \vdash_D^\delta c_1 \quad s_2 \vdash_D^\delta c_2 \quad (s_1, s_2) \rightsquigarrow s}{s \vdash_D^\delta c_1 \parallel c_2} \quad \frac{s_1 \vdash_D^\delta c_1 \quad s_2 \vdash_D^\delta c_2}{s_1 s_2 \vdash_D^\delta c_1; c_2} \\
\\
\frac{s \vdash_D^\delta c}{s \vdash_D^\delta \text{letrec } D \text{ in } c} \quad \frac{s \vdash_D^\delta c_1}{s \vdash_D^\delta c_1 + c_2} \quad \frac{s \vdash_D^\delta c_2}{s \vdash_D^\delta c_1 + c_2}
\end{array}$$

### 3.3 Contract Monitoring by Residuation

Extensionally, contracts classify traces (event sequences) into performing and nonperforming ones. We define the *extension* of a contract  $c$  to be the set of its performing executions:  $\mathcal{C}[[c]]^{D;\delta} = \{s : s \vdash_D^\delta c\}$ . We say  $c$  *denotes* a trace set  $S$  in context  $D, \delta$ , if  $\mathcal{C}[[c]]^{D;\delta} = S$ .<sup>3</sup>

We are not only interested in classifying complete event sequences once they have happened, though, but in *monitoring* contract execution as it unfolds in time under the arrival of events.

Given a trace set  $S$  denoted by a contract  $c$  and an event  $e$ , the *residuation function*  $\cdot/e$  captures how  $c$  can be satisfied if the first event is  $e$ . It is defined as follows:

$$S/e = \{s' \mid \exists s \in S : es' = s\}$$

Conceptually, we can map contracts to trace sets and use the residuation function to monitor contract execution as follows:

<sup>2</sup> We have not specified a particular language of Boolean expressions; we only require that it has a well-defined notion of substitution.

<sup>3</sup> A variant of  $\mathcal{C}[[c]]^{D;\delta}$  can be characterized compositionally, yielding a *denotational* semantics; see [AEH<sup>+</sup>04].

1. Map a given contract  $c_0$  to the trace set  $S_0$  that it denotes. If  $S_0 = \emptyset$ , stop and output “inconsistent”.
2. For  $i = 0, 1, \dots$  do:
  - Receive message  $e_i$ .
  - (a) If  $e_i$  is a transfer event, compute  $S_{i+1} = S_i/e_i$ . If  $S_{i+1} = \emptyset$ , stop and output “breach of contract”; otherwise continue.
  - (b) If  $e_i$  is a “terminate contract” message, check whether  $\langle \rangle \in S_i$ . If so, all obligations have been fulfilled and the contract can be terminated. Stop and output “successfully completed”. If  $\langle \rangle \notin S_i$ , output “cannot be terminated now”, let  $S_{i+1} = S_i$  and continue to receive messages.

To make the conceptual algorithm for contract life cycle monitoring from Section 3.3 *operational*, we need to represent the residual trace sets and provide methods for deciding tests for emptiness and failure. In particular, we would like to use contracts as representations for trace sets. Not all trace sets are denotable by contracts, however. In particular, given a contract  $c$  that denotes a trace set  $S_c$  it is not *a priori* clear whether  $S_c/e$  is denotable by a contract  $c'$ . If it is, we call  $c'$  the *residual contract of  $c$  after  $e$* .

### 3.4 Nullable and Guarded Contracts

In this section we characterize *nullability* of a contract and introduce *guarding*, which is a sufficient condition on contracts for ensuring that residuation can be performed by reduction on contracts.

---

**Fig. 5** Nullable contracts

---

$$\begin{array}{c}
 \frac{D \vdash c \text{ nullable} \quad (f[\mathbf{X}] = c) \in D}{D \vdash f(\mathbf{a}) \text{ nullable}} \quad \frac{D \vdash c \text{ nullable}}{D \vdash c + c' \text{ nullable}} \quad \frac{D \vdash c' \text{ nullable}}{D \vdash c + c' \text{ nullable}} \\
 D \vdash \text{Success} \text{ nullable} \quad \frac{D \vdash c \text{ nullable} \quad D \vdash c' \text{ nullable}}{D \vdash c \parallel c' \text{ nullable}} \quad \frac{D \vdash c \text{ nullable} \quad D \vdash c' \text{ nullable}}{D \vdash c; c' \text{ nullable}}
 \end{array}$$


---

Let us write  $D \models c$  nullable if  $\langle \rangle \in \mathcal{C}[[c]]^{D;\delta}$  for all  $\delta$ . We call such a contract *nullable* (or *terminable*): it can be concluded successfully, but may possibly also be continued. E.g., the contract  $\text{Success} + \text{transmit}(a_1, a_2, r, t|P)$  is nullable, as it may be concluded successfully (left choice). Note however, that it may also be continued (right choice). It is easy to see that nullability is independent of  $\delta$ :  $\langle \rangle \in \mathcal{C}[[c]]^{D;\delta}$  for some  $\delta$  if and only if  $\langle \rangle \in \mathcal{C}[[c]]^{D;\delta'}$  for any other  $\delta'$ . Deciding nullability is required to implement Step 2b in contract monitoring. The following proposition expresses that nullability is characterized by the inference system in Figure 5.

**Proposition 1.**  $D \models c$  nullable  $\iff D \vdash c$  nullable

A contract  $c$  is (*hereditarily*) *guarded* in context  $D$  if  $D \vdash c$  guarded is derivable from Figure 6; intuitively, guardedness ensures that in a contract with mutual recursion, we do not have (mutual) recursions such as  $\{f[\mathbf{X}] = g[\mathbf{X}], g[\mathbf{X}] = f[\mathbf{X}]\}$  that cause the residuation algorithm to loop infinitely.

### 3.5 Operational Semantics I: Deferred Matching

Residuation on trace sets tells us how to maintain the trace set under arrival of events. In this section we present a *reduction semantics* for contracts, which lifts residuation on trace sets to contracts and thus provides a *monitoring semantics* for contract execution.

**Fig. 6** Guarded contracts

$$\begin{array}{c}
D \vdash \text{Success guarded} \quad D \vdash \text{Failure guarded} \\
\\
D \vdash \text{transmit}(\mathbf{X} \mid P).c \text{ guarded} \quad \frac{D \vdash c \text{ guarded} \quad (f[\mathbf{X}] = c) \in D}{D \vdash f(\mathbf{a}) \text{ guarded}} \\
\\
\frac{D \vdash c \text{ guarded} \quad D \vdash c' \text{ guarded}}{D \vdash c + c' \text{ guarded}} \quad \frac{D \vdash c \text{ guarded} \quad D \vdash c' \text{ guarded}}{D \vdash c \parallel c' \text{ guarded}} \\
\\
\frac{D \vdash c \text{ guarded} \quad D \vdash c' \text{ guarded}}{D \vdash c; c' \text{ guarded}}
\end{array}$$

**Fig. 7** Deterministic reduction (delayed matching)

$$\begin{array}{c}
D, \delta \vdash_D \text{Success} \xrightarrow{e} \text{Failure} \quad D, \delta \vdash_D \text{Failure} \xrightarrow{e} \text{Failure} \\
\\
\frac{\delta \models P[\mathbf{a}/\mathbf{X}]}{D, \delta \vdash_D \text{transmit}(\mathbf{X} \mid P).c \xrightarrow{\text{transmit}(\mathbf{a})} c[\mathbf{a}/\mathbf{X}]} \quad \frac{\delta \not\models P[\mathbf{a}/\mathbf{X}]}{D, \delta \vdash_D \text{transmit}(\mathbf{X} \mid P).c \xrightarrow{\text{transmit}(\mathbf{a})} \text{Failure}} \\
\\
\frac{D, \delta \vdash_D c[\mathbf{a}/\mathbf{X}] \xrightarrow{e} c' \quad (f[\mathbf{X}] = c) \in D}{D, \delta \vdash_D f(\mathbf{a}) \xrightarrow{e} c'} \quad \frac{D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c + c' \xrightarrow{e} d + d'} \\
\\
\frac{D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c \parallel c' \xrightarrow{e} c \parallel d' + d \parallel c'} \quad \frac{D \vdash c \text{ nullable} \quad D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c; c' \xrightarrow{e} d; c' + d'} \\
\\
\frac{D \not\vdash c \text{ nullable} \quad D, \delta \vdash_D c \xrightarrow{e} d}{D, \delta \vdash_D c; c' \xrightarrow{e} d; c'} \quad \frac{D, \delta \vdash_D c \xrightarrow{e} c'}{\delta \vdash_D \text{letrec } D \text{ in } c \xrightarrow{e} \text{letrec } D \text{ in } c'}
\end{array}$$

The ability of representing residual contract obligations of a partially executed contract and thus any state of a contract as a *bona fide* contract carries the advantage that any analysis that is performed on “original” contracts automatically extends to partially executed contracts as well. E.g., an investment bank that applies valuations to financial contracts before offering them to customers can apply their valuations to their portfolio of contracts under execution; e.g., to analyze its risk exposure under current market conditions.

The reduction semantics is presented in Figure 7. The basic *matching rule* is

$$\frac{\delta \models P[\mathbf{a}/\mathbf{X}]}{D, \delta \vdash_D \text{transmit}(\mathbf{X} \mid P).c \xrightarrow{\text{transmit}(\mathbf{a})} c[\mathbf{a}/\mathbf{X}]}$$

It *matches* an event with a specific commitment in a contract. There may be multiple commitments in a contract that match the same event. The semantics captures the possibilities of matching an event against multiple commitments by applying all possible reductions in alternatives and concurrent contract forms and forming the sum of their possible outcomes (some of which may actually be Failure).

The rule

$$\frac{D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c + c' \xrightarrow{e} d + d'}$$

thus reduces both alternatives  $c$  and  $c'$  and then forms the sum of their respective results  $d, d'$ .

Finally, the rule

$$\frac{D \vdash c \text{ nullable} \quad D, \delta \vdash_D c \xrightarrow{e} d \quad D, \delta \vdash_D c' \xrightarrow{e} d'}{D, \delta \vdash_D c; c' \xrightarrow{e} d; c' + d'}$$

captures that  $e$  can be matched in  $c$  or, if  $c$  is nullable, in  $c'$ . Note that, if  $c$  is not nullable,  $e$  can only be matched in  $c$ , not  $c'$ , as expressed by the rule

$$\frac{D \not\vdash c \text{ nullable} \quad D, \delta \vdash_D c \xrightarrow{e} d}{D, \delta \vdash_D c; c' \xrightarrow{e} d; c'}$$

In this fashion the semantics keeps track of the results of all possible matches in a reduction sequence as explicit *alternatives* (summands) and *defers* the decision as to *which specific* commitment is matched by a particular event during contract execution until the very end: By selecting a particular summand in a residual contract after a number of reduction steps that represents Success (and the contract is thus terminable) a particular set of matching decisions is chosen *ex post*. As presented, the reduction semantics gives rise to an implementation in which the multiple reducts of previous reduction steps are reduced in parallel, since they are represented as summands in a single contract, and the rule for reduction of sums reduces both summands. It is relatively straightforward to turn this into a backtracking semantics by an asymmetric reduction rule for sums, which delays reduction of the right summand.

Guardedness is key to ensuring termination of contract residuation and thus that every (guarded) contract has a residual contract under any event in the reduction semantics of Figure 7.

**Theorem 1.** *If  $c \in \mathcal{C}^{\mathcal{P}}$  is guarded then for each event  $e$  there exists a unique  $c' \in \mathcal{C}^{\mathcal{P}}$  such that  $D, \delta \vdash_D c \xrightarrow{e} c'$ . Furthermore, we have that  $c'$  is guarded and  $D, \delta \models c/e = c'$ , which means  $\mathcal{C}[c]^{D;\delta}/e = \mathcal{C}[c']^{D;\delta}$ .*

Using this reduction semantics we can turn our conceptual contract monitoring algorithm into a real algorithm.

Proposition 1 provides a syntactic characterization of nullability, which can easily (not trivially) be turned into an algorithm. Inconsistency—whether a contract denotes the empty trace set or not—is not treated here; see the full report [AEH<sup>+</sup>04].

### 3.6 Operational Semantics II: Eager Matching

The deferred matching semantics of Figure 7 is flexible and faithful to the natural notion of contract satisfaction as defined in Figure 4. But from an accounting practice point of view it is weird because matching decisions are deferred. In bookkeeping standard *modus operandi* is that events are matched against specific commitments *eagerly*; that is online, as events arrive.<sup>4</sup>

We shall turn the deferred matching semantics of Figure 7 into an eager matching semantics (Figure 8). The idea is simple: Represent here-and-now choices as alternative *rules* (meta-level) as opposed to alternative contracts (object level). Specifically, we split the rules for reducing alternatives and concurrent subcontracts into multiple rules, and we capture the possibility of reducing in the second component of a sequential contract by adding  $\tau$ -transitions, which “spontaneously” (without a driving external event) reduce a contract of the form Success;  $c$  to  $c$ . For this to be sufficient we have to make sure that a nullable contract indeed can be reduced to Success, not just a contract that is *equivalent* with Success, such as Success  $\parallel$  Success. This is done by ensuring that  $\tau$ -transitions are strong enough to guarantee reduction to Success as required.

<sup>4</sup> There are standard accounting practices for changing such decisions, but both default and standard conceptual model are that matching decisions are made as early as possible. In general, it seems representing and deferring choices and applying *hypothetical* reasoning to them appears to be a rather unusual phenomenon in accounting.

**Fig. 8** Nondeterministic reduction (eager matching)

$$\begin{array}{c}
\frac{D, \delta \vdash_N \text{Success} \xrightarrow{e} \text{Failure} \quad D, \delta \vdash_N \text{Failure} \xrightarrow{e} \text{Failure}}{\frac{\delta \models P[\mathbf{a}/\mathbf{X}]}{D, \delta \vdash_N \text{transmit}(\mathbf{X} | P). c \xrightarrow{\text{transmit}(\mathbf{a})} c[\mathbf{a}/\mathbf{X}]} \quad \frac{\delta \not\models P[\mathbf{a}/\mathbf{X}]}{D, \delta \vdash_N \text{transmit}(\mathbf{X} | P). c \xrightarrow{\text{transmit}(\mathbf{a})} \text{Failure}}} \\
\frac{(f[\mathbf{X}] = c) \in D}{D, \delta \vdash_N f(\mathbf{a}) \xrightarrow{\tau} c[\mathbf{a}/\mathbf{X}]} \quad D, \delta \vdash_N c + c' \xrightarrow{\tau} c \quad D, \delta \vdash_N c + c' \xrightarrow{\tau} c' \\
\frac{D, \delta \vdash_N c \xrightarrow{\lambda} d}{D, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} d \parallel c'} \quad \frac{D, \delta \vdash_N c' \xrightarrow{\lambda} d'}{D, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} c \parallel d'} \\
D, \delta \vdash_N \text{Success} \parallel c \xrightarrow{\tau} c \quad D, \delta \vdash_N c \parallel \text{Success} \xrightarrow{\tau} c \quad D, \delta \vdash_N \text{Success}; c' \xrightarrow{\tau} c' \\
\frac{D, \delta \vdash_N c \xrightarrow{\lambda} d}{D, \delta \vdash_N c; c' \xrightarrow{\lambda} d; c'} \quad \frac{D, \delta \vdash_N c \xrightarrow{e} c'}{\delta \vdash_N \text{letrec } D \text{ in } c \xrightarrow{e} \text{letrec } D \text{ in } c'}
\end{array}$$

Based on these considerations we arrive at the reduction semantics in Figure 8, where meta-variable  $\lambda$  ranges over events  $e$  and the internal event  $\tau$ . Note that it is nondeterministic and not even confluent: A contract  $c$  can be reduced to two different contracts by the same event. Consider e.g.,  $c = a; b + a; b'$  where  $a, b, b'$  are commitments with suitable  $D, \delta$ , no two of which match the same event. For event  $e$  matching  $a$  we have  $D, \delta \vdash_N c \xrightarrow{e} b$  and  $D, \delta \vdash_N c \xrightarrow{e} b'$ , but neither  $b$  nor  $b'$  can be reduced to **Success** or any other contract by the same event sequence. In reducing  $c$  we have not only resolved it against  $e$ , but also made a *decision*: whether to apply it to the first alternative of  $c$  or to the second. Technically, the reduction semantics is not closed under residuation: Given  $c$  and  $e$  it is not always possible to find  $c'$  such that  $D, \delta \vdash_N c \xrightarrow{e} c'$  and  $D; \delta \models c/e = c'$ . It is sound, however, in the sense that the reduct always denotes a subset of the residual trace set:

- Proposition 2.** 1. If  $D, \delta \vdash_N c \xrightarrow{e} c'$  then  $D, \delta \models c' \subseteq c/e$ .  
2. If  $D, \delta \vdash_N c \xrightarrow{\tau} c'$  then  $D, \delta \models c' \subseteq c$ .

Even though individual eager reductions do not preserve residuation, the set of all reductions does so:

- Proposition 3.** If  $D, \delta \vdash_D c \xrightarrow{e} c'$  then there exist contracts  $c_1, \dots, c_n$  for some  $n \geq 1$  such that  $D, \delta \vdash_N c \xrightarrow{\tau^*} c'_i \xrightarrow{e} c_i$  for all  $i = 1 \dots n$  and  $D, \delta \models c' \subseteq \sum_{i=1}^n c_i$ . The notation  $\cdot \xrightarrow{\tau^*} \cdot$  indicates any number  $\geq 0$  of  $\tau$ -transitions.

As a corollary, Propositions 2 and 3 combined yield that the object-level nondeterminism (expressed as contract alternatives) in the deferred matching semantics is faithfully reflected in the meta-level nondeterminism (expressed as multiple applicable rules) of the eager matching semantics.

### 3.7 Operational Semantics III: Eager Matching with Explicit Routing

Consider the following execution model for contracts: Two or more parties each have a copy of the contract they have previously agreed upon and monitor its execution under the arrival of events. Even though they agree on prior contract state and the next event, the parties may



arrive at different residual contracts and thus different expectations as to the future events allowed under the contract. This is because of nondeterminacy in contract execution with eager matching; e.g., a payment of \$50 may match multiple payment commitments, and the parties may make different matches. We can remedy this by making *control* of contract reduction with eager matching explicit in order to make reduction deterministic: events are accompanied by control information that unambiguously prescribes how a contract is to be reduced. In this fashion parties that agree on what events have happened and on their associated control information, will reduce their contract identically. See the full technical report for details [AEH<sup>+</sup>04].

## 4 Example Contracts

For the purpose of demonstration we will afford ourselves a fairly advanced predicate language with basic arithmetic, logical connectives, lists and basic functions. The syntax is standard and straightforward, and the details will be obvious from the examples.

Consider the validity period specified in Section 3 of the Agreement to Provide Legal Services (Figure 2). Taken literally, it would imply, that the attorney shall render services in the month of December, but receive no fee in consideration since January 2005 is outside the validity period. Surely, this is not the intention; in fact, consideration will defeat most deadlines as is clearly the intent here. In the coding of the Agreement to Provide Legal Services the expiration date `end` has to be pushed down on all transmits despite its global nature to make sure that consideration would not be cut off.

The Agreement to Provide Legal Services fails to specify who decides if legal services should be rendered. In the coding it is simply assumed that the attorney is the initiator and that all services rendered over a month can be modelled as one event. Furthermore, the attorney is assumed to give the notice `nowork` if no work was done for the past month. This is an artifact introduced to guard the recursive call to `legal`.

---

### Fig. 9 Software Development Agreement

---

**Section 1.** The Developer shall develop software as described in Exhibit A (Requirements Specification) according to the schedule set forth in Exhibit B (Project Schedule and Deliverables). Specifically, the Developer shall be responsible for the timely completion of the deliverables identified in Exhibit B.

**Section 2.** The Client shall provide written approval upon the completion of each deliverable identified in Exhibit B.

**Section 3.** In the event of any delay by the Client, all the Developer's remaining deadlines shall be extended by the greater of the two following: (i) five working days, (ii) two times the delay induced by the Client. The Client's deadlines shall be unchanged.

**Section 4.** In consideration of services rendered the Client shall pay USD \$100,000 due on 7/1.

**Section 5.** If the Client wishes to add to the order, or if upon written approval of a deliverable, the Client wishes to make modifications to the deliverable, the Client and the Developer shall enter into a Change Order. Upon mutual agreement the Change Order shall be attached to this contract.

**Section 6.** The Developer shall retain all intellectual rights associated with the software developed. The Client may not copy or transfer the software to any third party without the explicit, written consent of the Developer.

**Exhibit A.** (omitted)

**Exhibit B.** Deadlines for deliverables and approval: (i) 1/1, 1/15; (ii) 3/1, 3/15, (final deadline) 7/1, 7/15.

---

Now consider the more elaborate Software Development Agreement in Figure 9. When coding the contract, one notices that the contract fails to specify the ramifications of the client's

**Fig. 10** Specification of Software Development Agreement – note that we assume (easily defined) abbreviations for  $\max(x, y)$  and allow subtraction on the domain Time.

---

```

letrec
  deliverables (dev, client, payment, deliv1, deadline1, approv1,
                deliv2, deadline2, approv2,
                delivf, deadlinef, approvf) =
    transmit(dev, client, deliv1, T1 | T1 <= deadline1)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, deliv2, T2 |
              T2 <= deadline2 + max(5d, (T - approv1) * 2)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, delivf, Tf |
              Tf <= deadlinef + max(5d, (T - approv2) * 2)).
    transmit(client, dev, "ok", T).
    transmit(dev, client, "done", T).
  Success

  software (dev, client, payment, paymentdeadline, ds) =
    deliverables (dev, client, deliv1, deadline1, approv1,
                  deliv2, deadline2, approv2,
                  delivf, deadlinef, approvf) ||
    transmit(client, dev, payment, T | T <= paymentdeadline)
in
  software ("Me", "Client", 100000, 2004.7.1, d1, 2004.1.1, 2004.1.15,
           d2, 2004.3.1, 2004.3.15, final, 2004.7.1, 2004.7.15)

```

---

non-approval of a deliverable. One also sees that the contract does not specify what to do if due to delay, some approval deadline comes before the postponed delivery date. In the current code, this is taken to mean further delay on the client's part even if the client gave approval at the same time as the deliverable was transmitted. It seems that contract coding is a healthy process in the sense that it will often unveil underspecification and errors in the natural language contract being coded. The Change Order described in Section 5 of the contract and the intellectual rights described in Section 6 are not coded due to certain limitations in our language. We will postpone the discussion of this this paper's Section 6.

## 5 Contract Analysis

The formal groundwork in order, we can begin to ask ourselves questions about contracts such as: What is my first order of business? When is the next deadline? How much of a particular resource will I gain from my portfolio and at what times? What is the monetary value of my portfolio? Will contract fulfillment require more than the  $x$  units I currently have in stock?

The attempt to answer such questions is broadly referred to as *contract analysis*. The residual property allows a contract analysis to be applied at any time (i.e. to any residual contract), and we can thus continuously monitor the execution of the contracts in our portfolio.

Recall that our contract specification language is parameterized over the language of predicates and arithmetic. There is a clear trade-off in play here: a sophisticated language buys expressiveness, but renders most of the analyses undecidable.

There is another source of difficulties. Variables may be bound to components of an event that is unknown at the time of analysis. An expression like  $\text{transmit}(a_1, a_2, R, T | \text{true})$ . offers little insight into the nature of  $R$  unless furnished with a probability vector over all resources.

Here we will circumvent these problems by making do with a restricted predicate language and accepting that analyses may not give answers on all input (but will give correct answers).

The predicate language is plugged in at two locations. In function application  $f(\mathbf{a})$  where all components of the vector  $\mathbf{a}$  must be checked according to the rules of the predicate language, and in  $\text{transmit}(a_1, a_2, r, t|P)$  where  $P$  must have the type Boolean. As previously we require that  $a_1, a_2, r$ , and  $t$  are either variables (bound or unbound) or constants. If some components are bound variables or constants, they must be equal to the corresponding components of an incoming event  $(a'_1, a'_2, r', t')$  for a match to occur.

Consider the syntax provided in figure 11. In addition to the types Agent, Resource, and Time, the language has the fundamental types Int and Boolean. Take  $\tau$  to range over  $\{\text{Int}, \text{Time}\}$ , take  $\sigma$  to range over  $\tau \cup \{\text{Agent}, \text{Resource}\}$ , and assume that constants can be uniquely typed (e.g. time constants are in ISO format, and agent and resource constants are known).

The language allows arithmetic on integers, simple propositional logic, and manipulation of the two abstract types Resource and Time. Given a time (date)  $t$  we may add an integral number of years, months or days. For example  $2004.1.1 + 3d + 1y$  yields  $2005.1.4$ . Resources permit a projection on a named component (field) and all fields are of type Int. E.g. to extract the total amount from an information resource named *invoice* we write  $\#(\text{invoice}, \text{total}, t)$  where  $t$  is some date<sup>5</sup>. The fields of resources may change over time; hence the third Time parameter.

Observables can now be understood simply as fields of a ubiquitous resource named *obs*. An Int may double for a Resource in which case the Int is understood to be a currency amount.

**Fig. 11** Example syntax for predicate language

---

$\frac{\Delta \vdash \Delta(\text{var}) = \sigma}{\Delta \vdash \text{var} : \sigma}$	$\frac{\Delta \vdash \text{type}(\text{const}) = \sigma}{\Delta \vdash \text{const} : \sigma}$	$\frac{\Delta \vdash e_1 : \text{Int} \quad \Delta \vdash e_2 : \text{Int} \quad \text{op} \in \{+, -, *, /\}}{\Delta \vdash e_1 \text{ op } e_2 : \text{Int}}$
$\frac{\Delta \vdash t : \text{Time} \quad \Delta \vdash e : \text{Int} \quad f \in \{\text{y}, \text{m}, \text{d}\} \quad \text{op} \in \{+, -\}}{\Delta \vdash t \text{ op } e f : \text{Time}}$		$\frac{\Delta \vdash e : \text{Time} \quad f \in \{\text{y}, \text{m}, \text{d}\}}{\Delta \vdash e \# f : \text{Int}}$
$\frac{\Delta \vdash r : \text{Resource} \quad \Delta \vdash t : \text{Time} \quad f \in \text{fields}(r)}{\Delta \vdash \#(r, f, t) : \text{Int}}$		$\frac{\Delta \vdash e : \text{Int}}{\Delta \vdash e : \text{Resource}}$
$\frac{\Delta \vdash e_1 : \tau \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1 < e_2 : \text{Boolean}}$		$\frac{\Delta \vdash e_1 : \sigma \quad \Delta \vdash e_2 : \sigma}{\Delta \vdash e_1 = e_2 : \text{Boolean}}$
$\frac{\Delta \vdash b_1 : \text{Boolean} \quad \Delta \vdash b_2 : \text{Boolean} \quad \text{op} \in \{\text{and}, \text{or}\}}{\Delta \vdash b_1 \text{ op } b_2 : \text{Boolean}}$		$\frac{\Delta \vdash b : \text{Boolean}}{\Delta \vdash \text{not } b : \text{Boolean}}$

---

Ideally, a contract analysis can be performed *compositionally*, i.e. can be implemented by recursively evaluating subcontracts. This section contains a simple analysis with this property. Space considerations prevent a walkthrough of more involved examples, but the basic idea should be clear. We will assume for simplicity that recursively defined contracts are *guarded*. The analyses are presented using inference systems defined by induction on syntax, emphasizing the declarative and compositional nature of the analyses.

<sup>5</sup> When a resource is introduced into the system through a match, it must be dynamically checked that it possesses the required fields. The set of required fields can be statically determined by a routine type check annotating resources with field names à la  $\{\text{date}, \text{total}, \text{paymentdeadline}\}\text{Resource}$ . To keep things simple we omit this type extension here.

## 5.1 Example: Next Point of Interest and Task List

Given a contract or a portfolio of contracts it is tremendously important for an agent to know when and how to act. To this end we demonstrate how a very simple *task list* can be compiled.

Consider the definition given in Figure 12. The function gives a structured response to reflect the decision structure (the task list) of the contract. It operates on a very simple subset of the predicate language that, however, is indicative of the bulk of temporal constraints in contracts: only interval conditions of the form  $a \leq T$  and  $T \leq b$  with  $T$  the time variable in the enclosing transmit commitment are admitted. Such a condition is abbreviated to  $[a; b]$ . It is important to notice that the result of the analysis may be incomplete. A task is only added if the agents agree (i.e.  $a = a_1$ ), but if  $a_1$  is not bound at the time of analysis, the task is simply skipped. A more elaborate dataflow analysis might reveal that in fact  $a_1$  is always bound to  $a$ .

Also notice the case for application  $f(a)$ . We expand the body of the named contract  $f$  given arguments  $a$  but only once. This measure ensures termination of the analysis, but reduces the function's look-ahead horizon. Hence, any task or point of interest more than one recursive unfolding away is not detected. This is unlikely to have practical significance for two reasons: (1) recursively defined contracts are guarded and so a `transmit` must be matched before a new unfold can occur. This `transmit` therefore is presumably more relevant than any other `transmits` further down the line; (2) it would be grossly unidiomatic that some `transmit`  $t_1$  was required to be matched before another `transmit`  $t_2$ , but nevertheless had a later deadline than that of  $t_2$ .

---

**Fig. 12** Task list analysis

---

$$\begin{array}{c}
D, \delta, a, t \vdash \text{Success} : [] \quad D, \delta, a, t \vdash \text{Failure} : [] \\
\\
\frac{\models a \neq a_1 \quad \mathbf{X} = (a_1, A, R, T)}{D, \delta, a, t \vdash \text{transmit}(\mathbf{X} \mid [x; y]).c : \text{do } []} \quad \frac{\models t \notin [x; y]}{D, \delta, a, t \vdash \text{transmit}(\mathbf{X} \mid [x; y]).c : \text{do } []} \\
\\
\frac{\models a = a_1 \quad \mathbf{X} = (a_1, A, R, T) \quad t \in [x; y]}{D, \delta, a, t \vdash \text{transmit}(\mathbf{X} \mid [x; y]) : \text{do } [\text{transmit}(\mathbf{X} \mid [x; y])]} \\
\\
\frac{D, \delta, a, t \vdash c_1 : l_1 \quad D, \delta, a, t \vdash c_2 : l_2}{D, \delta, a, t \vdash c_1 + c_2 : \text{choose}[l_1, l_2]} \\
\\
\frac{D \vdash c_1 \text{ nullable} \quad D, \delta, a, t \vdash c_1 : l_1 \quad D, \delta, a, t \vdash c_2 : l_2}{D, \delta, a, t \vdash c_1; c_2 : \text{choose}[l_1, l_2]} \\
\\
\frac{D \not\vdash c_1 \text{ nullable} \quad D, \delta, a, t \vdash c_1 : l_1}{D, \delta, a, t \vdash c_1; c_2 : l_1} \quad \frac{D, \delta, a, t \vdash c_1 : l_1 \quad D, \delta, a, t \vdash c_2 : l_2}{D, \delta, a, t \vdash c_1 \parallel c_2 : l_1 @ l_2} \\
\\
\frac{(f[\mathbf{X}] = c) \in D \quad D, \delta, a, t \vdash c : l}{D, \delta, a, t \vdash f(a) : l}
\end{array}$$


---

The examples given above, in their simplicity, may be extended given knowledge of the problem domain. In particular, knowledge of or forecasting about probable event sequences may be used in a manner “orthogonal” to the coding of analyses by appropriate function calls.

Analyses that are possible to implement in this way include resource flow forecasting (supply requirements); terminability by agent; latest termination; earliest termination; and valuation, or simply put: What is the value to an agent of a given contract?

## 6 Discussion and Future Work

The Software Development Agreement (Figure 9) provides a good setting to observe the limitations to our approach and the ramifications of the design choices made.

The Change Order is not coded. It might be cleverly coded in the current language, again using constraints on the events passed around, but a more natural way would be using higher-order contracts, i.e. contracts taking contracts as arguments. Thus, a Change Order would simply be the passing back and forth of a contract followed by an instantiation upon agreement.

Contracts often specify certain things that are not to be done (e.g. not copying the software). Such restrictions should intersect all other outstanding contracts and limit them appropriately. A higher-order language or predicates that could guard all `transmits` of an entire subcontract might ameliorate this in a natural way.

A fuller range of language constructions that programmers are familiar with is also desirable; in the present incarnation of the contract language, several standard constructions have been left out in order to emphasize the core event model. In practice, conditionals and various sorts of lambda abstractions would make the language easier to use, though not strictly more expressive, as they can be encoded through events, albeit in a non-intuitive way. A conditional that is *not* driven by events (i.e. an if-then-else) seems to be needed for natural coding in many real-world contracts. Also, a catch-throw mechanism for unexpected events would make contracts more robust.

Conversely, certain features of the language appear to be almost too strong for the domain; the inclusion of full recursion means that contracts active for an unlimited period of time, say leases, are easy to code, but make contract analysis significantly harder. In practice, contracts running for “unlimited” time periods often have external constraints (usually local legislation) forcing the contract to be reassessed by its parties, and possibly government representatives, from time to time. Having only a restricted form of recursion that suffices for most practical applications should simplify contract analysis.

The expressivity of the contract language and indeed the feasibility of non-trivial contract analysis depends heavily on the predicate language used. Predicates restricted to the form  $[a; b]$  are surely too limited, and further investigation into the required expressiveness of the predicate language is desirable.

While the language is parametrized over the predicate language used, almost all real-world applications will require some model of time and timed events to be incorporated. The current event model allows for encoding through the predicate language, but an extended set of events, with companion semantics, would make for easier contract programming; timer (or “trigger”) events appear to be ubiquitous when encoding contracts.

## 7 Related Work

The impetus for this work comes from two directions: the REA accounting model pioneered by McCarthy [McC82] and Peyton Jones, Eber and Seward’s seminal article on specification of financial contracts [JES00]. Furthermore, given that contracts specify protocols as to how parties bound by them are to interact with each other there are links to process and workflow models.

Peyton Jones, Eber and Seward [JES00] present a compositional language for specifying financial contracts. It provides a decomposition of known standard contracts such as zero coupon bonds, options, swaps, straddles, etc., into individual payment commitments that are combined declaratively using a small set of contract combinators. All contracts are two-party contracts, and the parties are implicit. The combinators (taken from [JE03], revised from [JES00]) correspond to `Success`, `· || ·`, `· + ·`, `transmit(·)` of our language  $\mathcal{C}^P$ ; it has no direct counterparts to `Failure`, `· ; ·` nor, most importantly, recursion or iteration. On the other hand, it

provides conditionals and predicates that are applicable to arbitrary contracts, not just commitments as in  $\mathcal{C}^P$ , something we have found to be worthwhile also for specifying commercial contracts.

Our contract language generalizes financial payment commitments to arbitrary transfers of resources and information, provides explicit agents and thus provides the possibility of specifying multi-party contracts.

Disregarding the structure of events and their temporal properties,  $\mathcal{C}^P$  is basically a process algebra. It corresponds to Algebra of Communicating Processes (ACP) with deadlock (Failure), free merge ( $\cdot \parallel \cdot$ ) and recursion, but without encapsulation [BW90]. This process algebra is also part of CSP [BHR84,Hoa85]. Note that contracts are to be thought as exclusively *reactive* processes, however: they respond to externally generated events, but do not autonomously generate them.

There are numerous timed variants of process algebras and temporal logics; see e.g. Baeten and Middelburg [BM02] for timed process algebras. Their relation to our base language is not evident at this point. This is in part because our base language is not fixed yet to accommodate expressing temporal (and other) constraints “naturally,” in part because the temporal notions of timed process languages seem rather low-level and distinct from the notions we have used in contract examples.

## 8 Acknowledgements

This work has been partially funded by the NEXT Project, which is a collaboration between Microsoft Business Solutions, The IT University of Copenhagen and the Department of Computer Science at the University of Copenhagen (DIKU). See <http://www.itu.dk/next> for more information on NEXT.

We would like to thank Simon Peyton Jones, Jean-Marc Eber, Kasper Østerbye, and Jesper Kiehn for valuable discussions on modeling financial contracts and extending that work to commercial contracts based on the REA accounting model.

## References

- [AE03] Jesper Andersen and Ebbe Elsberg. Compositional specification of commercial contracts. M.S. term project, December 2003.
- [AEH<sup>+</sup>04] Jesper Andersen, Ebbe Elsberg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. Technical report, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, July 2004. <http://topps.diku.dk/next/contracts>.
- [BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
- [BM02] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. Springer, 2002.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
- [Ebe02] Jean-Marc Eber. Personal communication, June 2002.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
- [JE03] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Palgrave Macmillan, 2003.
- [JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292. ACM Press, 2000.
- [McC82] William E. McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, LVII(3):554–578, July 1982.

# The Size-Change Termination Principle on Non-Well founded Data Types

James Avery (avery@diku.dk)  
Dep. of Computer Science, University of Copenhagen

September 19, 2005

## Abstract

Despite its simplicity, the size-change termination principle (SCT), presented by Lee, Jones and Ben-Amram in [LJB01], is surprisingly strong and is able to show termination for a large class of programs.

A significant limitation for its use, however, is the fact that the SCT requires data types to be well-founded, and that all mechanisms used to determine termination must involve decreases in these global, well-founded partial orders.

In the following, I will present an extension of the size-change principle that allows for non-well founded data types, and a realization of this principle for integer data types.

The extended size-change principle is realized through combining abstract interpretation over the domain of convex polyhedra with the use of size-change graphs. In the cases when data types *are* well founded, the method handles every case that is handled by LJB size-change termination.

The method has been implemented in a subject language independent shared library, as well as in the ANSI C specializer `C-MixIT`, for a subset of its internal language `Core-C`.

## 1 Introduction

The question of program termination is one of the classical undecidable problems. This notwithstanding, termination can be proved in quite many cases, either by constructing programming languages in such a way that only a certain class of terminating programs are expressible, or in a fuller language, by reasoning about the semantics of individual programs. [LJB01] presents a practical technique for the latter approach, by reasoning about data flow combined with well founded partial orders on the domains of data types.

These notes describe work in progress being done on a project at DIKU, [Ave05], supervised by Prof. Neil D. Jones. The goal of the project is to extend the size-change termination principle of [LJB01] to an imperative language with integer data types. Besides the issue of data types not being well founded is the challenge that imperative programs rarely terminate for reasons expressible by a global order on the data type (as is illustrated by the example in section 2), thus making size-change termination analysis difficult. As it turns out, both these issues can be addressed by replacing the requirement of well foundedness by the existence of local bounds on program state.

Many of the ideas presented in the following grew out of a project with Stefan Kumarage Schou ([AS04]) in 2004. We developed the methods in a rough form to help determine quasi-termination and generate binding time divisions in `C-MixIT` that guarantee that partial evaluation terminates. As such, much is owed to our supervisors at the time, Prof. John Gallagher and Arne Glenstrup.

Current work focuses on the development of a mathematically sound foundation for an extended size-change termination principle. To avoid redundancy, the reader is assumed to be familiar with the contents of [LJB01].

## 2 Extending SCT to Non-Well founded Types

Although much more powerful than its simplicity suggests, the size-change termination principle is limited in applicability by its requirement that data types must be well-founded.

Of course, any data type with values representable on a computer can be made well-founded by an appropriate ordering, since the domain is inherently countable and so can be bijectively mapped to  $\mathbb{N}$ . A possible strategy when one wishes to implement SCT for a particular language is then to look for usable well-ordering relations for each type in the language, along with operations that decrease the ranks of elements within these orderings.

However, it isn't at all obvious that such an ordering will correspond very well with the mechanisms that make programs terminate. Rather, it would seem somewhat unlikely. Types such as e.g. trees and lists are "naturally" well-founded, and this property is used consciously by programmers to make algorithms terminate. But if there is no such natural well-founded partial order, trying to find one that renders a significant class of programs size-change terminating seems like an exercise in futility.

An alternate course of action is to attempt to replace the requirement of well-founded data types with a less restrictive property that still allows an extended size-change analysis to prove termination.

### 2.1 Rough Overview

The basic idea presented in this report is to replace the requirement of well-founded data types, a property of the language, by discovering bounds on program state. These should not only be specific to an individual program, but also specific to the program points. With these in hand, one detects through size-change analysis the property that any infinite computation must violate such a bound. This has the added benefit that even when data types are well founded, a larger class of programs size-change terminate under the new condition, since relations among variables are taken into account.

Before delving into the details of how one actually goes about performing such an analysis, let's first take an informal look at a small example in order to build some intuition for the methods presented in the following sections. Consider the tiny C program snippet in figure 1: a simple double loop with two designated program points, *A* and *B*:

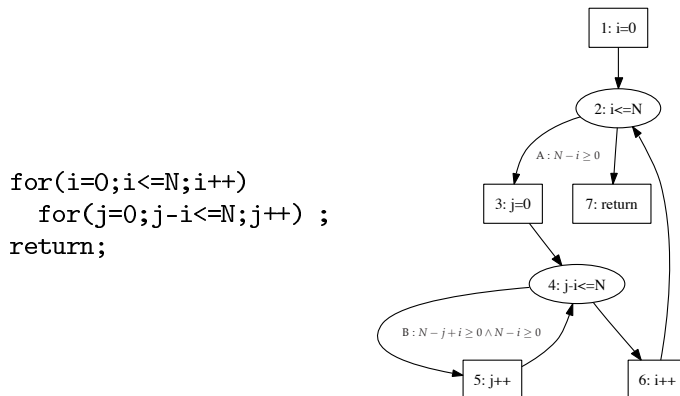


Figure 1: Simple double-loop



Despite its simplicity, no ordering of  $\mathbb{Z}$  would seem to make this program size-change terminate. But we immediately see that the inner loop terminates because  $N - j + i$  is bounded from below, and this expression is decreased in each iteration. Similarly for  $N - i$  in the outer loop.

Let's attempt to discover this property through size-change analysis. We break the analysis into three pieces:

1. **Obtain constraints that hold at each program point.**

In this example, we only look at arcs  $A$  and  $B$ , and we derive the inequalities in fig. 1 directly from the branch-tests.

2. **Construct size-change graphs for the basic blocks that change memory.**

The basic blocks that alter the store in the example are the blocks labeled 1, 3, 5 and 6 in figure 1. Their size-change graphs (SCGs) are:

$$\begin{array}{cccc} \overline{\mathcal{G}_1} & \overline{\mathcal{G}_3} & \overline{\mathcal{G}_5} & \overline{\mathcal{G}_6} \\ \begin{array}{c} N \rightarrow N \\ i \rightarrow i \\ j \rightarrow j \end{array} & \begin{array}{c} N \rightarrow N \\ i \rightarrow i \\ j \rightarrow j \end{array} & \begin{array}{c} N \rightarrow N \\ i \rightarrow i \\ j \uparrow j \end{array} & \begin{array}{c} N \rightarrow N \\ i \uparrow i \\ j \rightarrow j \end{array} \end{array} \quad (1)$$

3. **Combine this information to show that no infinite computation paths exist.**

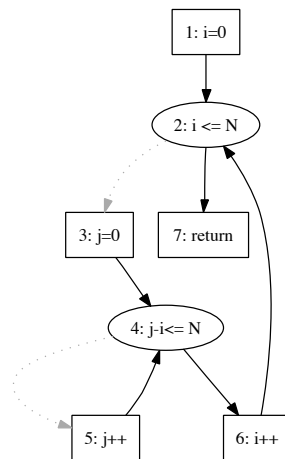
Let  $\mathcal{S}$  be the transitive closure of size-change graphs for all finite program point sequences. The subsets of graphs that go from point  $A$  back to  $A$ , and from  $B$  to  $B$ , have these data-flow relations:

$$\mathcal{S}_A = \left\{ \overline{\begin{array}{c} \mathcal{G}_3/\mathcal{G}_6 \\ N \rightarrow N \\ i \uparrow i \\ j \rightarrow j \end{array}} \right\}, \quad \mathcal{S}_B = \left\{ \overline{\begin{array}{c} \mathcal{G}_5 \\ N \rightarrow N \\ i \rightarrow i \\ j \uparrow j \end{array}}, \overline{\begin{array}{c} \mathcal{G}_5/\mathcal{G}_6/\mathcal{G}_3 \\ N \rightarrow N \\ i \uparrow i \\ j \rightarrow j \end{array}} \right\} \quad (2)$$

Thus, every time a computation executes in a path leading from  $A$  back to  $A$ ,  $N$  is preserved and  $i$  is decreased. The net result is a decrease of the expression  $N - i$ , which is bounded from below. Similarly, any computation segment going from  $B$  back to  $B$  will either decrease the expression  $N - i$  or decrease  $N - j + i$ , and both are non-negative.

Now, because the paths that decrease  $N - j + i$  at the same time increase  $N - i$ , one would intuitively suspect that an infinite computation path might possibly fail to decrease either of the expressions infinitely - i.e., although every finite computation path  $B \rightarrow B$  decreases one or the other, in an infinite sequence they could cancel out. We prove in theorem 2.8 that this can never happen.

This implies that  $A$  and  $B$  can only be visited finitely many times during any computation. Therefore, after a certain point in the sequence, any allegedly infinite computation path must run entirely within the graph with the arcs  $A$  and  $B$  removed:



But removing  $A$  and  $B$  from the program's flow-graph has left a graph with no strongly connected components, and thus no cycles. Consequently, the program as a whole must terminate.

The remainder of these notes supplies a formal framework for the method hinted at above. Although the formalism is only developed for integers, the general strategy - combining local lower bounds found by abstract interpretation with size-change analysis - could be applicable also to similar non-well founded data types.

## 2.2 Semantic Foundations

The subject language used in the following is imperative and without function calls, stack or dynamic memory. All variable values are integers.

### Definition 2.1.

1. A store is a tuple  $\vec{x} \in \mathbb{Z}^N$ , where  $N$  is the number of program variables.  $\mathbf{V} = \{x_1, \dots, x_N\}$  is the set of variable names.
2. A basic block is a sequence of assignments followed by a jump or a return.
3. A program point  $p$  is the transition between two basic blocks  $b$  and  $b'$ , corresponding to the arc from  $b$  to  $b'$  in the programs flow-graph. The program entry point is denoted  $p_0$ .
4. A state is a pair  $(p, \vec{x})$ .
5. The one-step state transition relation has the form  $(p, \vec{x}) \rightarrow (p', \vec{x}')$ .
6. A finite computation path is a finite program point sequence

$$\pi = p_1, p_2, \dots, p_m$$

following program control. We will also write  $\pi : p_1 \xrightarrow{*} p_m$ , and  $\pi : p_1 \xrightarrow{\pm} p_m$  if  $m > 0$ . In the case when  $m = 2$ , i.e.  $\pi = p_1, p_2$  is a single transition, we write  $\pi : p_1 \rightarrow p_2$ .

7. An infinite computation path is a sequence  $\pi = (p_i)_{\mathbb{N}_0}$  following program control.
8. For a finite computation path  $\pi : p_1 \xrightarrow{*} p_m$ , define its store transformation by

$$\pi \vdash \vec{x}_1 \xrightarrow{*} \vec{x}_m$$

iff

$$(p_1, \vec{x}_1) \rightarrow (p_2, \vec{x}_2) \rightarrow \dots \rightarrow (p_m, \vec{x}_m)$$

It is a partial function  $\mathbb{Z}^N \rightarrow \mathbb{Z}^N$ .

9. A store  $\vec{x}$  is reachable at  $p$  iff there exists an initial store  $\vec{x}_0$  and a finite computation path  $\pi : p_0 \xrightarrow{*} p$  such that

$$\pi \vdash \vec{x}_0 \xrightarrow{*} \vec{x}$$

We write  $\text{Reach}(p)$  for the set of all stores reachable at  $p$ .

**Definition 2.2** (Finite Computation Paths as Functions). For notational convenience, we will write  $\pi(\vec{x}) = \vec{x}'$  iff  $\pi \vdash \vec{x} \xrightarrow{*} \vec{x}'$ . We define the domain of  $\pi$  as

$$D(\pi) := \left\{ \vec{x} \in \mathbb{Z}^N \mid \exists \vec{x}' \in \mathbb{Z}^N : \pi \vdash \vec{x} \xrightarrow{*} \vec{x}' \right\}$$

and the range as

$$R(\pi) := \left\{ \vec{x}' \in \mathbb{Z}^N \mid \exists \vec{x} \in \mathbb{Z}^N : \pi \vdash \vec{x} \xrightarrow{*} \vec{x}' \right\} = \pi(D(\pi))$$

It is clear that  $\pi : D(\pi) \rightarrow R(\pi)$  is well defined as a function, in that a unique  $\pi(\vec{x})$  exists for all  $\vec{x} \in D(\pi)$ .

For an infinite computation path  $\pi = (p_i)_{\mathbb{N}_0}$ , define the domain as

$$D(\pi) := \left\{ \vec{x} \in \mathbb{Z}^N \mid \forall k \in \mathbb{N}_0 \exists \vec{x}' \in \mathbb{Z}^N : \pi_k \vdash \vec{x} \xrightarrow{*} \vec{x}' \right\}$$

where  $\pi_k = p_0, \dots, p_k$  is the length- $k$  prefix of  $\pi$ .

**Definition 2.3** (Realizable). A (finite or infinite) computation path  $\pi$  is realizable if and only if  $D(\pi) \neq \emptyset$ , i.e. if there exists some store that will cause  $\pi$  to be taken.

**Definition 2.4.** An assertion is a relation  $\rho \subseteq \mathbb{Z}^N$ . We write “ $\rho(\vec{x})$  holds” or simply “ $\rho(\vec{x})$ ” when  $\vec{x} \in \rho$ . An assertion holds at program point  $p$  iff for any variable values  $\vec{x}_0, \vec{x}$

$$(p_0, \vec{x}_0) \xrightarrow{*} (p, \vec{x}) \text{ implies } \rho(\vec{x})$$

or, put differently, iff  $\text{Reach}(p) \subseteq \rho$ .

**Definition 2.5.** We call an inequality of the form  $f(\vec{x}) \geq 0$  with  $f : \mathbb{Z}^N \rightarrow \mathbb{Z}$  a functional inequality.

Note, that if a functional inequality  $f(\vec{x}) \geq 0$  holds at program point  $p$ , then the restriction of  $f$  to  $\text{Reach}(p)$  maps into the naturals, i.e.

$$f|_{\text{Reach}(p)} : \text{Reach}(p) \rightarrow \mathbb{N}_0 \quad (3)$$

**Definition 2.6** (decreasing). A finite computation path  $\pi : p \xrightarrow{\pm} p$  is called decreasing iff there exists a functional inequality  $f(\vec{x}) \geq 0$  holding at  $p$  such that

$$\forall \vec{x} \in D(\pi) : f(\vec{x}) > f(\pi\vec{x})$$

If the above is true for some given  $f$ , we say that  $\pi$  decreases  $f$ .

### 2.3 Termination

We now have in place the tools needed to formulate a general termination principle based on inequality assertions and size change. We follow the philosophy of the example in section 2.1. Put simply: Find expressions that are bounded from below, then show them decreasing and finally use this information to show that certain program points are never passed infinitely many times. If the flow-graph remaining after removing these points has no strongly connected components, the program terminates. In the opposite case, we have localized the pieces of the program that may cause non-termination: the remaining strongly connected components.

**Definition 2.7** (Safe). A program point  $p$  is called safe iff any realizable computation path visits  $p$  at most finitely many times.

**Theorem 2.8.** If there exists a finite set  $I(p)$  of functional inequalities all holding at program point  $p$ , such that any  $\pi : p \xrightarrow{\pm} p$  decreases at least one  $f \in I(p)$ , then  $p$  is safe.

*Proof.* Assume, for the sake of contradiction, that the condition holds at  $p$  and that  $\pi = (p_i)_{\mathbb{N}_0}$  is a realizable computation path that passes  $p$  an infinite number of times. Define correspondingly  $\pi_{ij}$  as the finite computation sub-segment of  $\pi$  going from the  $i$ 'th to the  $j$ 'th occurrence of  $p$  in  $\pi$ , i.e.:

$$\pi : p_0 \xrightarrow{*} p \xrightarrow{\pm} p \xrightarrow{\pm} p \xrightarrow{\pm} p \cdots$$

$$\underbrace{\qquad \qquad \qquad}_{\pi_{12}} \quad \underbrace{\qquad \qquad \qquad}_{\pi_{23}} \quad \underbrace{\qquad \qquad \qquad}_{\pi_{34}} \quad \underbrace{\qquad \qquad \qquad}_{\pi_{14}}$$

$$\underbrace{\qquad \qquad \qquad}_{\pi_{13}}$$

For each pair  $(i, j) \in \mathbb{N}^2$  with  $i < j$ , define a “color”

$$c(i, j) := \{f \in I(p) \mid \pi_{ij} \text{ decreases } f\}$$

As  $c(i, j) \subseteq I(p)$ , the cardinality  $|c(i, j)|$  is finite and bounded from above for all  $i, j$  by  $|I(p)|$ . Also, from the assumption,  $c(i, j) \neq \emptyset$ . Now, for each  $c \in \mathcal{P}(I(p))$ , define the class  $P_c$  as

$$P_c := \{(i, j) \in \mathbb{N}^2 \mid i < j \text{ and } c(i, j) = c\}$$

The set  $\{P_c \mid c \in \mathcal{P}(I(p))\}$  is obviously finite and the classes  $P_{c'}, P_c$  with  $c' \neq c$  are mutually disjoint. By Ramsey’s theorem, there is an infinite subset  $J \subseteq \mathbb{N}$  and a “color”  $c_0$  such that for any two  $i, j \in J$  with  $i < j$ , we have  $c(i, j) = c_0$ .

Enumerate  $J$  in ascending order as  $\{j_1, j_2, \dots\}$ . Then, specifically,  $c(j_i, j_{i+1}) = c_0$  for all  $i \in \mathbb{N}$ . Set  $\pi_i$  to be the finite computation path from  $p_0$  to the  $j_i$ ’th occurrence of  $p$  in  $\pi$ :

$$\begin{array}{c} \pi : p_0 \xrightarrow{*} p \xrightarrow{*} p \xrightarrow{\pm} p \xrightarrow{\pm} p \xrightarrow{\pm} \dots \\ \underbrace{\hspace{10em}}_{\pi_1} \\ \underbrace{\hspace{10em}}_{\pi_2} \\ \underbrace{\hspace{10em}}_{\pi_3} \end{array}$$

Choose any  $f \in c_0$ . Certainly for any  $i, j \in \mathbb{N}$  with  $i < j$ , the computation path  $\pi_{j_i j_j}$  decreases  $f$ . We now have for any  $x_0 \in D(\pi)$ :

1.  $f(\pi_1 \vec{x}_0) > f(\pi_{j_1 j_2}(\pi_1 \vec{x}_0)) = f(\pi_2 \vec{x}_0)$
2. For  $n \geq 2$ :  $f(\pi_n \vec{x}_0) > f(\pi_{j_{n+1} j_{n+2}}(\pi_n \vec{x}_0)) = f(\pi_{n+1} \vec{x}_0)$

And so the sequence  $(f(\pi_k \vec{x}_0))_{k \in \mathbb{N}}$  is a strictly decreasing sequence. But by the assumption, we have  $f(\text{Reach}(p)) \subseteq \mathbb{N}_0$ , and for all  $k$  we have  $\pi_k \vec{x}_0 \in \text{Reach}(p)$ . Since there exist no infinitely decreasing sequences in  $\mathbb{N}_0$ , our assumption that  $\pi$  visits  $p$  an infinite number of times is false; there can exist such computation path.  $\square$

**Definition 2.9** (Safety of cycles and subgraphs). *We say that a cycle  $\gamma$  in the programs flow-graph  $F$ , i.e. a closed path visiting no program point more than once, is safe iff it contains a program point  $p$  that is safe. We define*

$$\Delta(F) := \{\gamma \text{ cycle in } F \mid \gamma \text{ is safe}\}$$

*We say that a sub-graph  $F'$  of the flow-graph is safe if each cycle  $\gamma$  in  $F'$  is safe, i.e. contains at least one safe program point  $p$ . Equivalently: it is safe if the graph*

$$F' \setminus \Delta(F')$$

*remaining after removing all safe program points from  $F'$  contains no strongly connected components. In general we’ll call this remaining graph the residual of  $F'$ .*

**Theorem 2.10.** *If a sub-graph  $F'$  of the flow-graph  $F$  is safe, then there exist no infinite realizable computation path lying entirely within  $F'$ .*

*Proof.* Assume that  $F'$  is safe and that such an infinite computation path  $\pi$  exists, consisting only of program points within  $F'$ . Since the number of cycles in  $F'$  is finite, there must exist a cycle  $\gamma \subseteq F'$  such that  $\gamma$  appears infinitely many times in  $\pi$ . But since  $F'$  is safe,  $\gamma$  contains a safe program point  $p$ . Since  $p$  cannot appear infinitely many times in  $\pi$ , certainly neither can  $\gamma$ . Thus there can exist no infinite realizable computation path in  $F'$ .  $\square$

**Corollary 2.11.** *If the entire flow-graph  $F$  of a program  $P$  is safe, in the sense of def. 2.9, then  $P$  terminates.*

*Proof.* Any infinite computation on  $P$  must correspond to some infinite computation path  $\pi$  in  $F$ . If  $F$  is safe, then per Theorem 2.10 there exists no such computation paths within  $F$ . Thus  $P$  must terminate on all input.  $\square$

Corollary 2.11 is of considerable interest when dealing directly with program termination. When using the size-change principle as an aid in performing e.g. bounded anchoring analysis (as described by Glenstrup and Jones in [GJ03]), or in other program analyses, we will in general want theorem 2.10.

In the next section, we learn how one can discover functional inequalities – the first requirement in theorem 2.8.

## 2.4 Invariant Relations Analysis

Even when a program has no explicit interdependence among variables, one may still find correlations and relations among them that hold for all the possible values they may assume.

Consider figure 2, which illustrates the possible stores for some program with two integer variables,  $x$  and  $y$ . Any particular program store corresponds to a point in  $\mathbb{Z}^2$ , and the reachable stores that the program can assume at some program point  $p$  is a subset  $Reach(p) \subseteq \mathbb{Z}^2$ . The nature of this set may be extremely complicated, yet we may still be successful in extracting from it neat relations among the variables that hold for all elements of  $Reach(p)$ . The aim is to find

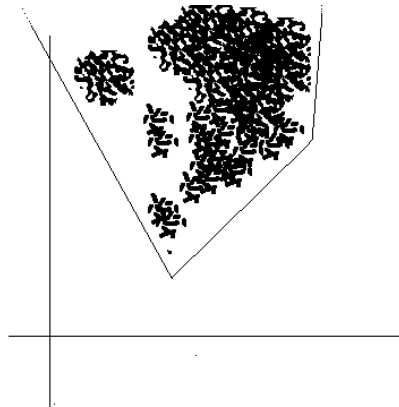


Figure 2: Linear over-approximation to possible program stores.

some approximation that is easily and efficiently expressed and manipulated, and that is safe in the sense that any reachable program store is always contained in the approximation.

In their 78 paper [CH78], Halbwachs and Cousot introduced a very powerful framework for approximating state. The method accomplishes automatic discovery of *invariant linear relations* among program variables by abstractly interpreting programs over the domain of convex polyhedra. For each of the subject language's constructs, they approximated its effect on the convex polyhedron enclosing the program store. The method is able to discover invariants that are not explicit in the program text, and that are often non-obvious. The following small example is given as an appetizer:

```

int subxy(int x, int y)
{
  int z, i;
  z = 0;
  i = x;
  if(y<=0 || x<= 0) return 0;

  while(i>0){
    i--;
    z++;
  }
  while(i<y){
    i++;
    z--;
  }
  return z;
}

int subxy(int x, int y)
{
  int z;
  int i;
  L1: z = 0;
  i = x;
  if (y <= 0 || x <= 0) goto L2; else goto L3;
  L3: if (i > 0) goto L4; else goto L5;
  L5: if (i < y) goto L6; else goto L7;
  L7: return z;
  L6: i = i + 1;
  z = z - 1;
  goto L5;
  L4: i = i - 1;
  z = z + 1;
  goto L3;
  L2: return 0;
}

```

Table 1: Small function that computes  $x - y$  (for  $x, y > 0$ ) in a roundabout way.

Table 1 shows a small function that computes and returns  $x - y$  if  $x, y > 0$  – however, there is no direct flow to  $z$  from  $x$  or from  $y$ . Figure 3 is output by our implementation of [CH78], building on [ea05], when run on `subxy.c`. As can be seen by inspecting the annotations of the arc to the return statement in block 7, we still find that  $z = x - y$  when the function returns – along with quite a few other linear relations that are not easily discovered by hand.

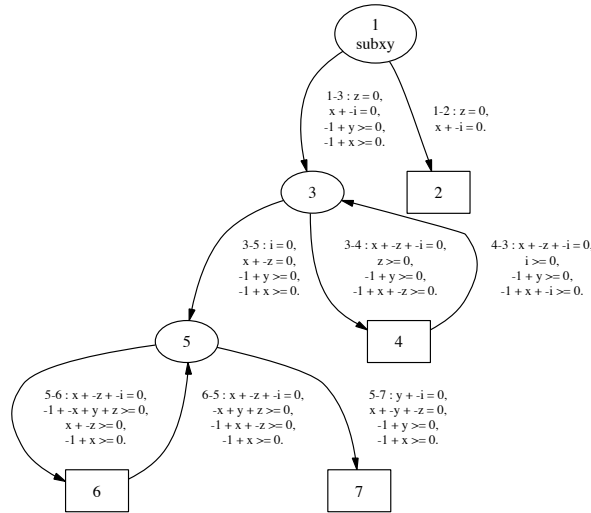


Figure 3: Linear relations discovered by abstractly interpreting `subxy.c` over the domain of convex polyhedra. Note that non-obvious relations are discovered. Specifically, at the return in block 7, we find that  $z = x - y$ , although  $x$  and  $y$  have no direct flow to  $z$ .

### 2.4.1 Convex polyhedra

A convex polyhedron over the integers of dimension  $n$  is a subset  $P \subseteq \mathbb{Z}^n$  that is the intersection of a finite number of affine half-spaces of  $\mathbb{Z}^n$ . This corresponds to a finite conjunction of linear inequalities, where a strict inequality defines an open half-space and a non-strict inequality de-

defines a closed half-space. We will write  $\mathbb{P}_n$  for the set of convex polyhedra with dimension  $n$ . Two important features are worth noticing:

- The **convex hull** of  $p$  and  $q$  is the smallest convex polyhedron  $r$  such that  $r \supseteq p \cup q$ . It can be shown that this exists and is unique given  $p$  and  $q$ . When equipped with set inclusion as its partial order, intersection as the *meet* operation and *convex hull* as its *join*,  $\mathbb{P}_n$  forms a lattice with  $\emptyset$  and  $\mathbb{Z}^n$  as its smallest respectively largest element. This important property allows us to look at increasing sequences  $(p_k)_{\mathbb{N}} \subseteq \mathbb{P}_n$  of polyhedra in which all the constraints of an element  $p_k$  of the sequence are satisfied by its successor  $p_{k+1}$ , and know that they will have a least upper bound.
- The second important virtue of  $\mathbb{P}_n$  is the existence of well-defined **widening-operators**  $\nabla : \mathbb{P}_n \times \mathbb{P}_n \rightarrow \mathbb{P}_n$ , defined on any  $p, q \in \mathbb{P}_n$  such that  $p \subseteq q$ . A widening-operator by definition satisfies two properties
  1.  $p \subseteq p \nabla q$  and  $q \subseteq p \nabla q$ . (i.e.  $\text{Hull}(p, q) \subseteq p \nabla q$ ).
  2. For any increasing chain  $q_0 \subseteq q_1 \subseteq q_2 \subseteq \dots$ , the increasing chain  $p_0 \subseteq p_1 \subseteq p_2 \dots$  defined by  $p_0 := q_0$  and  $p_{k+1} := p_k \nabla q_{k+1}$  has at most a finite number of strict increases.

The second property ensures that any sequence of widening operations converges to an upper bound in a finite number of steps. Specifically, although a sequence of increasingly lax constraints may give rise to an infinite strictly increasing sequence  $(q_k)_{\mathbb{N}}$  of polyhedra, we can utilize the widening operator to find a finite (in the sense that it reaches its upper bound) sequence  $(p_k)_{\mathbb{N}}$  such that  $q_k \subseteq p_k$ . In particular, the maximal element of the widened sequence contains the upper bound of  $(q_k)_{\mathbb{N}}$ . This allows us to iteratively propagate linear invariants throughout the program, finding in a finite number of steps a conjunction of linear equations for each program point, that are known to always hold at that point.

The method of abstract interpretation over the lattice of convex polyhedra is well described in the literature; our own programs are a straightforward implementation of [CH78], using the *Parma Polyhedral Library* [ea05] by Bagnara et al. for widening and basic operations on polyhedra.

## 2.5 Size-change Graphs

**Definition 2.12.** Let  $V = \{x_1, \dots, x_N\}$  be the set of program variables. In the following, a size-change graph is a pair  $g = (g^\downarrow, g^\uparrow)$  of bipartite graphs from  $V$  to  $V$  with labeled arcs:

$$\begin{aligned} g^\downarrow &\subseteq V \times \{\downarrow, \bar{\downarrow}\} \times V \\ g^\uparrow &\subseteq V \times \{\uparrow, \underline{\uparrow}\} \times V \end{aligned} \quad (4)$$

where  $[x_i \xrightarrow{\downarrow} x_j]$  and  $[x_i \xrightarrow{\bar{\downarrow}} x_j]$  or  $[x_i \xrightarrow{\uparrow} x_j]$  and  $[x_i \xrightarrow{\underline{\uparrow}} x_j]$  is not contained in the same graph.

A size-change graph is used to capture definite information about a finite computation path, as is apparent from the following definition:

**Definition 2.13** (Approximation of finite computation paths). A size-change graph  $g = (g^\downarrow, g^\uparrow)$  is said to approximate  $\pi$  iff for each edge  $[x_i \xrightarrow{\downarrow} x_j] \in g^\downarrow$  we have  $\pi \vdash \vec{x} \xrightarrow{*} \vec{x}'$  implies  $x'_j < x_i$ , for each edge  $[x_i \xrightarrow{\bar{\downarrow}} x_j] \in g^\downarrow$  we have  $\pi \vdash \vec{x} \xrightarrow{*} \vec{x}'$  implies  $x'_j \leq x_i$ , and the analogous statements hold true for  $g^\uparrow$ .

I.e., the graphs are “must-decrease” and “must-increase” graphs.

**Definition 2.14** (SCG Composition). Let  $g : p \rightarrow p'$  and  $h : p' \rightarrow p''$  be size-change graphs. The composite  $q = g; h$  is the size-change graph  $q : p \rightarrow p''$  with edges:

1.  $[x_i \xrightarrow{\downarrow} x_j] \in q^\downarrow$  if and only if  $\exists k : [x_i \xrightarrow{\delta_1} x_k] \in g^\downarrow$  and  $[x_k \xrightarrow{\delta_2} x_j] \in h^\downarrow$  with  $\delta_1, \delta_2 \in \{\bar{\downarrow}, \downarrow\}$  and  $\delta_1 = \downarrow$  or  $\delta_2 = \downarrow$ .

2.  $[x_i \overset{\bar{\tau}}{\rightarrow} x_j] \in q^\downarrow$  if and only if 1. does not apply, and  $\exists k : [x_i \overset{\bar{\tau}}{\rightarrow} x_k] \in g^\downarrow$  and  $[x_k \overset{\bar{\tau}}{\rightarrow} x_j] \in h^\downarrow$ .

and such that the analogous statements hold for  $q^\uparrow$ .

It is easy to show by transitivity of  $<$ ,  $\leq$ ,  $>$  and  $\geq$ , that if  $\pi : p \overset{\bar{\tau}}{\rightarrow} p'$  and  $\tau : p' \overset{\bar{\tau}}{\rightarrow} p''$  are finite computation paths approximated by  $g$  and  $h$  respectively, then the composite graph  $g;h$  approximates the finite computation path  $\pi\tau : p \overset{\bar{\tau}}{\rightarrow} p''$ .

**Definition 2.15** (Transitive closure). *Let  $G$  be a set of size-change graphs. The transitive closure  $\mathcal{S}(G)$  is the smallest set  $\mathcal{S}$  that fulfills*

$$G \subseteq \mathcal{S} \text{ and } \{g;h \mid g : p \rightarrow p' \in \mathcal{S} \wedge h : p' \rightarrow p'' \in \mathcal{S}\} \subseteq \mathcal{S} \quad (5)$$

This set exists and is finite, because the number of different size-change graphs over  $N$  variables is finite.

The transitive closure of a program's size-change graphs summarizes path effects. Every realizable finite computation path corresponds to a graph in the closure, giving us a *finite* description of all computation paths. It is computable by standard methods, e.g. as described in [AHU75].

Writing  $A^2 := \{0, \bar{\downarrow}, \downarrow\} \times \{0, \bar{\uparrow}, \uparrow\}$  (with  $A$  standing for "arrow"), we may express a size-change graph  $g = (g^\downarrow, g^\uparrow)$  as a map  $\mathbb{V} \times \mathbb{V} \rightarrow A^2$ :

**Definition 2.16** (SCGs as maps). *Let  $g = (g^\downarrow, g^\uparrow)$ , and define*

$$g(x_i, x_j) := (g^\downarrow(x_i, x_j), g^\uparrow(x_i, x_j))$$

$$g^\downarrow(x_i, x_j) := \begin{cases} \downarrow & \text{if } [x_i \overset{\bar{\downarrow}}{\rightarrow} x_j] \in g^\downarrow \\ \bar{\downarrow} & \text{if } [x_i \overset{\bar{\tau}}{\rightarrow} x_j] \in g^\downarrow \\ 0 & \text{otherwise} \end{cases} \quad g^\uparrow(x_i, x_j) := \begin{cases} \uparrow & \text{if } [x_i \overset{\bar{\uparrow}}{\rightarrow} x_j] \in g^\uparrow \\ \bar{\uparrow} & \text{if } [x_i \overset{\bar{\tau}}{\rightarrow} x_j] \in g^\uparrow \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

The map  $g : \mathbb{V} \times \mathbb{V} \rightarrow A^2$  is thus well-defined and corresponds to exactly one size-change graph. As a shorthand, we will write  $g(\mathbf{x}) := g(\mathbf{x}, \mathbf{x})$  and  $g_{ij} := g(x_i, x_j)$ .

The notation  $g_{ij} := g(x_i, x_j)$  is useful as more than just short-hand. One may define an algebra of  $N \times N$  matrices over  $A^2$ , such that the graphs and matrices are equivalent, and composition is matrix multiplication. Many of the operations we perform on size-change graphs are very simply expressed and efficiently computed in this representation. The details are left out here for the sake of brevity.

## 2.6 Size-change Termination with Polyhedra and Idempotent SCGs

We are now almost ready to introduce an extended size-change termination principle based on theorem 2.8, but allowing for straightforward computation using size-change graphs and convex polyhedra. The idea is to, for each program point  $p$ , find a polyhedron  $P \supseteq \text{Reach}(p)$  using the method of section 2.4, and then by way of size-change analysis show that a computation visiting  $p$  infinitely many times must leave  $P$  and therefore also  $\text{Reach}(p)$  – thus proving that  $p$  is visited at most finitely many times.

### 2.6.1 A note on the representation of convex polyhedra

It is customary to compute with a dual representation of convex polyhedra, since some operations are best suited for one representation, and some more efficiently implemented with the other. The first representation is a conjunction of constraints, each of the form

$$c_0 + \sum_{k=1}^n c_k x_k \bowtie 0 \quad (7)$$

where  $\bowtie$  is one of  $\{=, \geq, >\}$ . The other is a set of *points, closure points, rays* and *lines*. These are called, under one, *generators* of  $P$ , and a set full enough to describe  $P$  is called a *generating system*



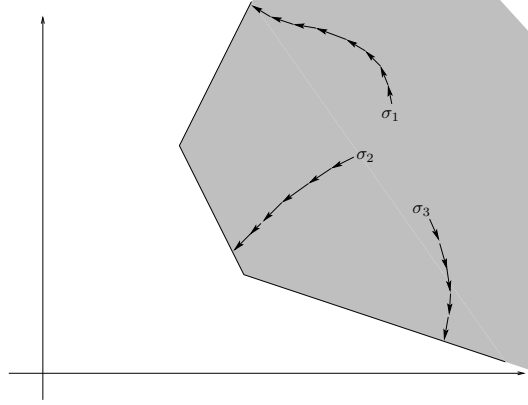


Figure 4: General strategy: Given a polyhedron  $P$  enclosing all reachable stores, show that any infinite computation must at some point leave  $P$ .

for  $P$ . Informally, the points of a minimal generating system used to represent a polyhedron are its vertices, and rays and lines describe its unbounded parts. Most operations needed for abstract interpretation in the style of [CH78] requires the generator representation. For the size-change termination analysis, we'll want the constraint-form. Specifically, we will assume that a polyhedron is of the form<sup>1</sup>

$$P = \left\{ \vec{x} \in \mathbb{Z}^N \mid f_1(\vec{x}) \geq 0 \wedge \dots \wedge f_m(\vec{x}) \geq 0 \right\} \quad (8)$$

(with each  $f_i(\vec{x}) = c_0^i + \sum_{k=1}^N c_k^i x_k$ ) and we also write, abusing notation slightly,

$$P = \{f_1, \dots, f_m\} \quad (9)$$

### 2.6.2 The effect of SCGs on linear expressions

**Definition 2.17** (Effect on linear expressions). Let  $g$  be a size-change graph and  $f(\vec{x}) = a_0 + \sum_{k=1}^N a_k x_k$ . Then we say that  $g$  decreases  $f$  iff the following hold

1.  $\forall k : \begin{array}{ll} a_k > 0 & \text{implies } g_{kk}^\downarrow = \overline{\downarrow} \text{ or } g_{kk}^\downarrow = \downarrow \\ a_k < 0 & \text{implies } g_{kk}^\uparrow = \underline{\uparrow} \text{ or } g_{kk}^\uparrow = \uparrow \end{array}$
  2.  $\exists k : \begin{array}{ll} a_k > 0 & \text{and } g_{kk}^\downarrow = \downarrow \text{ or} \\ a_k < 0 & \text{and } g_{kk}^\uparrow = \uparrow \end{array}$
- (10)

**Proposition 2.18.** Let  $\pi : p \dashrightarrow p'$  be a finite computation path and  $g$  a size-change graph approximating  $\pi$ . If  $g$  decreases  $f$  as in def. 2.17, then  $\pi$  also decreases  $f$  as in def. 2.6. In other words, “ $g$  decreases  $f$ ” implies

$$f(\pi\vec{x}) < f(\vec{x})$$

for any  $\vec{x} \in D(\pi)$ .

*Proof.* Let there be given  $f(\vec{x}) = a_0 + \sum_{k=1}^N a_k x_k$ , finite computation path  $\pi$  and size-change graph  $g$  approximating  $\pi$ . If  $g$  decreases  $f$  as in def. 2.17, then

- no term  $a_k x_k$  is ever increased by  $\pi$ , and
- at least one term is decreased by  $\pi$ .

Consequently,  $f$  as a whole is decreased by  $\pi$ . □

<sup>1</sup>We choose to restrict ourselves to closed polyhedra. We can do this without loss of generality, since on  $\mathbb{Z}^N$  the open and closed affine half spaces are the same. For example,  $x > 0 \iff x - 1 \geq 0$  on  $\mathbb{Z}$ .

### 2.6.3 Termination: Idempotent Graphs

Let in the following  $I(p) := \{f_1, \dots, f_m\}$  be a finite set of linear expressions such that for any  $\vec{x} \in \text{Reach}(p)$ ,

$$f_1(\vec{x}) \geq 0 \wedge \dots \wedge f_m(\vec{x}) \geq 0 \quad (11)$$

**Theorem 2.19.** *If each  $g : p \rightarrow p \in \mathcal{S}$  with  $g = g; g$  decreases some  $f \in I(p)$ , then  $p$  is safe.*

*Proof.* The theorem is proved very similarly to theorem 2.8. Assume the left hand side of the theorem to be true. Let  $\pi = (p_i)_{\mathbb{N}}$  be a realizable computation path passing  $p$  infinitely many times, and enumerate the occurrences of  $p$  in  $\pi$  as  $\alpha_i$ , i.e.  $\forall i \in \mathbb{N} : p_{\alpha_i} = p$ . Define  $g_{ij} := g_{p_i; \dots; p_{j-1}}$ , and let the class  $P_g$  be defined as

$$P_g := \left\{ (i, j) \in \mathbb{N}^2 \mid i < j \text{ and } g = g_{\alpha_i \alpha_j} \right\} \quad (12)$$

The set  $\{P_g \mid g : p \rightarrow p \in \mathcal{S}\}$  is finite (since  $\mathcal{S}$  is finite) and the classes are mutually disjoint. By Ramsey's theorem it then follows that there exists an infinite subset  $J = (j_i)_{\mathbb{N}}$  of  $\mathbb{N}$  and a size-change graph  $g_0 : p \rightarrow p \in \mathcal{S}$  such that for any two  $i < j$  in  $J$ ,  $g_{\alpha_i \alpha_j} = g_0$ .

If we now take  $i < j < k$  from  $J$ , we get

$$g_0 = g_{\alpha_i \alpha_k} = g_{\alpha_i \alpha_j}; g_{\alpha_j \alpha_k} = g_0; g_0 \quad (13)$$

Because of the assumption,  $g_0$  decreases some  $f \in I(p)$ . Define (similar to 2.8):

$$\begin{array}{c} \pi : p_0 \xrightarrow{*} p \xrightarrow{*} p \xrightarrow{\overbrace{\pm}^{\pi_{\alpha_1 \alpha_2}}} p \xrightarrow{\overbrace{\pm}^{\pi_{\alpha_2 \alpha_3}}} p \xrightarrow{\pm} \dots \\ \underbrace{\hspace{10em}}_{\pi_1} \\ \underbrace{\hspace{10em}}_{\pi_2} \\ \underbrace{\hspace{10em}}_{\pi_3} \end{array}$$

Then for any initial store  $\vec{x}_0$ , the sequence  $(f(\pi_k \vec{x}_0))_{k \in \mathbb{N}}$  is strictly decreasing. But because  $f \in I(p)$ , and  $\pi_k \vec{x}_0 \in \text{Reach}(p)$ , each  $f(\pi_k \vec{x}_0) \geq 0$ . Therefore the assumption that  $\pi$  passes  $p$  infinitely many times must be false. Consequently,  $p$  must be safe.  $\square$

## 3 Algorithm

The algorithm is given in much greater detail - both in pseudo-code and actual working code - in [AS04]. Here we'll be content with outlining the steps:

1. Find invariants using abstract interpretation in the style of Cousot and Halbwachs, resulting in a polyhedron  $\mathcal{P}[p]$  for each program point  $p$ , such that  $\mathcal{P}[p] \supseteq \text{Reach}(p)$ .
2. Generate size-change graphs for each program point  $p$  as  $\mathcal{G}[p]$ .
3. Compute the transitive closure  $\mathcal{S}$  of  $\mathcal{G}$ .
4. For each program point  $p$ , let  $\mathcal{I}[p] \subseteq \mathcal{S}$  be the idempotent size-change graphs  $g : p \rightarrow p$  in the closure  $\mathcal{S}$ .
5. Construct residual flow-graph  $F'$  containing the program points  $p$  that are not safe:
  - (a) For each  $g \in \mathcal{I}[p]$ :
    - Let  $\mathcal{P}[p] = \{f_1(\vec{x}) \geq 0 \wedge \dots \wedge f_m(\vec{x}) \geq 0\}$ .
    - If  $g$  decreases  $f_k$  (def. 2.17) for at least one  $1 \leq k \leq m$ , then  $p$  is safe.
6. If  $F'$  contains no strongly connected components, then  $P$  terminates. If  $F'$  does contain strongly connected components, then  $P$  may not terminate, and the strongly connected components are the cyclic parts which risk looping forever.

## 4 Concluding Remarks

We present an extension of the size-change termination principle of [LJB01] to programs with integer valued data types. The extended size-change termination principle is realized through a combination of polyhedral bounds on program state, discovered by abstract interpretation, with size-change analysis.

The methods have been implemented in the C specializer `C-MixIT` for a subset of its internal language `Core-C`. This subset corresponds to C programs without function calls, pointer aliasing, flat memory and dynamic allocation. It is planned to extend the implementation to handle a larger subset of C.

### 4.1 Related work

Some work by Henny Sipma and co-workers, as well as recent work by Siau-Cheng Khoo and Hugh Anderson, is very close in spirit to the methods presented here. Both their strategies for determining termination involve finding decreases in linear expressions that are bounded from below.

**Sipma and Colon ([SC02])** perform automatic discovery of bounded and decreasing linear expressions, leading to termination. This is accomplished through iterative forward propagation of invariants in the form of polyhedral cones. Rather than describing size-change by a finite approximation (as is the case with size-change graphs), they rely on widening operations to give results in a finite number of steps.

**Khoo and Anderson ([KA05])** extend the notion of size-change graphs to *affine graphs*, which are conjunctions of linear interparameter inequalities. Graphs are augmented by *guards*, which are bounds on program state. Then it is shown that infinite compositions of the affine graphs will violate one of these guards. Termination of the analysis is ensured by reducing affine graphs to abstract graphs, which are basically size-change graphs.

### 4.2 Ideas for Future Work

- Work is being done on a less precise version of the size-change analysis which runs in polynomial time. In [Lee02] and [BAL04], C.S. Lee and Amir Ben-Amram investigate a quadratic worst-time approximation to SCT. Correspondence with Lee and Bem-Amram has indicated that are adaptable with few modifications to the extended size-change termination principle presented here.
- It should be fairly easy to extend the theorems presented here to allow constraints to be disjunctions of conjunctions of linear inequalities. Constraints of this form can be discovered by using e.g. Pressburger arithmetic or interpreting over the powerset  $\mathcal{P}(\mathbb{P}_N)$  of polyhedra.
- Another way of determining size-change is by interpretation over the domain of polyhedral cones, which are  $2N$ -dimensional polyhedra linearly relating state before and after execution of a computation segment. This is the method being used for termination analysis in [SC02]. It would be interesting to compare the classes of programs handled by this method to those handled by the extended SCT. The differences are subtle: In one respect, polyhedral cones capture more information than size-change graphs, because linear intervariable relations are taken into account. In another respect, they capture less: where two distinct paths can give rise to two distinct size-change graphs, the polyhedral method must yield a single convex hull large enough to contain both.
- Many programs do not terminate on all input, but rely on implicit assumptions about the form of input being passed to them. An idea was recently suggested by Dr. W.N. Chin that would allow extending the presented method to find safe *preconditions* on input for which a program size-change terminates.

## References

- [AHU75] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1975.
- [AS04] James Avery and Stefan Schou. Stopping C-Mix: PE-termination ensuring binding-time division. Project at Roskilde Universitetscenter, 2004.
- [Ave05] James Avery. Size-change termination and bound analysis for an imperative language with integer data types. DIKU Report, July 2005.
- [BAL04] Amir Ben-Amram and Chin Soo Lee. A quadratic-time program termination analysis. Under preparation at MPI für Informatik, Saarbrücken, Germany, 2004.
- [CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
- [ea05] Roberto Bagnara et al. The parma polyhedra library, 2000-2005. University of Parma, <http://www.cs.unipr.it/ppl/>.
- [GJ03] Arne J. Glenstrup and Neil D. Jones. Termination analysis and specialization-point insertion in off-line partial evaluation. In *ACM Transactions on Programming Languages and Systems*. Department of Computer Science, University of Copenhagen, 2003. Preprint.
- [KA05] Sian-Cheng Khoo and Hugh Anderson. Bounded size-change termination (draft). School of Computing, National University of Singapore, 2005.
- [Lee02] Chin Soon Lee. Program termination analysis in polynomial time. *Generative Programming and Component Engineering '02*, LNCS 2487, October 2002.
- [LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. BenAmram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.
- [SC02] Henny Sipma and Michael Colon. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification*, volume LNCS 2404, pages 442–454. Springer Verlag, 2002.

# Termination analysis by the size-change principle of programs in the untyped $\lambda$ -calculus with arbitrary input from a well-defined input-set

Nina Bohr, IT-University of Copenhagen  
joint work with Neil D. Jones, University of Copenhagen

ninab@itu.dk

In an earlier work we have developed a method to safely analyse if call-by-value evaluation of a single closed term in the untyped  $\lambda$ -calculus will terminate [1]. The termination analysis is done by abstract interpretation. We have since extended this method, so that we can analyse if a  $\lambda$ -term will always terminate when applied to any input from a well-defined input set of  $\lambda$ -terms. Both versions of termination analysis can be fully automated.

The *size-change* principle for program termination was originally developed for first-order functional programs whose parameter values have a well-founded size order [2]. The method from that paper was the inspiration for the termination analysis of the untyped  $\lambda$ -calculus, but some new ideas were also required:

i) To add rules for calls to the call by value evaluation inference rules for the  $\lambda$ -calculus. A state  $s$  calls another state  $s'$  if evaluation of  $s'$  is an immediate sub-goal in the proof-tree for evaluation of  $s$ . By this calls relation we can identify nontermination by the existence of an infinite call-sequence.

ii) An equivalent environment-based version of the  $\lambda$ -calculus is defined. A state  $e : \rho$  is an expression together with an environment, that binds each free variable of  $e$  to a value of the form  $v : \rho'$ . Substitution is replaced by a "lazy substitution" that just updates the environment. An important observation is then, that in an attempt to evaluate a closed expression in the empty environment  $e : []$ , it will be the case that all calls and evaluations will be to states of the form  $e' : \rho'$  where  $e'$  is a subexpression of  $e$ . This property makes it possible to use the set of subexpressions of  $e$  as a finite set of program control points in the approximation.

iii) In order to apply the size change principle, we need a well-founded order where we can see decrease in variable bindings  $[x \mapsto e : \rho]$  in environments. This is based on two order-relations on states. We have  $e : \rho \succ e' : \rho'$  if at some level in  $\rho$  a variable is bound to  $e' : \rho'$ , also  $e : \rho \succ e' : \rho'$  if  $e'$  is a proper subexpression of  $e$  and the environments  $\rho$  and  $\rho'$  are identical for the free variables of  $e'$ . It is not possible infinitely to descend to a sub-environment without any increases of the environments in between. When a variable is evaluated  $x : [x \rightarrow e_x : \rho_x] \Downarrow e_x : \rho_x$  we find decreases  $x \succ y$  for all free variables  $y$  in  $e_x$ .

iv) We can make a finite approximation by removing all environment components. This means that we cannot look up a value for a variable in the environment, and we have to modify the variable-evaluation rule. This modification makes evaluation in the approximation nondeterministic. It holds that if  $e$  is a closed expression and there is a call-sequence  $e : [] \rightarrow^* e_1 : \rho_1$  and  $e_1 : \rho_1 \rightarrow e_2 : \rho_2, G$  where  $G$  is the associated size change graph, then in the approximation  $e \rightarrow^* e_1$  and  $e_1 \rightarrow e_2, G$ . It is still possible to identify places for size-decreases. We can base the termination analysis on the set of size change graphs for calls generated by abstract interpretation.

We often want to certify that a program will terminate when applied to any of its intended inputs. The termination analysis of [1] is only concerned with one single  $\lambda$ -term (which can represent for instance the Ackermann-function applied to 2 and 3). We have extended the  $\lambda$ -calculus to include expressions where nonterminals from a tree-grammar may take the place of some subexpressions. Such a nonterminal represents all the *pure*  $\lambda$ -terms, it can produce by the grammar in one or more steps. For instance a nonterminal may represent an arbitrary church numeral, and then a term in the extended  $\lambda$ -calculus may represent the Ackermann-function applied two arbitrary church

numerals. The termination analysis requires a careful definition of free variables and subexpressions for nonterminals, such that we can generate size change graphs which are safe for each representable pure term, even though the pure term may have fewer free variables than the extended term by which it is represented. The inference rules for calls and evaluations are extended to also handle nonterminals. A nonterminal will call each of the extended  $\lambda$ -expressions it can be rewritten to in one step by a production in the grammar, this means that an extended expression may evaluate to more than one value. If  $P$  is a closed expression in the extended language and  $Q$  is a closed expression in the pure  $\lambda$ -calculus, which is one of the expressions that  $P$  represents, then the call-graph for  $P : []$  includes a subgraph which can simulate the call-graph for  $Q : []$ . The generated size change graphs will not always be identical, but the size change graphs in the extended language can never give rise to illegal termination certification.

We can then make a finite approximation of the extended  $\lambda$ -calculus by removing all environment components. As when we approximated the pure  $\lambda$ -calculus, in this step we can rediscover all calls with identical size change graphs. If our analysis certifies that evaluation of a closed extended expression  $P$  "terminates", then this implies termination of all pure  $\lambda$ -expressions, that  $P$  represents.

## References

- [1] Neil D. Jones and Nina Bohr. Termination Analysis of the Untyped  $\lambda$ -Calculus. In *Rewriting Techniques and Applications: RTA 2004*. Lecture Notes in Computer Science. Springer. June, 2004.
- [2] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram. The Size-Change Principle for Program Termination. *POPL 2001: Proceedings 28<sup>th</sup> ACM SIGPLAN-SIGACT Principles of Programming Languages*

# IO Swapping Leads You There And Back Again

## (Extended Abstract) \*

Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics, University of Tokyo  
{Akimasa\_Morihata,kaz,hu,takeichi}@mist.i.u-tokyo.ac.jp

## 1 Introduction

TABA (“There And Back Again”) [DG02], proposed by Danvy and Goldberg, is a special but powerful programming pattern where a recursive function traverses lists at return time. Their idea is that the recursive calls get us there (typically to a empty list) and the returns get us back again while traversing the list. A typical example is the symbolic convolution function `cnv` which accepts two lists,  $[x_0, x_1, \dots, x_n]$  and  $[y_0, y_1, \dots, y_n]$ , and computes a new list  $[(x_0, y_n), (x_1, y_{n-1}), \dots, (x_n, y_0)]$ . This can be naively specified as follows.

```
cnv x y = zip x (reverse y)
```

This definition is not satisfactory; the list `y` is traversed by `reverse` to produce an intermediate list which will be again traversed by `zip`. A clever TABA solution, which avoids generation of the intermediate list, is as follows.

```
cnv x y = let ([],r) = walk x in r
           where walk [] = (y, [])
                 walk (a:x') = let (b:y',r) = walk x'
                               in (y', (a,b):r)
```

This program uses a bit unusual auxiliary function `walk`. When the input `x` is empty, `walk` uses the input `y` directly as a return value, and its return value will be traversed together while traversing `x`. Indeed this program is much different from the initial specification, but it actually computes symbolic convolution without the need of extra memory other than the resulting output.

TABA is truly tricky. It would be interesting to see whether there is a systematic way that may *lead us* to construct such TABA programs. One may wish to use and manipulate TABA-like computations for constructing a new kind of such iterations, i.e., iterations over some return values. In [DG02] Danvy and Goldberg gave a set of clever TABA programs, but neither derivation nor manipulation of them were presented sufficiently. In a recent paper [DG05], they

---

\* This is an extended abstract of the technical report [MKHT05]: A. Morihata, K. Kakehi, Z. Hu, and M. Takeichi. Reversing iterations: IO swapping leads you there and back again. *Technical Report METR 2005-11*, Department of Mathematical Informatics, University of Tokyo, May 2005. Available from <http://www.keisu.t.u-tokyo.ac.jp/Research/METR/2005/METR05-11.pdf>  
This was submitted to GPCE young researchers workshop 2005.

```

fst (a,b) = a
snd (a,b) = b
head [x0,x1,...,xn] = x0
tail [x0,x1,...,xn] = [x1,x2,...,xn]
reverse [x0,x1,...,xn] = [xn,xn-1,...,x0]
map f [x0,x1,...,xn] = [f x0,f x1,...,f xn]
zip [x0,x1,...,xn] [y0,y1,...,yn] = [(x0,y0),(x1,y1),..., (xn,yn)]
foldr f e [x0,x1,...,xn] = f x0 (f x1 (⋯ (f xn e)⋯))
foldl f e [x0,x1,...,xn] = f (⋯(f (f e x0) x1)⋯) xn

```

**Fig. 1.** Informal definitions of standard functions

showed how to derive TABA programs based on the two known transformations: CPS transformation and defunctionalization [DN01]. Though being systematic, the method is not constructive; that is to say, it is not incremental.

In this paper, we show a new and clear derivation of TABA programs by a novel program transformation rule called *IO swapping*. It swaps input and output values of a function, introduces iteration at return times and immediately derives TABA programs. We also demonstrate manipulations of TABA. We can incrementally construct bigger TABA programs from simpler TABA programs by program calculation [BdM96], a transformational approach to carrying programs from their naive definition to their efficient equivalent. The ability to manipulate TABA programs proves the effectiveness of program calculation.

Throughout the paper we use the notation of the functional programming language Haskell [Bir98]. The symbol  $\backslash$  is used instead of  $\lambda$  for  $\lambda$ -expressions. The symbol  $\cdot$  denotes function composition. We use many standard Haskell functions, whose informal definitions are given in Figure 1. We also assume that the size of structured data we are treating is finite.

## 2 Calculational Programming and IO Swapping

### 2.1 Calculational Programming

Functional programming languages provide a constructive way of programming, namely development of involved programs through composition of smaller and simpler functions. To improve efficiency of such compositional programming style, function fusion plays an important role, which fuses function composition into a single function and eliminates intermediate data structures passed between them. In this paper we will intensively use the following fusion (promotion) law [Bir89].

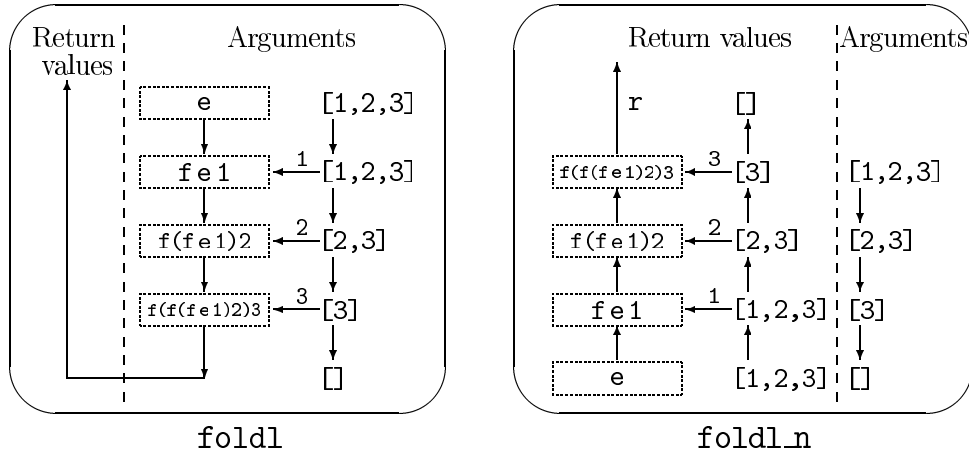
**Theorem 1 (Fold Fusion).**

$$f \cdot \text{foldr } (\oplus) e = \text{foldr } (\otimes) e'$$

provided that  $f e = e'$  and  $f (a \oplus y) = a \otimes (f y)$  hold for all  $a$  and  $y$ .  $\square$

This theorem indicates that finding a proper operator  $\otimes$  is enough for fusing programs. Such calculation over programs, which is often referred as *calculational*





**Fig. 2.** The models of computation processes of `foldl` and `foldl.n`

*programming* [BdM96] (or *program calculation*) is a powerful tool, as we later see TABA programs can be manipulated using calculational programming.

## 2.2 IO Swapping for `foldl`

The new and effective transformation rule proposed in this paper is *IO swapping*, which changes the view of functions: It “thinks upside down” about treatments of data structures through literally swapping the input and the output. Before going into the general framework, we explain the essence of the proposed transformation using a typical function `foldl` in the following theorem.

### Theorem 2 (IO Swapping for `foldl`).

The following two functions `foldl` and `foldl.n` are equivalent.

$$\begin{aligned}
 \text{foldl } f \ e \ [] &= e \\
 \text{foldl } f \ e \ (a:x) &= \text{foldl } f \ (f \ e \ a) \ x \\
 \\
 \text{foldl.n } f \ e \ x &= \text{let } ([], r) = \text{foldl}' \ x \ \text{in } r \\
 &\quad \text{where } \text{foldl}' \ [] = (x, e) \\
 &\quad \quad \text{foldl}' \ (b:y) = \text{let } ((a:x'), r') = \text{foldl}' \ y \\
 &\quad \quad \quad \text{in } (x', f \ r' \ a) \quad \square
 \end{aligned}$$

In `foldl'` the initial input list `x` of `foldl.n` is passed directly as the return value of the termination condition, and destructed in its recursive call. In short, this theorem achieves the transformation from `foldl` to its TABA form.

We make some remarks on the implications of this theorem. Pay attention to how the result is computed using the function parameter `f`. While `f` is applied to the accumulation parameter in the function `foldl`, it comes to the return value of `foldl.n`. This indicates the fact that IO swapping is a rule to swap the inputs (arguments) and the outputs (return values) of the original function. If the input list comes syntactically to the position as the output, consumption of lists in the

return value is a natural consequence. Figure 2 illustrates a computation process of `foldl` and `foldl.n`. Comparing two figures carefully, the idea of IO swapping becomes much more obvious: Turning over the the figure of `foldl` looks almost the same as that of `foldl.n`! It is possible to liken the input list as a tower where each floor stores a value. When the King, living at the top of this tower, commands servants to gather the values, some go downward from the top to the ground floor (like arguments) or others go upward from the ground floor to the top (like return values), as the phrase “`cdr` down, `cons` up” indicates. If the values in the tower are secretly rearranged upside-down, what these servants gather up are exchanged. Instead of such rearrangement, the equivalent effect can take place by transferring consumption of the list from the position of the argument to that of the return value: A return value arranges the values in the list, from the ground floor to the top, providing the reversed, upside-down order of values.

### 2.3 IO Swapping

It is possible to reinforce Theorem 2 so that it can deal with many more functions.

#### Theorem 3 (IO Swapping).

The following two functions `f1` and `f2` are equivalent.

$$\begin{aligned} \text{f1 } x \text{ h}_0 &= \text{let } r = \text{f1}' \text{ x } (g_3 \text{ r h}_0) \text{ in } r \\ &\quad \text{where } \text{f1}' \text{ [] h} = g_0 \text{ h} \\ &\quad \quad \text{f1}' \text{ (a:x')} \text{ h} = \text{let } r = \text{f1}' \text{ x'} (g_2 \text{ a r h}) \\ &\quad \quad \quad \text{in } g_1 \text{ a r h} \end{aligned}$$

$$\begin{aligned} \text{f2 } x \text{ h}_0 &= \text{let } ([], \text{ h}, \text{ r}') = \text{f2}' \text{ (x, g}_0 \text{ h)} \text{ in } r' \\ &\quad \text{where } \text{f2}' \text{ ([], r)} = (x, g_3 \text{ r h}_0, r) \\ &\quad \quad \text{f2}' \text{ (b:y, r)} = \text{let } (\text{a:x'}, \text{ h}, \text{ r}') = \text{f2}' \text{ (y, g}_1 \text{ a r h)} \\ &\quad \quad \quad \text{in } (\text{x'}, g_2 \text{ a r h}, \text{ r}') \end{aligned} \quad \square$$

Theorem 3 swaps the inputs and outputs of the auxiliary functions. In the definition of function `f1`, `g1` manages the computation of return value, but it manages that of accumulation parameter in `f2`. In contrast, `g2` manages the computation of the accumulation parameter in `f1` but it manages that of return value in `f2`. Applications of Theorem 3 are in [MKHT05].

## 3 Deriving TABA Programs by IO Swapping and Fusion

Now we show our derivation of TABA programs. We propose two methods: IO swapping and program calculation. The former directly derives a TABA program and the latter incrementally derives a bigger TABA program by promoting functions into a smaller TABA program.

### 3.1 List Reversal

We start by demonstrating a derivation of TABA-style `reverse` by the first method. Function `reverse` is defined by using `foldl` as follows.

```
reverse = foldl (\y a -> a:y) []
```

Applying Theorem 2 to `reverse`, we instantly get the following function `rev_n`, which is the TABA program for `reverse`.

```
rev_n x = let ([],r) = rev' x in r
  where rev' [] = (x,[])
        rev' (b:y) = let (a:x',r') = rev' y
                      in (x',a:r')
```

### 3.2 Symbolic Convolution

Next we show a systematic derivation of the TABA program for `cnv` in the introduction, starting from the following straightforward specification:

```
cnv x y = zip x (reverse y)
```

where we assume that `x` and `y` have the same length.

To derive TABA-style `cnv` we use the second method, program calculation, for we already get the TABA program for `reverse` namely `rev_n` in Section 3.1. We derive the TABA program for `cnv` by promoting `zip` into `rev_n`.

The function `rev_n` can be described in terms of `foldr` for being suitable for later fusion transformation.

```
rev_n x = snd (foldr (\b (a:x',r')->(x',a:r')) (x,[]) x)
```

Now we calculate TABA program for `cnv` by promoting the functions into `rev_n`.

```
cnv x y = zip x (rev_n y)
  => zip x (snd (foldr (\b (a:x',r)->(x',a:r)) (y,[]) y))
  => snd (id_zip (foldr (\b (a:x',r)->(x',a:r)) (y,[]) y) x)
  where id_zip (a,y) x = (a, zip x y)
```

To promote `id_zip` into `foldr` in the above, we check the following two conditions to apply the fusion law (Theorem 1).

```
id_zip (y,[]) x => (y,[])
id_zip ((\b (a:x',r)->(x',a:r)) b (a:x',r)) x
  => (x', (head x,a):zip (tail x) r)
  => step b (id_zip (a:x',r)) x
      where step b r' x = let (a:x', r) = r' (tail x)
                          in (x', (head x,a):r)
```

Therefore, the fusion transformation gives

```
cnv x y = snd (foldr step (\x->(y,[])) y x)
```

which is actually the following program after unfolding the `foldr`.

```
cnv x y = snd (cnv' y x)
  where cnv' [] = \x->(y,[])
        cnv' (b:y) = \x->let (a:x',r) = cnv' y (tail x)
                          in (x',(head x,a):r)
```

Finally, we make the program more concise with some known calculations. First,  $\eta$ -expansion to remove function values yields the following program, where the case `cnv' [] []` is obtained from the assumption that length of `x` and `y` are same.

```
cnv x y = let ([],r) = cnv' y x in r
          where cnv' [] [] = (y, [])
                cnv' (b:y) (d:z) = let (a:x',r) = cnv' y z
                                     in (x', (d,a):r)
```

Next we eliminate the unnecessary parameter: The first argument of `cnv'` is not used at all for producing results. It derives the efficient TABA program for `cnv`.

```
cnv x y = let ([],r) = cnv' x in r
          where cnv' [] = (y, [])
                cnv' (d:z) = let (a:x',r) = cnv' z
                                 in (x', (d,a):r)
```

### 3.3 List Reversal Revisited

As we have seen in successful derivation of `cnv`, program calculation works effectively if we have some TABA programs in hand. Making sure of it, we choose `rev_n`, which is obtained by IO swapping in Section 3.1, as a next example.

From the definition of `cnv`, we extract `rev_n` as follows.

```
rev_n x ⇒ map snd (zip x (reverse x))
        ⇒ map snd (cnv x x)
```

Starting from this equation, we obtain `rev_n` by fusing `map` with `cnv`.

```
rev_n x
  = map snd (cnv x x)
  ⇒ map snd (snd (foldr (\a (b:y',r)->(y',(a,b):r)) (x, []) x))
  ⇒ snd (id_map snd (foldr (\a (b:y',r)->(y',(a,b):r)) (x, []) x))
     where id_map f (a,b) = (a, map f b)
```

Now we apply Theorem 1 to fuse the above `id_map` with `foldr`. With checking the following conditions

```
id_map snd (x, []) ⇒ (x, [])
id_map snd ((\a (b:y',r)->(y',(a,b):r)) a (b:y',r))
  ⇒ (y', b:map snd r)
  ⇒ (\a (b:y',r)->(y',b:r)) a (id_map snd (b:y',r))
```

we get

```
rev_n x ⇒ snd (foldr(\a (b:y',r)->(y',b:r)) (x, []) x)
```

which is exactly `rev_n` in the form of `foldr`.

This process indicates that program calculation can be useful guidance for developing TABA programs. We can also derive more involved programs, such as the efficient palindrome detecting program. See [MKHT05].

## 4 Conclusion and Future Work

This paper presented a new approach to derive TABA programs systematically using a novel technique called IO swapping, which swaps the outputs and inputs of functions. Our approach confirms the competence of calculational programming for deriving efficient program from naive definition through these transformations of programs.

Our belief is that the effect of IO swapping is not limited to derivation of TABA. Investigation of some further applications still remains.

## Acknowledgement

We are very grateful to Olivier Danvy for introducing us the TABA work and its relation to defunctionalization and CPS transformation, and to Shin-Cheng Mu and Keisuke Nakano for their inspiring discussions at the laboratory seminars.

## References

- [BdM96] R. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.
- [Bir89] R. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.
- [Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, 1998.
- [DG02] O. Danvy and M. Goldberg. There and back again. In *Proc. of the 7th Int. Conf. on Functional programming*, pages 230–234, 2002.
- [DG05] O. Danvy and M. Goldberg. There and back again. Technical report, *BRICS Research Series RS-02-12*. Extended version of an article to appear in *Fundamenta Informaticae*, 2005.
- [DN01] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proc. of the 3rd Int. Conf. on Principles and practice of declarative programming*, pages 162–174, 2001.
- [MKHT05] A. Morihata, K. Takeichi, Z. Hu, and M. Takeichi. Reversing iterations: IO swapping leads you there and back again. *Technical Report METR 2005-11*, Department of Mathematical Informatics, University of Tokyo, 2005.

## On the Relations between Monadic Semantics

Andrzej Filinski  
DIKU, University of Copenhagen, Denmark  
andrzej@diku.dk

DIKU-IST Workshop on Foundations of Software  
24 September 2005

A. Filinski On the relations between monadic semantics 2005-09-24

---

### Background & motivation

Canonical reference for showing agreement of denotational semantics: [Reynolds 74: *On the relation between direct and continuation semantics*].

Not entirely undeserved reputation as a challenging paper.

#### Goals for the present work:

- Update [Reynolds 74] in light of more recent developments: computational monads [Moggi 89], invariant relations [Pitts 96]. Tame some of the complexity: *encapsulate* hard parts.
- Generalize to systematic treatment of agreement between *specification* and *implementation* semantics of languages with multiple effects. Ideas should apply to wide variety of paradigms: imperative, functional, logic, [concurrent?].

A. Filinski On the relations between monadic semantics 2005-09-24

---

### A simple, untyped functional language

#### Syntax:

$$t ::= \underline{n} \mid \text{succ} \mid x \mid \lambda^{\mathbf{n}}x.t \mid \lambda^{\mathbf{v}}x.t \mid t_1 t_2 \quad [n \in \mathbf{N}, x \in \mathbf{V}]$$

Natural-number constants intended for observation of final results only; must express proper computation using Church-coding, Y-combinator, etc.

#### Values:

$$v ::= \underline{n} \mid \text{succ} \mid \lambda^{\mathbf{n}}x.t \mid \lambda^{\mathbf{v}}x.t$$

**Big-step operational semantics:** for closed  $t$ , define  $t \Downarrow v$  by:

$$\frac{}{v \Downarrow v} \quad \frac{t_1 \Downarrow \text{succ} \quad t_2 \Downarrow \underline{n}}{t_1 t_2 \Downarrow \underline{n+1}} \quad \frac{t_1 \Downarrow \lambda^{\mathbf{n}}x.t'_1 \quad t'_1[t_2/x] \Downarrow v}{t_1 t_2 \Downarrow v} \\ \frac{t_1 \Downarrow \lambda^{\mathbf{v}}x.t'_1 \quad t_2 \Downarrow v_2 \quad t'_1[v_2/x] \Downarrow v}{t_1 t_2 \Downarrow v}$$

For simplicity, no distinction between “type errors” (successor of non-number, etc.) and divergence.

A. Filinski On the relations between monadic semantics 2005-09-24

## Denotational semantics

Formulated with predomains. Notation:  $\lfloor \cdot \rfloor$ : lifting-inclusion;  $f^*$ : strict extension;  $\iota_1, \iota_2$ : injections;  $[f_1, f_2]$ : case-splitting. Two variants: **direct** and **continuation**.

$$\begin{array}{ll}
D^d = E_{\perp}^d & D^c = (E^c \rightarrow R) \rightarrow R \quad [R \text{ cppo}] \\
E^d \cong N + (D^d \rightarrow D^d) & E^c \cong N + (D^c \rightarrow D^c) \\
\llbracket t \rrbracket^d \in (V \rightarrow D^d) \rightarrow D^d & \llbracket t \rrbracket^c \in (V \rightarrow D^c) \rightarrow D^c \\
\llbracket \text{succ} \rrbracket^d = \lambda \rho. \lfloor \iota_2(\lambda d. [\lambda n. \lfloor \iota_1(n+1) \rfloor, \lambda f. \perp]^* d) \rfloor & \llbracket \text{succ} \rrbracket^c = \lambda \rho'. \lambda k. k (\iota_2(\lambda d. \lambda k. d [\lambda n. k(\iota_1(n+1)), \lambda f. \perp])) \\
\llbracket x \rrbracket^d = \lambda \rho. \rho x & \llbracket x \rrbracket^c = \lambda \rho'. \lambda k. \rho' x k \\
\llbracket \lambda^0 x. t \rrbracket^d = \lambda \rho. \lfloor \iota_2(\lambda d. [\llbracket t \rrbracket^d \rho[x \mapsto d]] \rfloor & \llbracket \lambda^0 x. t \rrbracket^c = \lambda \rho'. \lambda k. k (\iota_2(\lambda d. \lambda k. [\llbracket t \rrbracket^c \rho'[x \mapsto d] k])) \\
\llbracket \lambda^1 x. t \rrbracket^d = \lambda \rho. \lfloor \iota_2(\lambda d. (\lambda e. [\llbracket t \rrbracket^d \rho[x \mapsto [e]]]^* d) \rfloor & \llbracket \lambda^1 x. t \rrbracket^c = \lambda \rho'. \lambda k. k (\iota_2(\lambda d. \lambda k. d (\lambda e. [\llbracket t \rrbracket^c \rho'[x \mapsto \lambda k'. k' e] k))) \\
\llbracket t_1 t_2 \rrbracket^d = \lambda \rho. [\lambda n. \perp, \lambda f. f(\llbracket t_2 \rrbracket^d \rho)]^* (\llbracket t_1 \rrbracket^d \rho) & \llbracket t_1 t_2 \rrbracket^c = \lambda \rho'. \lambda k. [\llbracket t_1 \rrbracket^c \rho' [\lambda n. \perp, \lambda f. f(\llbracket t_2 \rrbracket^c \rho') k]]
\end{array}$$

( $\llbracket - \rrbracket^c$  can be extended to language with control operators;  $\llbracket - \rrbracket^d$  cannot.)

Take  $R = N_{\perp}$ . Then would expect, for closed  $t$ ,

$$\llbracket \lambda n. [n], \lambda f. \perp \rfloor^* (\llbracket t \rrbracket^d (\lambda x. \perp)) = \llbracket t \rrbracket^c (\lambda x. \perp) [\lambda n. [n], \lambda f. \perp]$$

Surprisingly hard to show. But nasty part is *not* the continuations.

## Showing agreement

$$\begin{array}{ll}
D^d = E_{\perp}^d & D^c = (E^c \rightarrow R) \rightarrow R \quad [R \text{ cppo}] \\
E^d \cong N + (D^d \rightarrow D^d) & E^c \cong N + (D^c \rightarrow D^c) \\
\llbracket x \rrbracket^d = \lambda \rho. \rho x & \llbracket x \rrbracket^c = \lambda \rho'. \lambda k. \rho' x k \\
\llbracket \lambda^0 x. t \rrbracket^d = \lambda \rho. \lfloor \iota_2(\lambda d. [\llbracket t \rrbracket^d \rho[x \mapsto d]] \rfloor & \llbracket \lambda^0 x. t \rrbracket^c = \lambda \rho'. \lambda k. k (\iota_2(\lambda d. \lambda k. [\llbracket t \rrbracket^c \rho'[x \mapsto d] k])) \\
\llbracket t_1 t_2 \rrbracket^d = \lambda \rho. [\lambda n. \perp, \lambda f. f(\llbracket t_2 \rrbracket^d \rho)]^* (\llbracket t_1 \rrbracket^d \rho) & \llbracket t_1 t_2 \rrbracket^c = \lambda \rho'. \lambda k. [\llbracket t_1 \rrbracket^c \rho' [\lambda n. \perp, \lambda f. f(\llbracket t_2 \rrbracket^c \rho') k]]
\end{array}$$

**Proof strategy:** define relations  $(\sim) \subseteq D^d \times D^c$  and  $(\approx) \subseteq E^d \times E^c$ , such that

$$\begin{aligned}
d \sim d' &\iff (d = \perp \wedge d' = \lambda k. \perp) \vee (\exists e \approx e'. d = [e] \wedge d' = \lambda k. k e') \\
e \approx e' &\iff (\exists n. e = \iota_1(n) \wedge e' = \iota_1(n)) \vee \\
&\quad (\exists f, f'. e = \iota_2(f) \wedge e' = \iota_2(f') \wedge \forall d \sim d'. f(d) \sim f'(d'))
\end{aligned}$$

Then straightforward to show, by structural induction on  $t$ :

$$(\forall x \in V. \rho x \sim \rho' x) \Rightarrow \llbracket t \rrbracket^d \rho \sim \llbracket t \rrbracket^c \rho'$$

From which, desired result follows immediately. **But does  $\sim$  exist at all?** Unlike logical relation, not defined by induction on type structure. Yes: see [Reynolds 74].

## Plan

- Introduce metalanguage with recursive types. (Rather more general than needed for expressing example language; in particular, accommodates multiple effects.)
- Compositionally translate example language to metalanguage (i.e., syntax-to-syntax). Obtain both direct and continuation semantics as particular domain-theoretic interpretations of metalanguage.
- Develop general principles for constructing type-indexed relations between interpretations of metalanguage, together with general logical-relations lemma.
- Instantiate general framework to conclude agreement of the two semantics for the example language.

## Metalanguage: syntax

Multi-monadic metalanguage, M<sup>3</sup>L (cf. [Moggi 89]). Parameterized by signature: collection of base types  $b$ , effects  $e$  (partially ordered by *behavior inclusion*,  $\preceq$ ), constants  $c$  (possibly denoting effectful functions).

**Value and computation types:**

$$\begin{aligned}\tau &::= \alpha \mid b \mid 1 \mid \tau_1 \times \tau_2 \mid 0 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \sigma \\ \sigma &::= \langle e \rangle \tau \mid \sigma_1 \times \sigma_2 \mid \tau \rightarrow \sigma\end{aligned}$$

$\Delta = (\alpha_1, \dots, \alpha_n)$ . Kinding judgments:  $\vdash_{\Delta} \tau$  type,  $\vdash_{\Delta} \sigma$   $e$ -type.

Think of terms of  $e$ -type as representing *parameterized  $e$ -computations*. (For products, parameter is 1 or 2; for functions, argument value.)

**Terms:**

$$\begin{aligned}M &::= c \mid x \mid () \mid (M_1, M_2) \mid \mathbf{fst}(M) \mid \mathbf{snd}(M) \mid \mathbf{inl}(M) \mid \mathbf{inr}(M) \mid \mathbf{void}(M) \\ &\quad \mid \mathbf{case}(M, x_1.M_1, x_2.M_2) \mid \mathbf{in}_{\mu\alpha.\tau}(M) \mid \mathbf{out}_{\mu\alpha.\tau}(M) \\ &\quad \mid \lambda x^\tau.M \mid M_1 M_2 \mid \mathbf{val}^e M \mid \mathbf{glet}_\sigma^e x \leftarrow M_1.M_2\end{aligned}$$

$\Gamma = (x_1:\tau_1, \dots, x_n:\tau_n)$ . Typing judgment:  $\Gamma \vdash_{\Delta} M : \tau$ .

## Kinding and typing rules

**Kinding:**

$$\begin{aligned}\frac{\vdash_{\Delta} \tau \text{ type}}{\vdash_{\Delta} \langle e' \rangle \tau \text{ } e\text{-type}} (e \preceq e') & \quad \frac{\vdash_{\Delta} \sigma_1 \text{ } e\text{-type} \quad \vdash_{\Delta} \sigma_2 \text{ } e\text{-type}}{\vdash_{\Delta} \sigma_1 \times \sigma_2 \text{ } e\text{-type}} \\ \frac{\vdash_{\Delta} \tau \text{ type} \quad \vdash_{\Delta} \sigma \text{ } e\text{-type}}{\vdash_{\Delta} \tau \rightarrow \sigma \text{ } e\text{-type}} & \quad [\text{admissible: } \frac{\vdash_{\Delta} \sigma' \text{ } e'\text{-type}}{\vdash_{\Delta} \sigma \text{ } e\text{-type}} (e \preceq e')]\end{aligned}$$

**Typing:** (key rules)

$$\frac{\Gamma \vdash_{\Delta} M : \tau}{\Gamma \vdash_{\Delta} \mathbf{val}^e M : \langle e \rangle \tau} \quad \frac{\Gamma \vdash_{\Delta} M_1 : \langle e \rangle \tau \quad \Gamma, x:\tau \vdash_{\Delta} M_2 : \sigma \quad \vdash_{\Delta} \sigma \text{ } e\text{-type}}{\Gamma \vdash_{\Delta} \mathbf{glet}_\sigma^e x \leftarrow M_1.M_2 : \sigma}$$

**Definable:** effect-inclusion

$$\frac{\Gamma \vdash_{\Delta} M : \langle e \rangle \tau}{\Gamma \vdash_{\Delta} \mathbf{inc}_\tau^{e,e'} M : \langle e' \rangle \tau} (e \preceq e') \quad \mathbf{inc}_\tau^{e,e'} M \equiv \mathbf{glet}_{\langle e \rangle \tau}^e x \leftarrow M. \mathbf{val}^{e'} x$$

## Domain-theoretic semantics of metalanguage

**Interpretation** of metalanguage signature:

- To every base type  $b$ , a cpo  $B^b$ .
- To every effect  $e$ , a [strong] monad  $(T^e, \eta^e, \star^e)$  (Kleisli-triple formulation):  $T^e A$  cppo,  $\eta_A^e : A \rightarrow T^e A$ ,  $\star_{A,B}^e : T^e A \times (A \rightarrow T^e B) \rightarrow T^e B$ ; 3 laws.
- To every inclusion  $e \preceq e'$ , a monad morphism  $i^{e,e'} : i_A^{e,e'} : T^e A \rightarrow T^{e'} A$ ; 2 laws. Also, assignment must be functorial:  $i^{e,e} = id$ ,  $i^{e,e''} = i^{e',e''} \circ i^{e,e'}$ .

**Determines** (where  $\theta \in \mathbf{Cpo}^{\Delta}$  assigns interprets type variables from  $\Delta$ ):

- For every  $\vdash_{\Delta} \tau$  type, a cpo  $\llbracket \tau \rrbracket_{\theta}$ . More generally, mixed functorial action:  $\llbracket \tau \rrbracket^f : (\mathbf{Cpo}_{\perp}^{\leftarrow})^{\Delta} \rightarrow \mathbf{Cpo}_{\perp}$ ; 2 laws. (Needed for defining  $\llbracket \mu\alpha.\tau \rrbracket$ .)
- For every  $\vdash_{\Delta} \sigma$   $e$ -type, a  $T^e$ -algebra  $\llbracket \sigma \rrbracket_{\theta}^e = (D, \gamma : T^e D \rightarrow D)$ ; 2 laws. With mixed functorial action:  $\llbracket \sigma \rrbracket^a : (\mathbf{Cpo}_{\perp}^{\leftarrow})^{\Delta} \rightarrow \mathbf{Cpo}_{\perp}^{\leftarrow}$ ; 1+2 laws. Special case: algebras for *lifting monad* are the *pointed* cpos.



## Semantics of metalanguage (terms)

**Interpretation:** (continued)

- To every constant  $c : \tau$ , an element  $C^c \in \llbracket \tau \rrbracket$ .

**Determines:** (where  $\rho \in \llbracket \Gamma \rrbracket_\theta = \prod_{x \in \text{dom } \Gamma} \llbracket \Gamma(x) \rrbracket_\theta$  interprets variables from  $\Gamma$ )

- For every term  $\Gamma \vdash_\Delta M : \tau$ , a continuous function,  $\llbracket M \rrbracket_\theta : \llbracket \Gamma \rrbracket_\theta \rightarrow \llbracket \tau \rrbracket_\theta$ . Sample clauses:

$$\begin{aligned} \llbracket x \rrbracket_\theta \rho &= \rho x \\ \llbracket \text{val}^e M \rrbracket_\theta \rho &= \eta^e(\llbracket M \rrbracket_\theta \rho) \\ \llbracket \text{glet}_\sigma^e x \Leftarrow M_1. M_2 \rrbracket_\theta \rho &= \gamma(\llbracket M_1 \rrbracket_\theta \rho \star^e (\lambda a. \eta^e \llbracket M_2 \rrbracket_\theta \rho[x \mapsto a])) \\ &\text{where } (D, \gamma) = \llbracket \sigma \rrbracket_\theta^e \end{aligned}$$

In particular:

$$\begin{aligned} \llbracket \text{glet}_{(e)\tau}^e x \Leftarrow M_1. M_2 \rrbracket_\theta \rho &= \llbracket M_1 \rrbracket_\theta \rho \star^e \lambda a. \llbracket M_2 \rrbracket_\theta \rho[x \mapsto a] \\ \llbracket \text{inc}^{e,e'} M \rrbracket_\theta \rho &= i^{e,e'}(\llbracket M \rrbracket_\theta \rho) \end{aligned}$$

## Translation of example language

Let  $\mathbf{c}$  be an effect name,  $\tau_N$  type interpreted as  $\mathbf{N}$ , e.g.,  $\tau_N = \mu\alpha.1 + \alpha$ .

Take  $\tau_E = \mu\alpha.\tau_N + (\langle \mathbf{c} \rangle \alpha \rightarrow \langle \mathbf{c} \rangle \alpha)$ , and  $\sigma_D = \langle \mathbf{c} \rangle \tau_E$ . Let  $c_{\text{err}} : \sigma_D$ .

When  $FV(t) = \{x_1, \dots, x_n\}$ , define  $x_1 : \sigma_D, \dots, x_n : \sigma_D \vdash \langle t \rangle : \sigma_D$  by

$$\begin{aligned} \langle x \rangle &= x \\ \langle \mathbf{!} \rangle &= \text{val}^c \text{in}_{\tau_E}(\text{inl}(n)) \\ \langle \text{succ} \rangle &= \text{val}^c \text{in}_{\tau_E}(\text{inr}(\lambda d. \text{glet}_{\sigma_D}^c e \Leftarrow d. \text{case}(\text{out}_{\tau_E} e, n. \text{val}^f \text{in}_{\tau_E}(\text{inl}(n+1)), f. c_{\text{err}}))) \\ \langle \lambda^n x. t \rangle &= \text{val}^c \text{in}_{\tau_E}(\text{inr}(\lambda x. \langle t \rangle)) \\ \langle \lambda^v x. t \rangle &= \text{val}^c \text{in}_{\tau_E}(\text{inr}(\lambda d. \text{glet}_{\sigma_D}^c e \Leftarrow d. (\lambda x. \langle t \rangle)(\text{val}^f e))) \\ \langle t_1 t_2 \rangle &= \text{glet}_{\sigma_D}^c e \Leftarrow \langle t_1 \rangle. \text{case}(\text{out}_{\tau_E} e, n. c_{\text{err}}, f. f \langle t_2 \rangle) \end{aligned}$$

Consider now *two* interpretations of signature: **primary** and **secondary**. Each determines semantics of metalanguage types,  $e$ -types, and terms:  $\langle \llbracket - \rrbracket, \llbracket - \rrbracket \rangle$ .

Take  $T_1^c$  as lifting monad,  $T_n^c$  as continuation monad with answer type  $\mathbf{N}_\perp$ ; then  $\llbracket \sigma_D \rrbracket = D^d$  and  $\llbracket \sigma \rrbracket = D^c$ . Take  $C_1^{c_{\text{err}}} = \perp$  and  $C_n^{c_{\text{err}}} = \lambda k. \perp$ ; then for any example-language term  $t$ ,  $\llbracket \langle t \rangle \rrbracket (\rho|_{FV(t)}) = \llbracket t \rrbracket^d \rho$  and  $\llbracket \langle t \rangle \rrbracket (\rho'|_{FV(t)}) = \llbracket t \rrbracket^c \rho'$ .

## Relating interpretations

**Relational interpretation** of signature, wrt. two ordinary interpretations:

- To every base type  $b$ , an admissible (= inclusive) *relation*,  $B_r^b \in \mathbf{ARel}(B_1^b, B_n^b)$ .
- To every effect  $e$ , a *relational action*  $T_r^e : \mathbf{ARel}(A_1, A_n) \rightarrow \mathbf{ARel}(T_1^e A_1, T_n^e A_n)$ , such that:
  1.  $\forall \langle a_i, a_n \rangle \in R. \langle \eta_1^e a_i, \eta_n^e a_n \rangle \in T_r^e R$ .
  2. If  $\forall \langle a_i, a_n \rangle \in R. \langle f_1 a_i, f_n a_n \rangle \in T_r^e S$ , then  $\forall \langle t_i, t_n \rangle \in T_r^e R. \langle t_i \star_1^e f_1, t_n \star_n^e f_n \rangle \in T_r^e S$ .
  3. If  $e \preceq e'$ , then  $\forall \langle t_i, t_n \rangle \in T_r^e R. (i_1^{e,e'} t_i, i_n^{e,e'} t_n) \in T_r^{e'} R$ .

**Determines** (given  $\varrho$  s.t.,  $\forall \alpha \in \Delta. \varrho(\alpha) \in \mathbf{ARel}(\theta_1(\alpha), \theta_n(\alpha))^2$ ):

- For every  $\vdash_\Delta \tau$  type, a relation  $\llbracket \tau \rrbracket_\varrho \in \mathbf{ARel}(\llbracket \tau \rrbracket_\theta, \llbracket \tau \rrbracket_{\theta_n})$ , e.g.,

$$\llbracket \langle e \rangle \tau \rrbracket_\varrho = T_r^e \llbracket \tau \rrbracket_\varrho \quad \llbracket \tau \rightarrow \sigma \rrbracket_\varrho = \{ \langle f, f \rangle \mid \forall \langle a_i, a_n \rangle \in \llbracket \tau \rrbracket_\varrho^\dagger. \langle f_1 a_i, f_n a_n \rangle \in \llbracket \sigma \rrbracket_\varrho \}$$

**Fundamental theorem.** Assume that for all  $c : \tau$ ,  $\langle C_1^c, C_n^c \rangle \in \llbracket \tau \rrbracket$ . Let  $\Gamma \vdash_\Delta M : \tau$ . If  $\forall (x_i : \tau_i) \in \Gamma. \langle \rho_1(x_i), \rho_n(x_i) \rangle \in \llbracket \tau_i \rrbracket_\varrho$ , then  $\langle \llbracket M \rrbracket_\theta, \rho \rangle, \llbracket M \rrbracket_{\theta_n} \rho_n \in \llbracket \tau \rrbracket_\varrho$ .

## Constructing relations for $\mu$ -types

$\llbracket \mu\alpha.\tau \rrbracket = A$ , where  $\phi : \llbracket \tau \rrbracket_{[\alpha \mapsto A]} \rightarrow A$  isomorphism,  $\llbracket \mathbf{in}_{\mu\alpha.\tau} M \rrbracket \rho = \phi(\llbracket M \rrbracket \rho)$ .

**Question:** when does the meaning equation,

$$\llbracket \mu\alpha.\tau \rrbracket = \{ \langle \phi m_i, \phi m_n \rangle \mid \langle m_i, m_n \rangle \in \llbracket \tau \rrbracket_{[\alpha \mapsto \llbracket \mu\alpha.\tau \rrbracket]} \}$$

have a solution? [Pitts 96] (paraphrased): enough to know:

- Domain  $\llbracket \mu\alpha.\tau \rrbracket$  is a *minimal invariant*:  $\text{fix}(\lambda f. \phi \circ \llbracket \tau \rrbracket_{[\alpha \mapsto (f, f)]}^f \circ \phi^{-1}) = \text{id}$ .  
This is ensured “for free” by standard inverse-limit construction.
- The relational action of  $\tau$  respects the functorial action:

$$\text{If } \langle \varphi_i, \varphi_n \rangle \in \varrho \iff \varrho' \text{ and } (a_i, a_n) \in \llbracket \tau \rrbracket_{\varrho} \text{ then } \langle \llbracket \tau \rrbracket_{\varphi_i}^f a_i, \llbracket \tau \rrbracket_{\varphi_n}^f a_n \rangle \in T_r^\perp \llbracket \tau \rrbracket_{\varrho'}.$$

Standard verification for usual type constructors in  $\tau$ . For  $\langle e' \rangle \tau'$ , follows from assumptions on  $T_r^{e'}$ , since  $\llbracket \langle e' \rangle \tau' \rrbracket_{\varphi}^f = \lambda t'. [t' \star^{e'} \lambda a. i^{\perp, e'}(\llbracket \tau' \rrbracket_{\varphi}^f a)]$ .

[A bit trickier when  $\tau$  can contain type variables other than  $\alpha$ .]

## Constructing relational actions for effects

**Syntactically**

Suppose monad is *definable* in metalanguage, i.e., exists  $\sigma_T$ ,  $M_\eta$  and  $M_\star$ , such that  $T^e A = \llbracket \sigma_T \rrbracket_{[\alpha \mapsto A]}$ ,  $\eta_A^e = \llbracket M_\eta \rrbracket_{[\alpha \mapsto A]}$ , and  $\star_{A, B}^e = \llbracket M_\star \rrbracket_{[\alpha \mapsto A, \beta \mapsto B]}$ , in both primary and secondary interpretation.

Given actions for all effects occurring in  $\sigma_T$ : can take  $T_r^e R = \llbracket \sigma_T \rrbracket_{[\alpha \mapsto (R, R)]}$ . Then action conditions ensured by Fundamental Theorem applied to  $M_\eta$  and  $M_\star$ .

*Summary:* if effect realized the same way in both semantics, its relational action can be built from that of its constituent effects. (Cf. definable base types.)

**As inverse image by monad morphisms**

Given effects  $e \preceq e'$  and relational action for  $e'$ ; define, for  $R \in \mathbf{AREl}\langle A_i, A_n \rangle$ ,  $T_r^e R = \{ \langle t_i, t_n \rangle \mid \langle i_i^{e, e'} t_i, i_n^{e, e'} t_n \rangle \in T_r^{e'} R \} \in \mathbf{AREl}\langle T_i^e A, T_n^e A \rangle$ .

Action conditions follow from monad-morphism laws. E.g., if  $\langle a_i, a_n \rangle \in R$ , then  $\langle \eta_i^e a_i, \eta_n^e a_n \rangle \in T_r^e R$ , because  $\langle i_i^{e, e'}(\eta_i^e a_i), i_n^{e, e'}(\eta_n^e a_n) \rangle = \langle \eta_i^{e'} a_i, \eta_n^{e'} a_n \rangle \in T_r^{e'} R$ .

*Summary:* if effect embeds into larger effect, can inherit relational action of that effect.

## Direct and continuation semantics revisited

Three effects:  $\mathbf{l} \preceq \mathbf{c} \preceq \mathbf{k}$ :

- **l** always interpreted as lifting monad,  $T_i^{\mathbf{l}} A = T_n^{\mathbf{l}} A = A_\perp$ .  $T_r^{\mathbf{l}} R = \{ \langle \perp, \perp \rangle \} \cup \{ \langle [a_i], [a_n] \rangle \mid \langle a_i, a_n \rangle \in R \}$ . In particular,  $\llbracket \langle \mathbf{l} \rangle \tau_N \rrbracket_{\varrho} = \{ \langle d, d \rangle \mid d \in \mathbf{N}_\perp \} = (=)$ .
- **k** interpreted as continuation monad. Definable:  $\sigma_T^{\mathbf{k}} = (\alpha \rightarrow \langle \mathbf{l} \rangle \tau_N) \rightarrow \langle \mathbf{l} \rangle \tau_N$ ,  $M_{\eta^{\mathbf{k}}} = \lambda a. \lambda k. k a$ ,  $M_{\star^{\mathbf{k}}} = \lambda(t, f). \lambda k. t(\lambda a. f a k)$ . Determines  $i^{\mathbf{l}, \mathbf{k}} t = \lambda k. k^* t$ ,  $T_r^{\mathbf{k}} R = \{ \langle t_i, t_n \rangle \mid \forall k_i, k_n. (\forall \langle a_i, a_n \rangle \in R. k_i a_i = k_n a_n) \Rightarrow t_i k_i = t_n k_n \}$ .
- **c** interpreted like **l** in primary semantics, like **k** in secondary. Relational action given by inverse image of **k**-action:  $T_r^{\mathbf{c}} R = \{ \langle t_i, t_n \rangle \mid \langle i^{\mathbf{c}, \mathbf{k}} t_i, i^{\mathbf{c}, \mathbf{k}} t_n \rangle \in T_r^{\mathbf{k}} R \} = \{ \langle t_i, t_n \rangle \mid \langle i^{\mathbf{l}, \mathbf{k}} t_i, i^{\mathbf{k}, \mathbf{k}} t_n \rangle \in T_r^{\mathbf{k}} R \}$ .

In context of Reynolds's original proof, corresponds to taking:

$$d \sim d' \iff \forall k, k'. (\forall e \approx e'. k e = k' e') \Rightarrow k^* d = d' k'$$

Weaker than  $d \sim d' \iff (d = \perp \wedge d' = \lambda k. \perp) \vee (\exists e \approx e'. d = [e] \wedge d' = \lambda k. k e')$ , but still sufficient for equivalence proof, more robust.

## General principle: effect-embeddings

**Primary semantics** For reasoning (human and automated). Full(er) abstraction: terms with indistinguishable behaviors *should* have equal denotations.

**Secondary semantics:** For implementation. Favor monadic representations that can be efficiently realized.

**Example:** checked vs. unchecked exceptions (Java).

Given set  $E$  of *exception names*. Every  $e \subseteq_{\text{fin}} E$  determines an effect, ordered by  $e \preceq e' \Leftrightarrow e \subseteq e'$ .  $\langle e \rangle \tau$  is type of computations that may raise exceptions from  $e$ .

- Specification interpretation:  $T_1^e A = A + e$ , keeps precise track of exception behavior, separate handlers for each.
- Implementation interpretation:  $T_{\parallel}^e A = A + E$ . One global handler.

**Ultimate consequence:** All effects in language realized by the same *universal* implementation monad (control + state).

Cf. types interpreted as subdomains of universal domain.

## Conclusions & future work

### Summary:

- Extended standard logical-relation construction (incl. recursive types) to models of effects. Enough extra semantic structure to justify dedicated investigation.
- Multimonadic metalanguage interesting in own right.

### Future work:

- Extend to *Kripke* logical relations: relating models of dynamic name creation, allocation of storage.
- Extend to *parametric polymorphism*, *syntax/semantics* relations (e.g., for computational adequacy), . . .
- Application: modular construction of multi-effect object languages, with call/cc+state-based implementation. (First cut: [Filinski 99: *Representing Layered Monads*]: finite linear order on effects.)

# Infinitary Combinatory Reduction Systems

## (Extended Abstract)

Jeroen Ketema<sup>1</sup> and Jakob Grue Simonsen<sup>2</sup>

<sup>1</sup> Department of Computer Science, Vrije Universiteit Amsterdam  
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands  
`jketema@cs.vu.nl`

<sup>2</sup> Department of Computer Science, University of Copenhagen (DIKU)  
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark  
`simonsen@diku.dk`

**Abstract.** We define infinitary combinatory reduction systems (iCRSs). This provides the first extension of infinitary rewriting to higher-order rewriting. We lift two well-known results from infinitary term rewriting systems and infinitary  $\lambda$ -calculus to iCRSs:

1. every reduction sequence in a fully-extended left-linear iCRS is compressible to a reduction sequence of length at most  $\omega$ , and
2. every complete development of the same set of redexes in an orthogonal iCRS ends in the same term.

## 1 Introduction

One of the main reasons for the initial research in infinitary rewriting was to have a model of lazy or stream-based programming languages easily accessible to people familiar with term rewriting. Two notions of infinitary rewriting were developed: infinitary (first-order) term rewriting systems (iTRSs) [1–3] and infinitary  $\lambda$ -calculus (i $\lambda$ c) [3, 4]. However, the standard notion of rewriting employed to model higher-order programs is *higher-order* rewriting, and thus goes beyond  $\lambda$ -calculus. The absence of a general notion of infinitary higher-order rewriting thus constitutes a gap in the arsenal of the rewriting theorist bent on modelling lazy or stream-based languages.

In the present paper we aim to plug this gap by investigating infinitary higher-order rewriting.

As for iTRSs and i $\lambda$ c some finitary system needs to be chosen as a starting point. We choose the notion of higher-order rewriting most familiar to the authors, namely combinatory reduction systems (CRSs) [3, 5, 6].

The definition of infinitary combinatory reduction systems (iCRSs) consists of a combination of the usual four-stage definition of CRSs and the corresponding four-stage definition of iTRSs and i $\lambda$ c:

CRSs	iTRSs/i $\lambda$ c
1a. Meta-terms	
1b. Terms	1. Infinite terms
2. Substitutions	2. Substitutions
3. Rewrite rules	3. Rewrite rules
4. Rewrite relation	4. Rewrite relation

Given the definition of iCRSs, we seek to answer two of the most pertinent questions asked for any notion of infinitary rewriting:

1. Are reduction sequences compressible to reduction sequences of length at most  $\omega$ ?
2. Do complete developments of the same set of redexes end in the same term?

For iTRSs these questions have positive answers under assumption of respectively left-linearity and orthogonality. For i $\lambda$ c the same holds as long as the  $\eta$ -rule is not introduced. Apart from the definition of iCRSs, the main contribution of this paper is that similar positive answers can be given in the case of iCRSs.

The remainder of this paper is organised as follows. In Sect. 2 we give some preliminary definitions, and in Sect. 3, we define infinite (meta-)terms and substitutions. Thereafter, in Sect. 4 we define infinitary rewriting and prove compression, and in Sect. 5 we investigate complete developments. Finally, in Sect. 6 we give directions for further research.

## 2 Preliminaries

Prior knowledge of CRSs [6] and infinitary rewriting [3] is not required, but will greatly improve the reader's understanding of the text.

Throughout the paper we assume a signature  $\Sigma$ , each element of which has finite arity. We also assume a countably infinite set of variables, and, for each finite arity, a countably infinite set of meta-variables. Countably infinite sets are sufficient, given that we can employ 'Hilbert hotel'-style renaming. We denote the first infinite ordinal by  $\omega$ , and arbitrary ordinals by  $\alpha, \beta, \gamma, \dots$

The set of *finite meta-terms* is defined as follows:

1. each variable  $x$  is a finite meta-term,
2. if  $x$  is a variable and  $s$  is a finite meta-term, then  $[x]s$  is a finite meta-term,
3. if  $Z$  is a meta-variable of arity  $n$  and  $s_1, \dots, s_n$  are finite meta-terms, then  $Z(s_1, \dots, s_n)$  is a finite meta-term,
4. if  $f \in \Sigma$  has arity  $n$  and  $s_1, \dots, s_n$  are finite meta-terms, then  $f(s_1, \dots, s_n)$  is a finite meta-term.

A finite meta-term of the form  $[x]s$  is called an *abstraction*. Each occurrence of the variable  $x$  in  $s$  is *bound* in  $[x]s$ . If  $s$  is a finite meta-term, we denote by  $root(s)$  the root symbol of  $s$ .

The *set of positions* of a finite meta-term  $s$ , denoted  $Pos(s)$ , is the set of finite strings over  $\mathbb{N}$ , with  $\epsilon$  the empty string, such that:

- if  $s = x$  for some variable  $x$ , then  $\mathcal{Pos}(s) = \{\epsilon\}$ ,
- if  $s = [x]t$ , then  $\mathcal{Pos}(s) = \{\epsilon\} \cup \{0 \cdot p \mid p \in \mathcal{Pos}(t)\}$ ,
- if  $s = Z(t_1, \dots, t_n)$ , then  $\mathcal{Pos}(s) = \{\epsilon\} \cup \{i \cdot p \mid 1 \leq i \leq n, p \in \mathcal{Pos}(t_i)\}$ ,
- if  $s = f(t_1, \dots, t_n)$ , then  $\mathcal{Pos}(s) = \{\epsilon\} \cup \{i \cdot p \mid 1 \leq i \leq n, p \in \mathcal{Pos}(t_i)\}$ .

Given  $p, q \in \mathcal{Pos}(s)$ , we say that  $p$  is a *prefix* of  $q$ , denoted  $p \leq q$ , if there exists an  $r \in \mathcal{Pos}(s)$  such that  $p \cdot r = q$ . If  $r \neq \epsilon$ , then we say that the prefix is *strict* and we write  $p < q$ . Moreover, if neither  $p < q$  nor  $q < p$ , then we say that  $p$  and  $q$  are *parallel*, which we denote  $p \parallel q$ . We denote by  $s|_p$  the subterm of  $s$  at position  $p$ .

### 3 (Meta-)Terms and Substitutions

In iTRSs and  $\text{i}\lambda\text{c}$ , terms are defined by means of introducing a metric on the set of finite terms and subsequently taking the completion of the metric. That is, taking the least set of objects containing the set finite terms such that every Cauchy sequence converges [2,4,7]. Intuitively, in such a metric, two terms  $s$  and  $t$  are close to each other if the first ‘conflict’ between them occurs ‘deep’ according to some depth measure. In iTRSs, a conflict is a position  $p$  such that  $\text{root}(s|_p) \neq \text{root}(t|_p)$ . In  $\text{i}\lambda\text{c}$ , a conflict is defined similarly, but also takes into account  $\alpha$ -equivalence. The metric, denoted  $d(s, t)$ , is defined as 0 when no conflict occurs between  $s$  and  $t$  and otherwise as  $2^{-k}$ , where  $k$  denotes the minimal depth such that a conflict occurs between  $s$  and  $t$ .

To define terms and meta-terms for iCRSs, we first define the notions of a conflict and  $\alpha$ -equivalence for finite meta-terms. In the definition we denote by  $s[x \rightarrow y]$  the replacement in  $s$  of the occurrences of the free variable  $x$  by the variable  $y$ .

**Definition 3.1.** *Let  $s$  and  $t$  be finite meta-terms. A conflict of  $s$  and  $t$  is a position  $p \in \mathcal{Pos}(s) \cap \mathcal{Pos}(t)$  such that:*

1. *if  $p = \epsilon$ , then  $\text{root}(s) \neq \text{root}(t)$ ,*
2. *if  $p = i \cdot q$  for  $i \geq 1$ , then  $\text{root}(s) = \text{root}(t)$  and  $q$  is a conflict of  $s|_i$  and  $t|_i$ ,*
3. *if  $p = 0 \cdot q$ , then  $s = [x_1]s'$  and  $t = [x_2]t'$  and  $q$  is a conflict of  $s'[x_1 \rightarrow y]$  and  $t'[x_2 \rightarrow y]$ , where  $y$  does not occur in either  $s'$  or  $t'$ .*

*The finite meta-terms  $s$  and  $t$  are  $\alpha$ -equivalent if no conflict exists [4].*

We next define the depth measure  $D$ .

**Definition 3.2.** *Let  $s$  be a meta-term and  $p \in \mathcal{Pos}(s)$ . Define:*

$$\begin{aligned} D(s, \epsilon) &= 0 \\ D(Z(t_1, \dots, t_n), i \cdot p') &= D(t_i, p') \\ D([x]t, 0 \cdot p') &= 1 + D(t, p') \\ D(f(t_1, \dots, t_n), i \cdot p') &= 1 + D(t_i, p') \end{aligned}$$

Note that meta-variables are not counted by  $D$ . Changing the second clause to  $D(Z(t_1, \dots, t_n), i \cdot p') = 1 + D(t_i, p')$  yields the ‘usual’ depth measure, which counts the number of symbols in a position.

The measure  $D$  is employed in the definition of the metric, which is defined precisely as in the case of iTRSs and  $\lambda\mathcal{C}$ .

**Definition 3.3.** *Let  $s$  and  $t$  be meta-terms. The metric  $d$  is defined as:*

$$d(s, t) = \begin{cases} 0 & \text{if } s \text{ and } t \text{ are } \alpha\text{-equivalent} \\ 2^{-k} & \text{otherwise,} \end{cases}$$

where  $k$  is the minimal depth with respect to the measure  $D$  such that a conflict occurs between  $s$  and  $t$ .

Following precisely the definition of terms in the case of iTRSs and  $\lambda\mathcal{C}$ , we define the meta-terms.

**Definition 3.4.** *The set of meta-terms over a signature  $\Sigma$  is the metric completion of the set of finite meta-terms with respect to the metric  $d$ .*

Note that, by definition of metric completion, the set of finite meta-terms is a subset of the set of meta-terms.

The notions of a set of positions and a subterm of a finite meta-term carry over directly to the meta-terms, we use the same notation in both cases.

The metric completion allows precisely those meta-terms such that the depth measure  $D$  increases to infinity along all infinite paths in the meta-term. Thus, by the definition of  $D$  and  $d$ , no meta-term has a subterm  $s$  such that there exists an infinite string  $p$  over  $\mathbb{N}$  with the property that each finite prefix  $q$  of  $p$  is a position of  $s$  with  $\text{root}(s|_q)$  a meta-variable. Informally, *no meta-term has an infinite chain of meta-variables.*

Examples of candidate ‘meta-terms’ that are disallowed by the definition of meta-term are:

$$\begin{aligned} & Z(Z(\dots(Z(\dots)))) \\ & Z_1(Z_2(\dots(Z_n(\dots)))) \end{aligned}$$

A construction that *is* allowed is an infinite number of *finite* chains of meta-variables ‘guarded’ by abstractions or function symbols. For example, the following is allowed:

$$[x_1]Z_1([x_2]Z_2(\dots([x_n]Z_n(\dots))))$$

If we had wanted to include ‘meta-terms’ with infinite chains of meta-variables we should have used the usual depth measure on finite meta-terms instead of the measure  $D$ .

We explain the reason for the exclusion of meta-terms with infinite chains of meta-variables after the definition of substitutions. The idea of the exclusion of certain meta-terms comes from  $\lambda\mathcal{C}$  where it is possible to define subsets of the set of infinite  $\lambda$ -terms by slightly changing the notion of the depth measure on which the metric is based [4]. It is, for example, possible to define a subset in which no  $\lambda$ -terms with infinite chains of  $\lambda$ -abstractions occur, i.e., subterms of the form  $\lambda x_1.\lambda x_2 \dots \lambda x_n \dots$  are disallowed.

The terms can now be defined as in the finite case [3,5,6]. The only difference is that meta-terms now occur in the definition instead of finite meta-terms.

**Definition 3.5.** *The set of terms is the largest subset of the set of meta-terms, such that no meta-variables occur in the meta-terms.*

Note that the definition of meta-terms, as defined by the measure  $D$ , only restricts meta-terms containing meta-variables, not meta-terms *without* meta-variables. Hence, the set of terms is independent of the use of either  $D$  in Definition 3.3 or the usual depth measure. As a consequence, both the set of (infinite) first-order terms and the set of (infinite)  $\lambda$ -terms are easily shown to be included in the set of terms.

We next define substitutions. The required definitions are the same as in the case of CRSs [3, 6], except that coinduction is employed instead of induction. This is identical to what is done in the case of iTRSs and  $i\lambda c$  with respect to the finite systems they are based on. In the definitions we use  $\mathbf{x}$  and  $\mathbf{t}$  as a short-hands for respectively the sequences  $x_1, \dots, x_n$  and  $t_1, \dots, t_n$  with  $n \geq 0$ . We assume  $n$  fixed in the next two definitions.

**Definition 3.6.** *A substitution of the terms  $\mathbf{t}$  for distinct variables  $\mathbf{x}$  in a term  $s$ , denoted  $s[\mathbf{x} := \mathbf{t}]$ , is coinductively defined as:*

1.  $x_i[\mathbf{x} := \mathbf{t}] = t_i$ ,
2.  $y[\mathbf{x} := \mathbf{t}] = y$  if  $y$  does not occur in  $\mathbf{x}$ ,
3.  $([y]s')[\mathbf{x} := \mathbf{t}] = [y](s'[\mathbf{x} := \mathbf{t}])$ ,
4.  $f(s_1, \dots, s_m)[\mathbf{x} := \mathbf{t}] = f(s_1[\mathbf{x} := \mathbf{t}], \dots, s_m[\mathbf{x} := \mathbf{t}])$ .

The above definition implicitly takes into account the variable convention [8] in the third clause to avoid the binding of free variables by the abstraction.

**Definition 3.7.** *An  $n$ -ary substitute is a mapping denoted  $\underline{\lambda}\mathbf{x}_1, \dots, \mathbf{x}_n.s$  or  $\underline{\lambda}\mathbf{x}.s$ , with  $s$  a term, such that:*

$$(\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n) = s[\mathbf{x} := \mathbf{t}]. \quad (1)$$

Reading Eq. (1) from left to right gives rise to the rewrite rule

$$(\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n) \rightarrow s[\mathbf{x} := \mathbf{t}].$$

This rule can be seen a *parallel  $\beta$ -rule*. That is, a variant of the  $\beta$ -rule from  $i\lambda c$  which substitutes for multiple variables simultaneously. The root of  $(\underline{\lambda}\mathbf{x}.s)$  is called the  $\underline{\lambda}$ -abstraction and the root of the left-hand side of the parallel  $\beta$ -rule is called the  $\underline{\lambda}$ -application.

**Definition 3.8.** *A valuation  $\bar{\sigma}$  is an extension of a function  $\sigma$  which assigns  $n$ -ary substitutes to  $n$ -ary meta-variables. It is coinductively defined as:*

1.  $\bar{\sigma}(x) = x$ ,
2.  $\bar{\sigma}([x]s) = [x](\bar{\sigma}(s))$ ,
3.  $\bar{\sigma}(Z(s_1, \dots, s_m)) = \sigma(Z)(\bar{\sigma}(s_1), \dots, \bar{\sigma}(s_m))$ ,
4.  $\bar{\sigma}(f(s_1, \dots, s_m)) = f(\bar{\sigma}(s_1), \dots, \bar{\sigma}(s_m))$ .

Similar to Definition 3.6, the above definition implicitly takes into account the variable convention in the second clause to avoid the binding of free variables by the abstraction.

Thus, applying a substitution means applying a valuation and proceeds in two steps: In the first step each subterm of the form  $Z(t_1, \dots, t_n)$  is replaced by a subterm of the form  $(\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n)$ . In the second step Eq. (1) is applied to each subterm of the form  $(\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n)$  as introduced in the first step.



In the light of the rewrite rule introduced just below Definition 3.7 the second step can be viewed as a complete development of the parallel  $\beta$ -redexes introduced in the first step. This is obviously a complete development in a variant of  $\text{i}\lambda\text{c}$ . The variant has the parallel  $\beta$ -rule and a signature containing the  $\underline{\lambda}$ -application, the  $\underline{\lambda}$ -abstraction, the abstractions, the meta-variables, and the elements of  $\Sigma$ .

As in the finite case [5, Remark II.1.10.1], we need to prove that the application of a valuation to a meta-term yields a unique term.

**Proposition 3.9.** *Let  $s$  be a meta-term and  $\bar{\sigma}$  a valuation. There exists a unique term that is the result of applying  $\bar{\sigma}$  to  $s$ .*

*Proof (Sketch).* That the first step in applying  $\bar{\sigma}$  to  $s$  has a unique result is an immediate consequence of being defined coinductively. We denote the result of the first step by  $s_\sigma$ . The set of parallel  $\beta$ -redexes in  $s_\sigma$  is denoted  $\mathcal{U}$ .

To prove that the second step also has a unique result we employ the rewriting terminology as introduced above. Although omitted, the definitions of a development and a complete development can be easily derived from the  $\text{i}\lambda\text{c}$  definitions.

Note that to repeatedly rewrite the root of  $s_\sigma$  by means of the parallel  $\beta$ -redex, the root must look like

$$(\underline{\lambda}x_i)(t_1, \dots, t_n),$$

with  $1 \leq i \leq n$  and  $t_i$  again such a redex. This is only possible if there exists in  $s_\sigma$  an infinite chain of such redexes which starts at the root. However, this requires an infinite chain of meta-variables to be present in  $s$ , which is not allowed by the definition of meta-terms. Thus, the root can only be rewritten finitely often in a development. Applying the same reasoning to the roots of the subterms, gives that a complete development is obtained by reducing the redexes in  $\mathcal{U}$  in an outside-in fashion. As all parallel  $\beta$ -redexes occur in  $\mathcal{U}$  and as no  $\underline{\lambda}$ -applications and  $\underline{\lambda}$ -abstractions occur in  $s$  the result of the complete development, which we denote  $\bar{\sigma}(s)$ , is necessarily a term.

To show that each complete development ends in  $\bar{\sigma}(s)$ , note that we can view each parallel  $\beta$ -redex  $(\underline{\lambda}x_1, \dots, x_n.s)(t_1, \dots, t_n)$  as a sequence of  $\beta$ -redexes:

$$(\lambda x_1(\dots((\lambda x_n.s)t_n)\dots))t_1.$$

This means that each complete development in our variant of  $\text{i}\lambda\text{c}$  corresponds to a complete development in  $\text{i}\lambda\text{c}$  extended with some function symbols. As each complete development in  $\text{i}\lambda\text{c}$  ends in the same term, a result independent of added function symbols, the complete developments of the second step must also end in the same term. Hence,  $\bar{\sigma}(s)$  is unique.  $\square$

Let us now see why we excluded ‘meta-terms’ with infinite chains of meta-variables from Definition 3.4. Consider the ‘meta-term’

$$Z(Z(\dots(Z(\dots))))).$$

Applying the valuation that assigns to  $Z$  the substitute  $\underline{\lambda}x.x$  yields:

$$(\underline{\lambda}x.x)((\underline{\lambda}x.x)(\dots((\underline{\lambda}x.x)(\dots))))$$

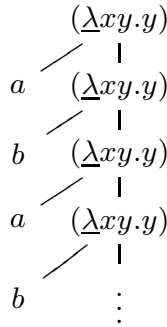
which has no complete development, as no matter how many parallel  $\beta$ -redexes are contracted, it reduces only to itself and not to a term. This is inadequate, as rewrite steps in iCRSs need to relate terms to terms.

The previous problem does not depend on only a single meta-variable being present in the ‘meta-term’. The same behaviour can occur with different meta-variables of different arities. In that case, we can define a valuation that assigns  $\underline{\lambda}\mathbf{x}.y$  to each meta-variable  $Z$  in the ‘meta-term’ with  $y$  in  $\mathbf{x}$  such that  $y$  corresponds to an argument of  $Z$  which is a chain of meta-variables.

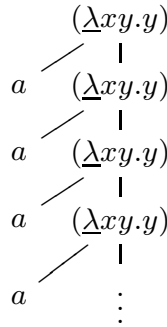
The above ‘meta-term’ still has the nice property that it exhibits confluence with respect to the parallel  $\beta$ -rule. Unfortunately, there are ‘meta-terms’ that do not have this property. Consider a signature with constants  $a$  and  $b$  and also consider the ‘meta-term’

$$Z(a, Z(b, Z(a, Z(b, Z(\dots))))).$$

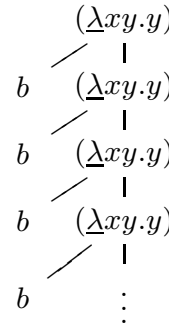
Applying the valuation that assigns to  $Z$  the substitute  $\underline{\lambda}xy.y$  yields the ‘ $\underline{\lambda}$ -term’ of Fig. 1. It reduces by means of two different developments to the  $\underline{\lambda}$ -terms of Fig. 2 and Fig. 3. These last two  $\underline{\lambda}$ -terms have no common reduct with respect to parallel  $\beta$ -reduction. They reduce only to themselves. Note that this problem also occurs in  $\text{i}\lambda\text{c}$  [4, Sect. 4].



**Fig. 1.**



**Fig. 2.**



**Fig. 3.**

Concluding, when we allow ‘meta-terms’ with infinite chains of meta-variables we have two problems. First, substitution in such a ‘meta-term’ does not always yield a term. Second, substitution may yield distinct results, none of which are terms. We can overcome these problems by not allowing infinite chains of meta-variables to occur in meta-terms, as shown in Proposition 3.9.

## 4 Infinitary Rewriting

We continue to combine the definitions of iTRSs and  $\text{i}\lambda\text{c}$  and those of CRSs. We start with a definition that comes directly from CRS theory.

**Definition 4.1.** *A finite meta-term is a pattern if each of its meta-variables has distinct bound variables as its arguments. Moreover, a meta-term is closed if all its variables occur bound.*

We next define rewrite rules and iCRSs. In analogy to the rewrite rules of iTRSs, the definition is identical to the one in the finitary case, but without the finiteness restriction on the right-hand sides of the rewrite rules [1,2].

**Definition 4.2.** *A rewrite rule is a pair  $(l, r)$ , denoted  $l \rightarrow r$ , where  $l$  is a finite meta-term and  $r$  is a meta-term, such that:*

1.  $l$  is a pattern and of the form  $f(s_1, \dots, s_n)$  with  $f \in \Sigma$  of arity  $n$ ,
2. all meta-variables that occur in  $r$  also occur in  $l$ , and
3.  $l$  and  $r$  are closed.

An infinitary combinatory reduction system (iCRS) is a pair  $\mathcal{C} = (\Sigma, R)$  with  $\Sigma$  a signature and  $R$  a set of rewrite rules.

As the rewrite rules of iTRSs and  $\text{i}\lambda\text{c}$  only have finite chains of meta-variables when their rules are considered as rewrite rules in the above sense, it follows easily that iTRSs and  $\text{i}\lambda\text{c}$  are iCRSs.

A context is a term over  $\Sigma \cup \{\square\}$  where  $\square$  is a fresh constant. One-hole contexts are defined in the usual way. We now define redexes and rewrite steps.

**Definition 4.3.** *Let  $l \rightarrow r$  be a rewrite rule. Given a valuation  $\bar{\sigma}$ , the term  $\bar{\sigma}(l)$  is called a  $l \rightarrow r$ -redex. If  $s = C[\bar{\sigma}(l)]$  for some context  $C[\square]$  with  $\bar{\sigma}(l)$  a  $l \rightarrow r$ -redex and  $p$  the position of the hole in  $C[\square]$ , then an  $l \rightarrow r$ -redex, or simply a redex, occurs at position  $p$  and depth  $D(s, p)$  in  $s$ . A rewrite step is a pair  $(s, t)$ , denoted  $s \rightarrow t$ , such that a  $l \rightarrow r$ -redex occurs in  $s = C[\bar{\sigma}(l)]$  and such that  $t = C[\bar{\sigma}(r)]$ .*

We can now define what a transfinite reduction sequence is. The definition copies the definition from iTRSs and  $\text{i}\lambda\text{c}$  verbatim [2,4].

**Definition 4.4.** *A transfinite reduction sequence of ordinal length  $\alpha$  is a sequence of terms  $(s_\beta)_{\beta < \alpha+1}$  such that  $s_\beta \rightarrow s_{\beta+1}$  for all  $\beta < \alpha$ . For each rewrite step  $s_\beta \rightarrow s_{\beta+1}$ , let  $d_\beta$  denote the depth of the contracted redex. The reduction sequence is weakly convergent or Cauchy convergent if for every ordinal  $\gamma \leq \alpha$  the distance between  $t_\beta$  and  $t_\gamma$  tends to 0 as  $\beta$  approaches  $\gamma$  from below. The reduction sequence is strongly convergent if it is weakly convergent and if  $d_\beta$  tends to infinity as  $\beta$  approaches  $\gamma$  from below.*

*Notation 4.5.* By  $s \twoheadrightarrow^\alpha t$ , respectively  $s \twoheadrightarrow^{\leq \alpha} t$ , we denote a strongly convergent transfinite reduction sequence of ordinal length  $\alpha$ , respectively of ordinal length less than or equal to  $\alpha$ . By  $s \twoheadrightarrow t$  we denote a strongly convergent transfinite reduction sequence of arbitrary ordinal length and by  $s \twoheadrightarrow^* t$  we denote a reduction sequence of finite length.

As in [2–4], we prefer to reason about strongly converging reduction sequences. This ensures that we can restrict our attention to reduction sequences of length at most  $\omega$  by the so-called *compression property*. To prove the property we need the following lemma and definitions.

**Lemma 4.6.** *If  $s \rightarrow t$ , then the number of steps contracting redexes at depths less than  $d \in \mathbb{N}$  is finite for any  $d$ .*

*Proof.* This is exactly the proof of [2, Lemma 3.5]. □

**Definition 4.7.** *A rewrite rule  $l \rightarrow r$  is left-linear, if each meta-variable occurs at most once in  $l$ . Moreover, an iCRS is left-linear if all its rewrite rules are left-linear.*

**Definition 4.8.** *A pattern is fully-extended [9, 10], if, for each of its meta-variables  $Z$ , and each abstraction  $[x]$  having  $Z$  in its scope,  $x$  is an argument of  $Z$ . Moreover, an iCRS is fully-extended if the left-hand sides of all rewrite rules are fully-extended.*

Left-linearity and fully-extendedness ensure no redex is created by either making two subterms equal in an infinite number of steps or by erasing some variable in an infinite number of steps.

**Theorem 4.9 (Compression).** *For every fully-extended, left-linear iCRS, if  $s \rightarrow^\alpha t$ , then  $s \twoheadrightarrow^{\leq \omega} t$ .*

*Proof (Sketch).* Let  $s \rightarrow^\alpha t$ , and proceed by ordinal induction on  $\alpha$ . By [3, Theorem 12.7.1] it suffices to show that the theorem holds for  $\alpha = \omega + 1$ : The cases where  $\alpha$  is 0, a limit ordinal, or a successor ordinal greater than  $\omega + 1$  do not depend on the definition of rewriting.

For  $\alpha = \omega + 1$  it follows by Lemma 4.6 that we can write  $s \rightarrow^\alpha t$  as  $s \rightarrow^* s' \twoheadrightarrow^\omega s'' \rightarrow t$ , such that all rewrite steps in  $s' \twoheadrightarrow^\omega s''$  occur below the meta-variable positions of the redex contracted in the step of  $s'' \rightarrow t$ . By fully-extendedness and left-linearity it follows that a redex of which the redex contracted in  $s'' \rightarrow t$  is a residual occurs in  $s'$ . Hence, we can contract the redex in  $s'$ , which yields a term  $t'$ .

The result now follows if we can construct a strongly convergent reduction sequence  $t' \twoheadrightarrow^{\leq \omega} t$ . To construct such a reduction sequence, assume  $t_0 = t'$  and construct for each  $d > 0$  a reduction sequence  $t_{d-1} \rightarrow^* t_d$  where all rewrite steps occur at depths greater or equal to  $d - 1$ , and where  $d(t_d, t) \leq 2^{-d}$ . That the construction of these reduction sequences is possible follows by a proof that is similar to the proof of compression for  $\text{i}\lambda\text{c}$  [4]. Using the fact that only finite chains of meta-variables occur in meta-terms is essential to the proof. By the requirements on the constructed reduction sequences, it follows that  $t_0 \rightarrow^* t_1 \rightarrow^* \dots \rightarrow^* t_{d-1} \rightarrow^* t_d \rightarrow^* \dots \rightarrow^* t$  is a strongly convergent reduction sequence of length at most  $\omega$ . As  $s \rightarrow^* t'$ , we then have that  $s \twoheadrightarrow^{\leq \omega} t$ , as required. □

The previous theorem does not hold in general for iCRSs that are not left-linear or fully-extended. For left-linearity, this follows from the iTRS counterexample in [2]. For fully-extendedness, this follows from the infinitary  $\lambda\beta\eta$ -calculus in which reduction sequences occur that are not compressible to reduction sequences of length at most  $\omega$  [3, 4]. The  $\eta$ -rule is not fully-extended.

## 5 Developments

In this section we prove that each complete development of the same set of redexes in an orthogonal iCRS ends in the same term. As all the left-hand sides of the rewrite rules in iCRSs are finite, the definition of orthogonality carries over immediately from CRSs.

**Definition 5.1.** *Let  $R = \{l_i \rightarrow r_i \mid i \in I\}$  be a set of rewrite rules.*

1.  *$R$  is non-overlapping if it holds that:*
  - *each  $l_i \rightarrow r_i$ -redex that occurs at a position  $p$  in an  $l_j \rightarrow r_j$ -redex with  $i \neq j$  occurs such that there exists a position  $q \leq p$  with  $q \in \text{Pos}(l_j)$  and  $\text{root}(l_j|_p)$  a meta-variable,*
  - *likewise for  $p \neq \epsilon$  and  $i = j$ .*
2.  *$R$  is orthogonal if it is left-linear and non-overlapping.*
3. *An iCRS is orthogonal if its set of rewrite rules is orthogonal.*

In the remainder of this section we assume an orthogonal iCRS, a term  $s$ , and a set  $\mathcal{U}$  of redexes in  $s$ .

### 5.1 Descendants and Residuals

Before we can consider developments, we need to define descendants and residuals. The definition of descendant across a rewrite step  $\bar{\sigma}(l) \rightarrow \bar{\sigma}(r)$  follows the definition of substitution, and is thus defined in two steps. The first step defines descendants in  $\bar{\sigma}(r)$  where only the valuation is applied and not Eq. (1). The second step defines descendants across application of Eq. (1).

Given that the second step of the substitution is just a complete development in a variant of  $\text{i}\lambda\text{c}$ , the second step in the definition of descendants is just a variant of descendants in  $\text{i}\lambda\text{c}$  [3, 4]. For this reason, the step is not made explicit here.

We next give a definition of the first step. In the definition we denote by 0 the position of the subterm on the left-hand side of a  $\underline{\lambda}$ -application and also the position of the body of a  $\underline{\lambda}$ -abstraction. By  $1, \dots, n$  we denote the positions of the subterms on the right-hand side of the  $\underline{\lambda}$ -application. This means that  $(\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n)|_0 = (\underline{\lambda}\mathbf{x}.s)$ ,  $\underline{\lambda}\mathbf{x}.s|_0 = s$ , and  $Z(t_1, \dots, t_n)|_i = (\underline{\lambda}\mathbf{x}.s)(t_1, \dots, t_n)|_i = t_i$  for  $1 \leq i \leq n$ . We denote by  $\bar{\sigma}(l) \rightarrow r_\sigma$  the rewrite step  $\bar{\sigma}(l) \rightarrow \bar{\sigma}(r)$  when only the first step of the substitution applied to  $r$ .

**Definition 5.2.** *Let  $l \rightarrow r$  be a rewrite rule,  $\bar{\sigma}$  a valuation, and  $p \in \text{Pos}(\bar{\sigma}(l))$ . Suppose  $u : \bar{\sigma}(l) \rightarrow r_\sigma$ . The set  $p/{}^1u$  is defined as follows:*

- *if a position  $q \in \text{Pos}(l)$  exists such that  $p = q \cdot q'$  and  $\text{root}(l|_q) = Z$ , then define  $p/{}^1u = \{p' \cdot 0 \cdot 0 \cdot q' \mid p' \in P\}$  with  $P = \{p' \mid \text{root}(r|_{p'}) = Z\}$ ,*
- *if no such position exists, then define  $p/{}^1u = \emptyset$ .*

Note that  $\mathcal{Pos}(r) \subseteq \mathcal{Pos}(r_\sigma)$  by the notation of positions in subterms of the form  $(\underline{\lambda}x.s)(t_1, \dots, t_n)$ . From this it follows that  $P \subseteq \mathcal{Pos}(r_\sigma)$ .

We can now give a complete definition of a descendant across a rewrite step.

**Definition 5.3.** Let  $u : C[\bar{\sigma}(l)] \rightarrow C[\bar{\sigma}(r)]$  be a rewrite step, such that  $p$  is the position of the hole in  $C[\square]$ , and let  $q \in \mathcal{Pos}(C[\bar{\sigma}(l)])$ . The set of descendants of  $q$  across  $u$ , denoted  $q/u$ , is defined as  $q/u = \{q\}$  in case  $p \parallel q$  or  $p < q$ . In case  $q = p \cdot q'$ , it is defined as  $q/u = \{p \cdot q'' \mid p'' \in Q\}$ , where  $Q$  is the set of descendants of  $q'/^1u'$  with  $u' : \bar{\sigma}(l) \rightarrow r_\sigma$  across complete development of the parallel  $\beta$ -redexes in  $r_\sigma$ .

Descendants across a reduction sequence are defined as for iTRSs and  $\lambda c$ .

**Definition 5.4.** Let  $s_0 \twoheadrightarrow^\alpha s_\alpha$  and let  $P \subseteq \mathcal{Pos}(s_0)$ . The set of descendants of  $P$  across  $s_0 \twoheadrightarrow^\alpha s_\alpha$ , denoted  $P/(s_0 \twoheadrightarrow^\alpha s_\alpha)$ , is defined as follows:

- if  $\alpha = 0$ , then  $P/(s_0 \twoheadrightarrow^\alpha s_\alpha) = P$ ,
- if  $\alpha = 1$ , then  $P/(s_0 \twoheadrightarrow s_1) = \bigcup_{p \in P} p/(s_0 \rightarrow s_1)$ ,
- if  $\alpha = \beta + 1$ , then  $P/(s_0 \twoheadrightarrow^{\beta+1} s_{\beta+1}) = (P/(s_0 \twoheadrightarrow^\beta s_\beta))/(s_\beta \rightarrow s_{\beta+1})$ ,
- if  $\alpha$  is a limit ordinal, then  $p \in P/(s_0 \twoheadrightarrow^\alpha s_\alpha)$  iff  $p \in P/(s_0 \twoheadrightarrow^\beta s_\beta)$  for all large enough  $\beta < \alpha$ .

By orthogonality, if there exists a redex at a position  $p$  using a rewrite rule  $l \rightarrow r$  that is not contracted in rewrite step and if  $p$  has descendants across the step, then there exists a redex at each descendant of  $p$  also employing the rule  $l \rightarrow r$ . Hence, there exists a well-defined notion of *residual* by strongly convergent reduction sequences. We overload the notation  $\cdot/\cdot$  to denote both the descendant and the residual relation.

## 5.2 Complete Developments

We now define developments. Recall that we assume we are working in an orthogonal iCRS and that  $\mathcal{U}$  is a set of redexes in a term  $s$ .

**Definition 5.5.** A development of  $\mathcal{U}$  is a strongly convergent reduction sequence such that each step contracts a residual of a redex in  $\mathcal{U}$ . A development  $s \twoheadrightarrow t$  is complete if  $\mathcal{U}/(s \twoheadrightarrow t) = \emptyset$ .

To prove that each complete development of the same set of redexes ends in the same term, we extend the technique of the Finite Jumps Developments Theorem [3] to orthogonal iCRSs. The theorem employs notions of paths and path projections. In essence, paths and path projections are ‘walks’ through terms starting at the root and proceeding to greater and greater depths. An important property of paths and path projections is that when a walk encounters a redex to be contracted in a development, a ‘jump’ is made to the right-hand side of the employed rewrite rule. It continues there until a meta-variable is encountered, at which point a jump back to the original term occurs.

In the following definition, we denote by  $p_u$  the position of the redex  $u$  in  $s$ .

**Definition 5.6.** A path of  $s$  with respect to  $\mathcal{U}$  is a sequence of nodes and edges. Each node is labelled either  $(s, p)$  with  $p \in \mathcal{P}os(s)$  or  $(r, p, q)$  with  $r$  a right-hand side of a rewrite rule,  $p \in \mathcal{P}os(r)$ , and  $q = p_u$  with  $u \in \mathcal{U}$ . Each directed edge is either unlabelled or labelled with an element of  $\mathbb{N}$ .

Every path starts with a node labelled  $(s, \epsilon)$ . If a node  $n$  of a path is labelled  $(s, p)$  and if it has an outgoing edge to a node  $n'$ , then:

1. if the subterm at  $p$  is not a redex in  $\mathcal{U}$ , then for some  $i \in \mathcal{P}os(s|_p) \cap \mathbb{N}$  the node  $n'$  is labelled  $(s, p \cdot i)$  and the edge from  $n$  to  $n'$  is labelled  $i$ ,
2. if the subterm at  $p$  is a redex  $u \in \mathcal{U}$  with  $l \rightarrow r$  the employed rewrite rule, then the node  $n'$  is labelled  $(r, \epsilon, p_u)$  and the edge from  $n$  to  $n'$  is unlabelled,
3. if  $s|_p$  is a variable  $x$  bound by an abstraction  $[x]$  occurring in the left-hand side of the rule  $l \rightarrow r$  of a redex  $u \in \mathcal{U}$ , then the node  $n'$  is labelled  $(r, p' \cdot i, p_u)$  and the edge from  $n$  to  $n'$  is unlabelled, such that  $(r, p', p_u)$  was the last node before  $n$  with  $p_u$ ,  $\text{root}(r|_{p'}) = Z$ , the unique position of  $Z$  in  $l$  is  $q$ , and  $l|_{q \cdot i} = x$ .

If a node  $n$  of a path is labelled  $(r, p, p_u)$  and if it has an outgoing edge to a node  $n'$ , then:

1. if  $\text{root}(r|_p)$  is not a meta-variable, then for some  $i \in \mathcal{P}os(r|_p) \cap \mathbb{N}$  the node  $n'$  is labelled  $(r, p \cdot i, p_u)$  and the edge from  $n$  to  $n'$  is labelled  $i$ ,
2. if  $\text{root}(r|_p)$  is a meta-variable  $Z$ , then the node  $n'$  is labelled  $(s, q \cdot q')$  and the edge from  $n$  to  $n'$  is unlabelled, such that  $l \rightarrow r$  is the rewrite rule employed in  $u$ ,  $q$  is the position of  $u$  in  $s$ , and  $q'$  is the unique position of  $Z$  in  $l$ .

We say that a path is *maximal* if it is not a proper prefix of another path. We write a path  $P$  as a (possibly infinite) sequence of alternating nodes and edges  $P = n_1 e_1 n_2 \dots$ .

**Definition 5.7.** Let  $P = n_1 e_1 n_2 \dots$  be a path of  $s$  with respect to  $\mathcal{U}$ . The path projection of  $P$  is a sequence of alternating nodes and edges  $\phi(P) = \phi(n_1) \phi(e_1) \phi(n_2) \dots$  such that for each node  $n$  in  $P$ :

1. if  $n$  is labelled  $(t, p)$ , then  $\phi(n)$  is unlabelled if  $\text{root}(t|_p)$  is a redex in  $\mathcal{U}$  or a variable bound by some redex in  $\mathcal{U}$  and it is labelled  $\text{root}(t|_p)$  otherwise,
2. if  $n$  is labelled  $(r, p, q)$ , then  $\phi(n)$  is unlabelled if  $\text{root}(r|_p)$  is a meta-variable and it is labelled  $\text{root}(r|_p)$  otherwise.

For each edge  $e$ , if  $e$  is labelled  $i$ , then  $\phi(e)$  has the same label, and if  $e$  is unlabelled, then  $\phi(e)$  is labelled  $\epsilon$ .

*Example 5.8.* Consider the iCRS with the following rewrite rule  $l \rightarrow r$ :

$$f([x]Z(x), Z') \rightarrow Z(g(Z(Z'))).$$

Also, consider the terms  $s = f([x]g(x), a)$  and  $t = g(g(g(a)))$ , the meta-term  $r = Z(g(Z(Z')))$ , and the set  $\mathcal{U}$  containing the only redex in  $s$ . Obviously,  $s \rightarrow t$  is a complete development.

The term  $s$  has one maximal path with respect to  $\mathcal{U}$ :

$$(s, \epsilon) \rightarrow (r, \epsilon, \epsilon) \rightarrow (s, 10) \rightarrow_1 (s, 101) \rightarrow (r, 1, \epsilon) \rightarrow_1 (r, 11, \epsilon) \\ \rightarrow (s, 10) \rightarrow_1 (s, 101) \rightarrow (r, 111, \epsilon) \rightarrow (s, 2)$$

The term  $t$  has one maximal path with respect to  $\mathcal{U}/\mathcal{U} = \emptyset$ :

$$(t, \epsilon) \rightarrow_1 (t, 1) \rightarrow_1 (t, 11) \rightarrow_1 (t, 111).$$

The path projections of the maximal paths are respectively

$$\cdot \rightarrow_\epsilon \cdot \rightarrow_\epsilon g \rightarrow_1 \cdot \rightarrow_\epsilon g \rightarrow_1 \cdot \rightarrow_\epsilon g \rightarrow_1 \cdot \rightarrow_\epsilon \cdot \rightarrow_\epsilon a$$

and

$$g \rightarrow_1 g \rightarrow_1 g \rightarrow_1 a.$$

Let  $\mathcal{P}(s, \mathcal{U})$  denote the set of path projections of maximal paths of  $s$  with respect to  $\mathcal{U}$ . The following result can be witnessed in the above example.

**Lemma 5.9.** *Let  $u \in \mathcal{U}$  and let  $s \rightarrow t$  be the rewrite step contracting  $u$ . There is a surjection from  $\mathcal{P}(s, \mathcal{U})$  to  $\mathcal{P}(t, \mathcal{U}/u)$ . Given a path projection  $\phi(P) \in \mathcal{P}(s, \mathcal{U})$ , its image under the surjection is acquired from  $\phi(P)$  by deleting finite sequences of unlabelled nodes and  $\epsilon$ -labelled edges from  $\phi(P)$ .*

*Proof (Sketch).* By straightforwardly, but very tediously, tracing through the construction of paths, it is evident that the set of maximal paths of  $t$  with respect to  $\mathcal{U}/u$  can be obtained from the set of maximal paths of  $s$  with respect to  $\mathcal{U}$  by replacing or deleting nodes of the form  $(r, p, p_u)$ . If a maximal path of  $t$  is obtained from a maximal path of  $s$  in this way, then they have identical path projections, except that sequences of  $\epsilon$ -labelled edges and unlabelled nodes may have been deleted (due to the contraction of  $u$ ). This establishes the desired surjection. It is easy to see that the sequences of deleted edges can only be infinite if there is an infinite chain of meta-variables in the right-hand side of the rule of  $u$ , which is impossible by definition of meta-terms.  $\square$

We next define a property for sets  $\mathcal{P}(s, \mathcal{U})$ : the finite jumps property. We also define some terminology to relate a term to a set  $\mathcal{P}(s, \mathcal{U})$ .

**Definition 5.10.** *If no path projection occurring in  $\mathcal{P}(s, \mathcal{U})$  contains an infinite sequences of unlabelled nodes and  $\epsilon$ -labelled edges, then we say that  $\mathcal{U}$  has the finite jumps property. Moreover, we say that a term  $t$  matches  $\mathcal{P}(s, \mathcal{U})$ , if, for all  $\phi(P) \in \mathcal{P}(s, \mathcal{U})$ , and for all prefixes of  $\phi(P)$  ending in a node  $n$  labelled  $f$ , we have that  $\text{root}(t|_p) = f$ , where  $p$  is the concatenation of the edge labels in the prefix (starting at the first node of  $\phi(P)$  and ending at  $\phi(n)$ ).*

We have the following.

**Proposition 5.11.** *If  $\mathcal{U}$  has the finite jumps property, then there exists a unique term, denoted  $\mathcal{T}(s, \mathcal{U})$ , that matches  $\mathcal{P}(s, \mathcal{U})$ .*



*Proof.* The proof is identical to the proof of Proposition 12.5.8 in [3].  $\square$

We can now finally prove the Finite Jumps Developments Theorem:

**Theorem 5.12 (Finite Jumps Developments Theorem).** *If  $\mathcal{U}$  has the finite jumps property, then:*

1. every complete development of  $\mathcal{U}$  ends in  $\mathcal{T}(s, \mathcal{U})$ ,
2. for any  $p \in \mathcal{P}os(s)$ , the set of descendants of  $p$  by a complete development of  $\mathcal{U}$  is independent of the complete development,
3. for any redex  $u$  of  $s$ , the set of residuals of  $u$  by a complete development of  $\mathcal{U}$  is independent of the complete development, and
4.  $\mathcal{U}$  has a complete development.

*Proof (Sketch).* The proof is identical to the proof of Proposition 12.5.9 in [3], except that Lemma 5.9 is employed instead of tracing.  $\square$

With the Finite Jumps Developments Theorem in hand, we can now precisely characterise the sets of redexes having complete developments. This characterisation seems to be new.

**Lemma 5.13.** *The set  $\mathcal{U}$  has a complete development if and only if  $\mathcal{U}$  has the finite jumps property.*

*Proof.* To prove that the finite jumps property follows if  $\mathcal{U}$  has a complete development, suppose  $\mathcal{U}$  does not have the finite jumps property. In this case there is a path projection which ends in an infinite sequence of unlabelled nodes and  $\epsilon$ -labelled edges.

By Lemma 5.9 we have for each step  $s \rightarrow t$  contracting a redex in  $\mathcal{U}$  that there is a surjection from  $\mathcal{P}(s, \mathcal{U})$  to  $\mathcal{P}(t, \mathcal{U}/u)$  which deletes only finite sequences of unlabelled nodes and  $\epsilon$ -labelled edges. Hence, for all path projections we have that the nodes and edges left after the contraction of a redex in  $\mathcal{U}$  either stay at the same distance from the first node of the path projection in which they occur or move closer to the first node. But then it follows immediately by ordinal induction that a path projection with an infinite sequence of unlabelled nodes and  $\epsilon$ -labelled edges is present after each development. In particular, such an infinite sequence is present after the complete development. However, by definition of paths and path projections this means that a descendant of a redex in  $\mathcal{U}$  is present in the final term of the complete development. But this contradicts the fact that no descendants of redexes in  $\mathcal{U}$  exist in the final term of a complete development. Hence,  $\mathcal{U}$  has the finite jumps property.

That  $\mathcal{U}$  has a complete development if it has the finite jumps property is an immediate consequence of Theorem 5.12(4).  $\square$

The result we were aiming at now follows easily.

**Theorem 5.14.** *If  $\mathcal{U}$  has a complete development then all complete developments of  $\mathcal{U}$  end in the same term.*

*Proof.* By Lemma 5.13, if  $\mathcal{U}$  has a complete development then it has the finite jumps property. But then each complete development of  $\mathcal{U}$  ends in the same final term by Theorem 5.12(1).  $\square$

## 6 Further Directions

We have defined and proved the first results for iCRSs, but a number of questions that have been answered for iTRSs and  $\lambda c$  remain open: Does there exist a notion of meaningless terms [11] that allows for the construction of Böhm-like trees? Can we prove a partial confluence property [2, 3, 11] showing infinitary confluence up to equivalence of meaningless terms?

Furthermore, can the treatment of iCRS in this paper be extended to the other formats of higher-order rewriting? The fact that CRSs have a clean separation of abstractions (in terms and rewrite rules) and substitutions which is not present in some of the other forms of higher-order rewriting [3] may constitute a stumbling block in this respect.

Finally, it is as yet unclear how to relax the requirement that no infinite chains of meta-variables are allowed in meta-terms while still retaining a meaningful notion of substitution.

## References

1. Dershowitz, N., Kaplan, S., Plaisted, D.A.: Rewrite, rewrite, rewrite, rewrite, rewrite, . . . . TCS **83** (1991) 71–96
2. Kennaway, R., Klop, J.W., Sleep, R., de Vries, F.J.: Transfinite reductions in orthogonal term rewriting systems. I&C **119** (1995) 18–38
3. Terese: Term Rewriting Systems. Cambridge University Press (2003)
4. Kennaway, J.R., Klop, J.W., Sleep, M., de Vries, F.J.: Infinitary lambda calculus. TCS **175** (1997) 93–125
5. Klop, J.W.: Combinatory Reduction Systems. PhD thesis, Rijksuniversiteit Utrecht (1980)
6. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. TCS **121** (1993) 279–308
7. Arnold, A., Nivat, M.: The metric space of infinite trees. Algebraic and topological properties. Fundamenta Informaticae **3** (1980) 445–476
8. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Second edn. Elsevier Science (1985)
9. Hanus, M., Prehofer, C.: Higher-order narrowing with definitional trees. In Ganzinger, H., ed.: Proc. of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA'96). Volume 1103 of LNCS., Springer-Verlag (1996) 138–152
10. van Oostrom, V.: Higher-order families. In Ganzinger, H., ed.: Proc. of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA '96). Volume 1103 of LNCS., Springer-Verlag (1996) 392–407
11. Kennaway, R., van Oostrom, V., de Vries, F.J.: Meaningless terms in rewriting. The Journal of Functional and Logic Programming **1** (1999)

# Multiset discrimination for acyclic data

Fritz Henglein  
DIKU, University of Copenhagen  
henglein@diku.dk



DIKU/IST Workshop, 2005/09/24

## Overview

- Discrimination: Partitioning input into equivalence classes
- Basics: Types, equivalence classes, discriminators
- Top-down MSD for unshared data
- Bottom-up MSD for shared data (briefly!)
- Discussion



DIKU/IST Workshop, 2005/09/24

## Multiset discrimination: The problem

- Partition a sequence of inputs into equivalence classes according to a given equivalence relation
- Examples:
  - Same word occurrences in text
  - Anagram classes of dictionary
  - Equal terms or (sub)trees
  - Equivalent states of finite state automaton
  - Bisimulation classes of labeled transition system
- Note: Generalization of equality/equivalence to from 2 to  $n$  arguments.



DIKU/IST Workshop, 2005/09/24

## Multiset discrimination: The problem...



- Occurs frequently as auxiliary or key step in other problems; e.g.,
  - Compiling:
    - Symbol table management
    - Is there a duplicate identifier in a formal parameter list?
  - Optimization: Replace multiple equivalent data structures by (pointers to) a single data structure
- Is frequently solved by use of hashing, possibly in connection with sorting

DIKU/IST Workshop, 2005/09/24

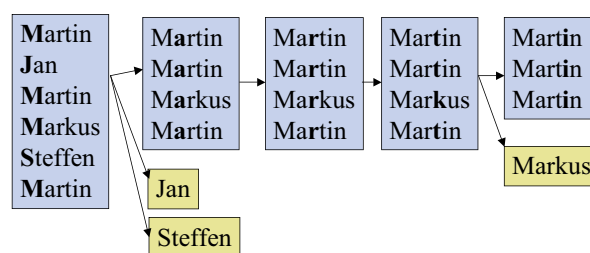
## Multiset discrimination: The techniques



- Worst-case optimal techniques for multiset discrimination without hashing or sorting
- Basic idea (for string discrimination): Partition multiset of strings according to first character, then refine blocks according to second character and so on

DIKU/IST Workshop, 2005/09/24

## MSD: Basic idea



DIKU/IST Workshop, 2005/09/24



## Basics: Values

- Universe  $U$  of first-order values:
  - $v ::= () \mid a \mid \text{inl}(v) \mid \text{inr}(v) \mid (v, v)$
  - $a ::= \langle \text{atomic values from finite set, e.g., characters} \rangle$
- Examples of values:  
 $(\text{'a'}, \text{'b'})$ ,  $\text{inl}(\text{'J'})$ ,  $\text{inl}(\text{'a'})$ ,  $\text{inl}(\text{'n'})$ ,  $\text{inr}()$
- Notation: The latter value is also denoted by  $[\text{'J'}, \text{'a'}, \text{'n'}]$  and "Jan".
- Sizes of values (bit size of untyped representation):
  - $|v, v'| = |v| + |v'|$
  - $|\text{inl}(v)| = |\text{inr}(v)| = 1 + |v|$
  - $|()| = 0$
  - $|a| = O(\log_2 |A|)$ , where  $a \in A$

DIKU/IST Workshop, 2005/09/24



## Basics: Types

- **Type:**  
 A partial equivalence relation (per) on  $U$ ; that is, a subset  $S$  of  $U$  together with an equivalence relation on  $S$
- **Type expressions:**
  - $T ::= 1 \mid T * T \mid T + T \mid A \mid t \mid \mu t. T \mid \text{Bag}(T) \mid \text{Set}(T)$
  - $A ::= \langle \text{atomic type names, e.g., Char} \rangle$
- **Abbreviations:**  $\text{Seq}(T) = \mu t. 1 + T * t$   
 $\text{String} = \text{Seq}(\text{Char})$   
 $\text{Bool} = 1+1$

DIKU/IST Workshop, 2005/09/24



## Basics: Types...

- Each type expression denotes a type:
  - $A$ : primitive values with built-in equality (e.g., characters with character equality)
  - $1$ :  $\{()\}$  with  $() = ()$
  - $T * T'$ :  $\{(t, t') : t \in T, t' \in T'\}$  with canonically induced equivalence
  - $T + T'$ :  $\{\text{inl}(t) : t \in T\} \cup \{\text{inr}(t') : t' \in T'\}$  with canonically induced equivalence
  - $t$ : Type bound to  $t$  in context

DIKU/IST Workshop, 2005/09/24

## Basics: Types...



- continued:
  - $\mu t. T$ : smallest *per*  $X$  such that  $X = T[X/t]$
  - $Bag(T)$ :  $\{ [v_1 \dots v_n] : v_i \in T \}$  where  $[v_1 \dots v_n] =_{Bag(T)} [w_1 \dots w_n]$  if  $v_i =_T w_{\pi(i)}$  for some permutation  $\pi$  for all  $i=1..n$ .
  - $Set(T)$ :  $\{ [v_1 \dots v_n] : v_i \in T \}$  where  $[v_1 \dots v_n] =_{Set(T)} [w_1 \dots w_m]$  if:
    - for all  $i$  there exists  $j$  such that  $v_i =_T w_j$ , and
    - for all  $j$  there exists  $i$  such that  $v_i =_T w_j$ .

DIKU/IST Workshop, 2005/09/24

## Example equivalences:



- Consider the sequence “Jann”. It is an element of  $Seq(Char)$ ,  $Bag(Char)$  and  $Set(Char)$ :
  - As element of  $Seq(Char)$  it is equivalent to “Jann”, but neither “nJan” nor “Jna”.
  - As element of  $Bag(Char)$  it is equivalent to “Jann” and “nJan”, but not “Jna”.
  - As element of  $Set(Char)$  it is equivalent to “Jann”, “nJan”, and “Jna”.
- $[[4, 9, 4], [1, 4, 4], [9, 4, 4, 9], [4, 1]] =_{Set(Set(int))} [[1, 4, 1], [9, 4, 9, 4]]$

DIKU/IST Workshop, 2005/09/24

## Discriminator



- A discriminator for type  $T$  is a function  $D[T]: \forall t. Seq(T^*t) \rightarrow Seq(Seq(t))$  such that, if  $D[T]([l_1, v_1], \dots, [l_n, v_n]) = [V_1, \dots, V_k]$ :
  - $V_1 \dots V_k$  is a permutation of  $[v_1, \dots, v_n]$ ;
  - Iff  $l_i =_T l_j$  then there is a block  $V_h$  that contains both  $v_i$  and  $v_j$ .

DIKU/IST Workshop, 2005/09/24

## Top-down Discrimination



- Polytypic definition of discriminators:
  - $D[T] [(l_1, v_1)] = [[v_1]]$  for any  $T$  (\* Note:  $O(1)$ ! \*)
  - $D[A] \text{ xss} = D_A \text{ xss}$  (given discriminator for  $A$ )
  - $D[1] [(l_1, v_1), \dots, (l_n, v_n)] = [[v_1, \dots, v_n]]$
  - $D[T^*T] [(l_{11}, l_{12}), v_1, \dots, (l_{n1}, l_{n2}), v_n] =$   
 $\text{let } [B_1, \dots, B_k] = D[T] [(l_{11}, (l_{12}, v_1)), \dots, (l_{n1}, (l_{n2}, v_n))]$   
 $\text{let } (W_1, \dots, W_k) = (D[T'] B_1, \dots, D[T'] B_k)$   
 $\text{in concat } (W_1, \dots, W_k)$

DIKU/IST Workshop, 2005/09/24

## Top-down discrimination...



- Polytypic definition contd.:
  - $D[T+T'] \text{ xss} =$   
 $\text{let } (B_1, B_2) = \text{splitTag xss}$   
 $\text{let } (W1, W2) = (D[T] B_1, D[T'] B_2)$   
 $\text{in concat } (W1, W2)$
  - $D[t] \text{ xss} = D_t \text{ xss}$  where  $D_t$  is discriminator bound to  $t$  in context
  - $D[\mu t. T] \text{ xss} = D[T] \text{ xss}$  in context where  $t$  is bound to  $D[\mu t. T]$  (recursive definition!)

DIKU/IST Workshop, 2005/09/24

## Discriminator combinators



- Note that the definitions of  $D[T+T']$  and  $D[T^*T']$  require  $D[T]$  and  $D[T']$  only
- Thus for each type constructor  $^*$ ,  $+$  we can define a corresponding *discriminator combinator*, also denoted by  $^*$ ,  $+$  that compose given discriminators for  $T$ , and  $T'$  to discriminators for  $T^*T'$  and  $T+T'$ , respectively.
- **Note:** Combinators are ML-typable, except for recursively defined ones (require polymorphic recursion)

DIKU/IST Workshop, 2005/09/24

## Example: Sequence discriminator



- $D[\text{Seq}(T)] = D[\mu t. 1 + T * t] =$   
 $= D[1 + T * t]$  with  $t := D[\text{Seq}(T)]$   
 $= D[1] + D[T * t] =$   
 $= D[1] + D[T] * D[\text{Seq}(T)]$
- That is,  $D[\text{Seq}(T)] = f$  where  $f$  is recursively defined:  
 $f = D[1] + D[T] * f$
- E.g.,  $D[\text{Seq}(\text{Char})]$  is the canonical string discriminator.

DIKU/IST Workshop, 2005/09/24

## Discrimination for bags and sets



- We can discriminate for bag equivalence by:
  - sorting the input labels (each of which is a sequence) according to a common sorting order, then
  - eliminating successive equivalent elements (for set equivalence only), and
  - applying ordinary sequence discrimination to the thus sorted sequences

DIKU/IST Workshop, 2005/09/24

## Weak sorting



- Weak sorting sorts each sequence in a multiset according to some common sorting order.
- Basic idea:
  - Associate each element with all the sequences it occurs in.
  - Then traverse the elements and add them to their sequences.
  - In this fashion all sequences will contain their elements in the same order.

DIKU/IST Workshop, 2005/09/24





## Optimal discrimination

- **Theorem:**  $D[T]$  xss executes in time  $O(|xss|)$  for all type expressions  $T$ .
- **Observation:** The discriminators need not always inspect all the input since discrimination stops as soon as a singleton equivalence class is identified.

DIKU/IST Workshop, 2005/09/24



## Applications:

- $D[Seq(Char)]$ : Finding unique words and all their occurrences in a text
- $D[Bag(Char)]$ : Finding the anagram classes of a dictionary (set of words)
- $D[ut. 1 + Bag(t) + (t * t)]$ : Discrimination of simple type expressions under associativity and commutativity of product type constructor in linear time (Zibin, Gil, Considine [2003], Jha, Palsberg, Shao, Henglein [2003])
- $D[ut. (String * Bag(t)) + (String * Set(t)) + (String * Seq(t))]$ : Discriminating terms with associative, associative-commutative and associative-commutative-idempotent operators in linear time (word problem)

DIKU/IST Workshop, 2005/09/24



## Bottom-up discrimination

- Top-down discrimination is optimal for *unshared* data.
- Consider a dag defined by:
 
$$n'_0 = (n_1, n_1), n_0 = (n_1, n_1)$$

$$n_1 = (n_2, n_2)$$

$$\dots$$

$$n_k = ((), ())$$
- Treating this as an element of  $ut. (t+1) * (t+1)$  (trees!) would require time  $O(2^k)$ .

DIKU/IST Workshop, 2005/09/24



## Bottom-up discrimination

- The problem is that shared data (nodes, boxes, references) may occur in multiple calls during top-down MSD.
- Basic idea:
  - Stratify nodes into ranks according to their heights in the dag.
  - Discriminate (partition) *all* nodes of the same rank in one go. Do this in a bottom up fashion since discrimination of rank  $k$  nodes requires discrimination according to rank  $k-1$  nodes.

DIKU/IST Workshop, 2005/09/24



## Bottom-up discrimination

- Extend the type language with  $Box(T)$  (pointers to values of type  $T$  under value equivalence) and  $Ref(T)$  (pointers to values of type  $T$  with pointer equivalence)
- **Theorem:**  $D[T] S$  xss for store (graph)  $S$  and input sequence  $xss$  executes in time and space  $O(|S| + |xss|)$ .

DIKU/IST Workshop, 2005/09/24



## Applications:

- $D[\mu t. Box(Seq(String * t)) * Bool]$ : Minimization of acyclic finite state automata (Revuz [1992], Cai/Paige [1995])
- Construction of Reduced Ordered Binary Decision Diagrams (ROBDD) without hashing (Henglein [2005])
- Compacting garbage collection (Ambus [2004], see plan-x.org)
- Type-directed pickling (Kennedy [2004], Elsmann [2004])
- Compacting garbage collection (Appel/Goncalves [1993])

DIKU/IST Workshop, 2005/09/24



## References (Acyclic MSD):

- Paige, Tarjan, "Three Partition Refinement Algorithms", SIAM J. Computing, 16(6):973-989, 1987 (Section 2: lexicographic sorting)
- Cai, Paige, "Look Ma, no hashing, and no arrays neither", POPL 1991 (applications of string msd)
- Cai, Paige, "Using multiset discrimination to solve language processing problems without hashing", TCS 145(1-2):189-228, 1995 (based on POPL 1991 paper)

DIKU/IST Workshop, 2005/09/24



## References...

- Paige, "Optimal translation of user input in dynamically typed languages", unpublished manuscript, 1991 (weak sorting, bag/set equivalence, bottom-up msd for trees and dags)
- Paige, "Efficient translation of external input in a dynamically typed language", Proc. 13th World Computer Congress, Vol. 1, 1994 (optimal-time preprocessing of serialized input into internal data structures)

DIKU/IST Workshop, 2005/09/24



## References...

- Paige, Yang, "High level reading and data structure compilation", POPL 1997 (underpinnings and refinement of efficient preprocessing)
- Zibin, Gil, Considine, "Efficient algorithms for isomorphisms of simple types", POPL 2003 (application of basic msd to isomorphism with distributivity)

DIKU/IST Workshop, 2005/09/24



## References (Cyclic MSD):

Note: Term "MSD" not used in works below.

Downey, Sethi, Tarjan, "Variations on the common subexpression problem", JACM 1980 (list equivalence in cyclic graph)

Cardon, Crochemore, "Partitioning a graph in  $O(|A| \log |V|)$ ", TCS 1982 (bag equivalence in cyclic graph)

Paige, Tarjan, "Three Partition Refinement Algorithms", SIAM J. Computing, 16(6):973-989, 1987 (Section 3: coarsest partition refinement; set equivalence in cyclic graph)

DIKU/IST Workshop, 2005/09/24

## Conclusions



- Optimal discriminators that can be generated automatically from definition of equivalence relation (can be extended to richer language for equivalence classes)
- Note: No pointers required!
- Practical performance of handcoded MSD typically comparable with hashing (in some cases better)
- References in strongly typed languages can be made discriminable without making them comparable or hashable

DIKU/IST Workshop, 2005/09/24

## Discussion



- MSD techniques (historically for strings and graphs) can be "disassembled" into atomic components ( $*$ ,  $+$ ,  $\mu$ , ...) and then orthogonally combined freely to arrive at assembly of MSD-techniques
- Identification of type of discriminators has been crucial for admitting inductive/polytypic definition of discriminators
- Discriminators stress ML-polymorphism: Reference discrimination (semantically safe side effects, but prohibited by ML reference typing) and discrimination for recursively defined types (polymorphic recursion required)
- Reference discrimination (instead of equality) would be an easy useful extension to ML without performance or semantic penalties, yet support for linear-time discrimination (presently requires  $O(n^2)$  time using reference equality alone).
- Discriminators can be extended to cyclic data at cost of  $\log(n)$  factor. Requires more refined algorithmic techniques.

DIKU/IST Workshop, 2005/09/24

## Open questions



- Automatic generation of *efficient* (not handcoded) discriminators ; e.g., by partial evaluation
- Algorithm engineering: I/O, cache-sensitivity analysis
- Empirical evaluation of MSD in a variety of applications (e.g., ROBDDs, coalescing garbage collection, run-time verification, type checking)
- Identification of scenarios where 'weak' machine model required by MSD is an advantage
- Extension of MSD to scoped values (e.g., alpha-congruence), other extensions

DIKU/ST Workshop, 2005/09/24

## More information



- Paper(s) under preparation [Plan-x.org/msd](http://Plan-x.org/msd)

DIKU/ST Workshop, 2005/09/24

## Natural Numbers Type in Call-by-Value Based on CPS Semantics

Yoshihiko Kakutani  
University of Tokyo  
`kakutani@is.s.u-tokyo.ac.jp`

### Call-by-Name Natural Numbers Type (1/2)

It is well-known that the (simply typed) call-by-name  $\lambda$ -calculus can be extended with a natural numbers type  $\mathbb{N}$ .

$$\frac{\Gamma \vdash \text{zero} : \mathbb{N} \quad \Gamma \vdash \text{succ} : \mathbb{N} \rightarrow \mathbb{N}}{\Gamma \vdash M : A \quad \Gamma \vdash F : A \rightarrow A} \quad \Gamma \vdash \text{it}^A \langle M, F \rangle : \mathbb{N} \rightarrow A$$

$$(\text{it} \langle M, F \rangle)(\text{zero}) = M$$

$$(\text{it} \langle M, F \rangle)((\text{succ})N) = F((\text{it} \langle M, F \rangle)N)$$

### Call-by-Name Natural Numbers Type (2/2)

A natural numbers type is **strong** if the following holds. (Otherwise, we call one a weak natural numbers type.)

If a term  $G$  satisfies  $G(\text{zero}) = M$  and  $G((\text{succ})N) = F(GN)$  for any  $N : \mathbb{N}$ , then  $G = \text{it} \langle M, F \rangle$  holds.

The strongness property implies the induction principle. For example, let  $P$  be  $\text{it} \langle \lambda x. x, \lambda f. \lambda x. \text{succ}(fx) \rangle : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ . Since  $P(\text{zero})M = M$  and  $P((\text{succ})N)M = (\text{succ})(PNM)$  hold,  $P$  is uniquely determined by the strongness and must be the inductive definition of  $+$ .

### Call-by-Value Calculus (1/4)

The aim of this work is to characterize a natural numbers type for call-by-value languages in a similar way to the call-by-name case.

The call-by-value  $\lambda$ -calculus is usually given by **the CPS translation**:

$M = N$  holds in the call-by-value  $\lambda$ -calculus if  $\llbracket M \rrbracket = \llbracket N \rrbracket$  holds in the call-by-name  $\lambda$ -calculus.

### Call-by-Value Calculus (2/4)

The CPS translation is defined inductively as follows:

$$\begin{aligned}\llbracket c \rrbracket k &= kc \\ \llbracket x \rrbracket k &= kx \\ \llbracket \lambda x. M \rrbracket k &= k(\lambda x. \llbracket M \rrbracket) \\ \llbracket MN \rrbracket k &= \llbracket M \rrbracket(\lambda m. \llbracket N \rrbracket(\lambda n. mnk))\end{aligned}$$

### Call-by-Value Calculus (3/4)

The call-by-value  $\lambda$ -calculus defined by the CPS translation is exactly equivalent to the  $\lambda_c$ -calculus, which is an axiomatic calculus introduced by Moggi.

$$\begin{aligned}V &::= c \mid x \mid \lambda x^A. M \\ \text{let } x \text{ be } N \text{ in } M &\equiv (\lambda x. M)N \\ \text{let } x \text{ be } V \text{ in } M &= M[V/x] \\ \lambda x. Vx &= V \\ \text{let } x \text{ be } N \text{ in let } y \text{ be } L \text{ in } M &= \text{let } y \text{ be } (\text{let } x \text{ be } N \text{ in } L) \text{ in } M \\ \text{let } x \text{ be } M \text{ in } x &= M\end{aligned}$$

### Call-by-Value Calculus (4/4)

Side-effects make the difference between call-by-name and call-by-value clear.

For example, while  $(\lambda x. y)(\mu a. [b]z) = y$  holds in the call-by-name  $\lambda\mu$ -calculus,  $(\lambda x. y)(\mu a. [b]z) = \mu a. [b]z$  holds in the call-by-value one. (Informally,  $\mu a. M$  is like `callcc` structure and  $[a]M$  is like `throw` structure.)

Note that the call-by-value  $\lambda\mu$ -calculus, which also can be defined by the CPS translation and axiomatizable, is a conservative extension of the call-by-value  $\lambda$ -calculus.

### Call-by-Value Natural Numbers Type (1/3)

The call-by-value  $\lambda\mu$ -calculus with a natural numbers type has the following syntax.

$$\begin{aligned}
 M &::= \dots \mid \text{zero} \mid \text{suc} \mid \text{it}\langle V, \lambda x. W \rangle \\
 V, W, X &::= \dots \mid \text{zero} \mid \text{suc} \mid (\text{suc})V \\
 &\quad \mid \text{it}\langle V, \lambda x. W \rangle \mid (\text{it}\langle V, \lambda x. W \rangle)X
 \end{aligned}$$

A general form of  $\text{it}\langle M, F \rangle$  is not allowed in order to define the CPS translation properly.

### Call-by-Value Natural Numbers Type (2/3)

A (weak/strong) natural numbers type  $\mathbb{N}$  in the call-by-value  $\lambda\mu$ -calculus is given by the following CPS translation. (The codomain of the CPS translation is the call-by-name  $\lambda$ -calculus with a (weak/strong) natural numbers type.)

$$\begin{aligned}
 &\dots \\
 \llbracket \text{zero} \rrbracket k &= k(\text{zero}) \\
 \llbracket \text{suc} \rrbracket k &= k(\text{suc}) \\
 \llbracket \text{it}\langle V, \lambda x. W \rangle \rrbracket k &= \llbracket V \rrbracket (\lambda v. k(\text{it}\langle v, \lambda x. X \rangle)) \\
 &\quad \text{where } \llbracket W \rrbracket k = kX
 \end{aligned}$$



### Call-by-Value Natural Numbers Type (3/3)

The call-by-value  $\lambda\mu$ -calculus with a natural numbers type can be axiomatized.

$$\begin{aligned}(\text{it}\langle V, \lambda x. W \rangle)(\text{zero}) &= V \\ (\text{it}\langle V, \lambda x. W \rangle)((\text{suc})X) &= (\lambda x. W)((\text{it}\langle V, \lambda x. W \rangle)X)\end{aligned}$$

The strongness property is also restricted to values:

If a value  $U$  satisfies  $(\lambda y. U)(\text{zero}) = V$  and

$(\lambda y. U)((\text{suc})X) = (\lambda x. W)((\lambda y. U)X)$  for any value  $X : \mathbb{N}$ , then  $\lambda y. U = \text{it}\langle V, \lambda x. W \rangle$  holds.

### Categorical Semantics (1/3)

It is known that **control categories** provide a sound and complete class of models for the call-by-value  $\lambda\mu$ -calculus. (Note that the call-by-value  $\lambda$ -calculus is also sound and complete for control categories.)

The subcategory of a control category that consists of effect-free morphisms is called its **focus**. An effect-free morphism is called a **focal map**.

A term  $M$  is **focal** if

$\text{let } x \text{ be } M \text{ in let } y \text{ be } N \text{ in } L = \text{let } y \text{ be } N \text{ in let } x \text{ be } M \text{ in } L$  for any  $x, y, N$  and  $L$ .

Any value is focal but a focal term is not necessarily a value.

### Categorical Semantics (2/3)

The call-by-value  $\lambda\mu$ -calculus with a (weak/strong) natural numbers type is sound and for control categories with (weak/strong) natural numbers types on their focuses.

(Precisely speaking, natural numbers types should be **parameterized**. The details of the parameterization can be found in another work with Hasegawa.)

For the completeness, there are two possible ways:

Modifying the CPS semantics and modifying the categorical semantics.

### Categorical Semantics (3/3)

A subcategory of the focus that satisfies the appropriate conditions is called a **value category**.

Value categories are not unique for a control category and the focus itself is a value category.

Control categories with (weak/strong) natural numbers types on their value categories provide a sound and complete class of models for the call-by-value  $\lambda\mu$ -calculus with a (weak/strong) natural numbers type.

(Note that the parameterization still requires consideration of the focuses.)

### Further Work

Since values and focal terms coincide in function types, we hope to fill a gap between values and focal terms. (However, we have some negative answers.)

A natural numbers type is a kind of so-called **inductive type**.

While an inductive type is characterized by an **initial algebra** in the call-by-name  $\lambda$ -calculus, such a type is characterized by an initial algebra on values in the call-by-value calculus.

### Conclusion

A natural numbers type for call-by-value languages is defined by the CPS translation.

A sound and complete axiomatization of the call-by-value  $\lambda$ -calculus with a natural numbers type is given.

It has been shown that the call-by-value  $\lambda\mu$ -calculus with a natural numbers type is sound and complete for the categorical semantics.

The above results can be generalized to general inductive types.

# Verification of Liveness Properties

Carl Christian Frederiksen  
IST  
University of Tokyo

September 24, 2005

## Overview

- Predicate abstraction: *the need for stronger methods for liveness properties*
- Size-change termination
- A method for verification of liveness properties (*as opposed to safety properties*)
- Improvements of the verification method
  - Scalability
  - Precision

## Background

- *Podelski & Rybalchenko*: A method for verification of liveness properties
- Extends predicate abstraction to *transition predicate abstraction*
  - The termination property can be retrieved from the node labels and
  - The fairness assumptions from the edges labels
- Abstraction preserves the property to be verified, in particular:
  - the termination property
  - the fairness assumptions

## Predicate Abstraction

- Idea: partition the state space into a *finite* set of equivalence classes (abstract states) using predicates over states
- Abstract program represented by a labeled abstract state program
- Limited to safety properties
- The *Augmented Abstraction Framework* can be used to annotate the finite-state abstraction by progress monitors
  - Drawback: requires manual construction of ranking fens

### Example: Readers/Writers

- Shared database access with two process types:
  - Readers
  - Writers
- Number of processes is unbounded
- Access protocol
  - A lock must be acquired before the resource can be accessed
  - Concurrent read access is permitted, but
  - Write access is exclusive
- Prove: Write requests are eventually granted, assuming *fair scheduling*

#### Readers/Writers

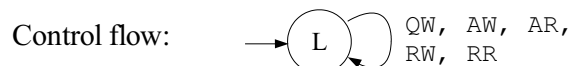
```

r: integer [0,1,...] // read locks
w: boolean          // write lock
q: integer [0,1,...] // pending writers

r:=0;
w:=F;
q:=0;

while(T)
@L: [] -> QW: q:=q+1
    [] r=0 && !w && q>0 -> AW: w=T; q:=q-1
    [] !w && q=0 -> AR: r:=r+1
    [] w -> RW: w:=F
    [] r>0 && (w || q>0) -> RR: r:=r-1

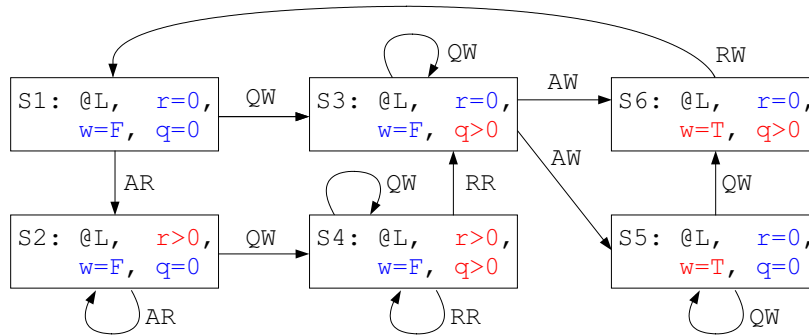
```



## State Predicate Abstraction

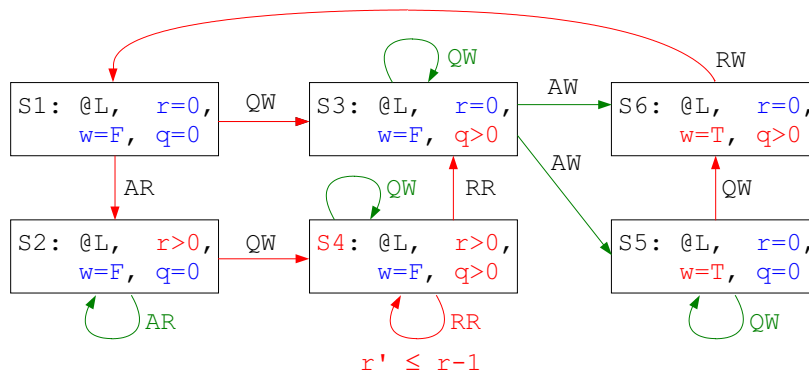
```

q, r: integer [0, 1, ...], w: boolean
while(T)
@L: []
    [] r=0 && !w && q>0 -> QW: q:=q+1
    [] r=0 && !w && q>0 -> AW: w=T; q:=q-1
    [] !w && q=0 -> AR: r:=r+1
    [] w -> RW: w:=F
    [] r>0 && (w || q>0) -> RR: r:=r-1
    
```



### Problem: How to Verify the Claim?

- Claim:  $\text{@QW} \Rightarrow \text{AF}(\text{@AW})$
- A possible solution:
  - Show that computation cannot end in  $\text{AR}^\omega$  or  $\text{QW}^\omega$  (unfair)
  - Add a progress monitor for the RR transition from S4 to S4



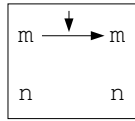
## Size-Change Termination

- Podelski's method for verification of liveness properties has strong similarities to *Size-Change termination* (Lee, Jones, Ben-Amram)
- SCT: A principle for conservatively deciding termination of 1. order function programs with well-founded data
- Observe how the size of parameters relate over function calls:
  - If all infinite flow-legal call sequences have infinite descent in a chain of size relations, then the program terminates for all input

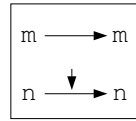
## Size-Change Termination

$a(m, n) =$  if  $m=0$  then  $n+1$  else  
 if  $n=0$  then **1**:  $a(m-1, 1)$   
 else **2**:  $a(m-1, 3$  **3**:  $a(m, n-1)$ )

Size-change graphs

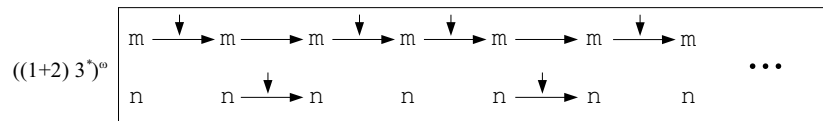
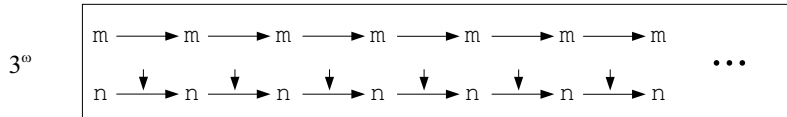


G1,2: call 1 and 2



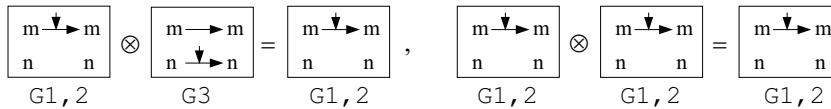
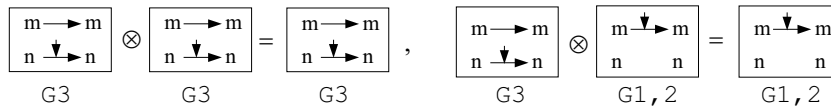
G3: call 3

- Two types of infinite call sequences:  $3^\omega$  and  $((1+2)3^*)^\omega$

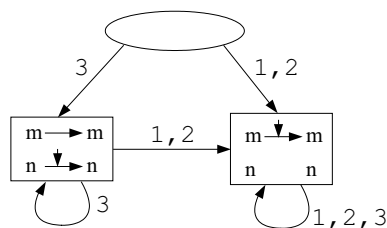
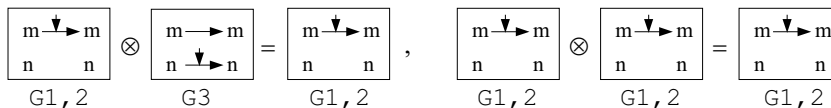
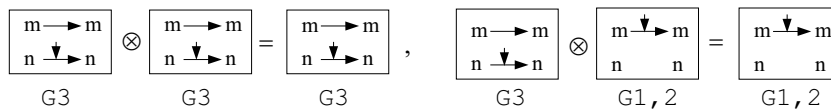


## Abstracting Call Sequences

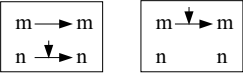
- Define a composition operator  $\otimes$ :
  - $G1 \otimes G2$ : has arc  $x \rightarrow z$  if  $x \rightarrow y \in G1$  and  $y \rightarrow z \in G2$
  - $G1 \otimes G2$ : has arc  $x \downarrow z$  if  $x \xrightarrow{a} y \in G1$  and  $y \xrightarrow{b} z \in G2$  and  $(a = \downarrow \text{ or } b = \downarrow)$



## Relating Call Sequences to Their Abstractions



## Detecting Size-Change Termination

- Closure set under  $\otimes$ : 
- The set size-change graphs is finite
- Suppose  $cs=c_1, c_2, \dots$  is an infinite call sequence
  - For any pair  $(i, j)$  there exists a SCG abstracting  $c_i, \dots, c_j$
- The Infinite Ramsey Theorem:
  - There exists a sequence:  $I = i_1, i_2, \dots$  such that
  - For any  $a, b \in I$ : such that  $a < b$  :
  - $c_a, \dots, c_b$  is abstracted to the same SCG
- Implication: the tail behavior of any infinite call sequence can be abstracted by a single SCG.
- To check termination:
  - verify all idempotent SCGs have in-situ descent.

## Verification of Liveness Properties

- Prove that *undesired* infinite computation does not occur
- Use the Size-Change Termination principle to rule out such traces
  - Compute size relations for the variables over *transitions*
  - If there exists a non-increasing chain of size relations with infinitely many strict decreases, then the corresponding data value cannot be bounded

## Transition Predicate Abstraction

- Termination argument lies not in the states but in the transitions
- Use transition predicates (binary relation on states) instead of state predicates
  - Abstraction: states  $\rightarrow$  state relations
- Abstract program is represented as a graph:
  - Nodes: represent abstract transition sequences
  - Edges: maps concrete transition seqs. to their abstractions

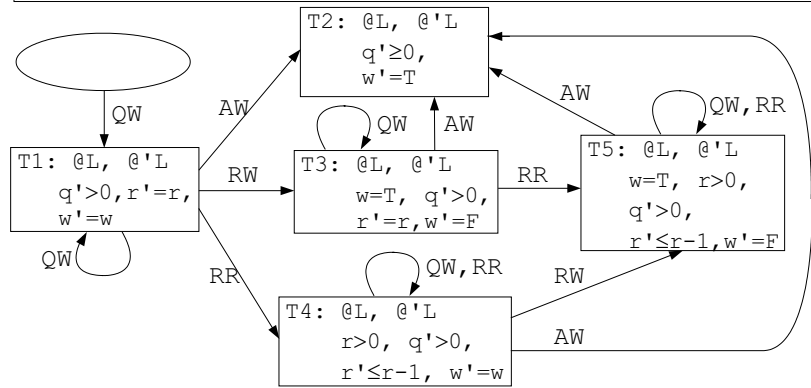
# Verification Procedure

1. Reduce the liveness property + any fairness assumptions to a path condition
2. Apply transition predicate abstraction to subject program to obtain a node-labeled, edge labeled graph
3. Mark all nodes in the abstraction which satisfy the path condition
4. Termination check: Check whether all marked nodes with a self-loop represent finite transition sequences
5. Return "property verified" if each marked node passes the termination check

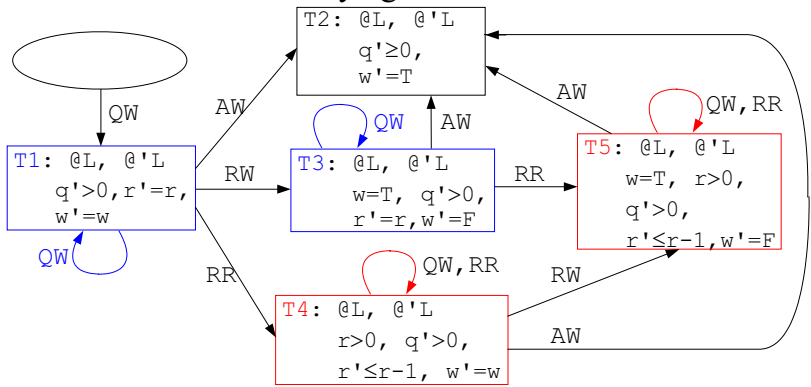
## Transition Predicate Abstraction

```

q, r: integer [0,1,...], w: boolean
while(T)
@L: []
    [] -> QW: q:=q+1
    [] r=0 && !w && q>0 -> AW: w=T; q:=q-1
    [] !w && q=0 -> AR: r:=r+1
    [] w -> RW: w:=F
    [] r>0 && (w || q>0) -> RR: r:=r-1
    
```



## Verifying the Claim



- Claim:  $@QW \Rightarrow AF(@AW)$
- Fairness assumption:
  - $\forall t \in \{QW, AW, AR, RW, RR\} : AG(enabled(t) \Rightarrow AF(t))$
- Marked nodes: **T4** and **T5**
  - Enabledness for the **QW** transitions in **T1** and **T3** have overlapping enabled with other outgoing transitions (unfair)
- Both **T4** and **T5** are terminating loops  $\Rightarrow$  property verified

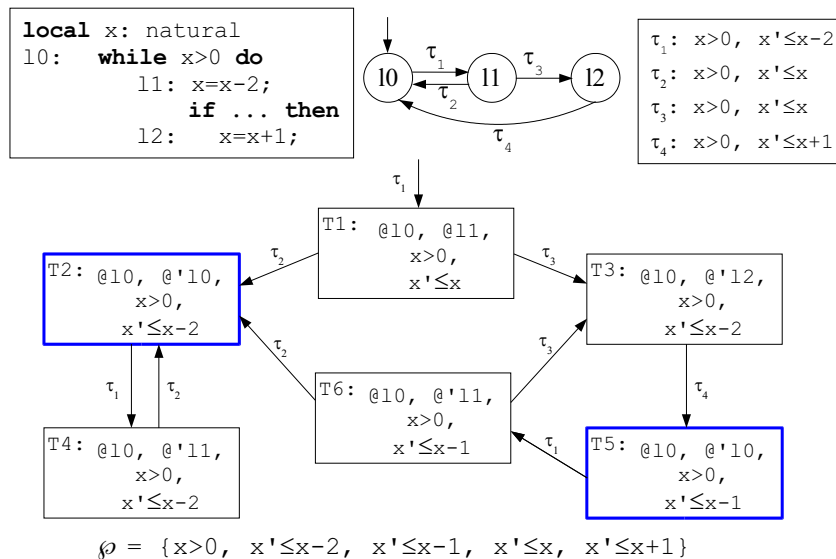


# Improvements of the Transition Predicate Abstraction

## Improvements of the Core Method

- The closure set computation has poor scalability
- Simple steps to improve efficiency:
  - Collapse linear transition sequences prior to the closure set computation
  - Partition the flowgraph into strongly connected components
- Improving precision:
  - Can handle non-monotonous ranking fcn's by using:
  - Abstract values:
    - $x' \leq x-n, \dots, x' \leq x-1, x' \leq x, x' \leq x+1, \dots, x' \leq x+n$
  - But to ensure convergence and for the main thm to hold, the abstract domain must be finite

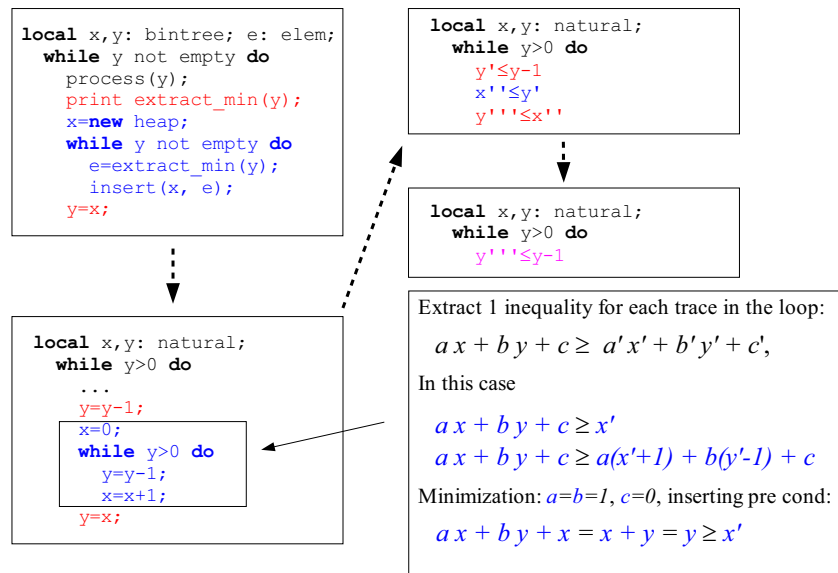
## Non-monotone Descent



# Well-nested Loops

- Process the flow-graph starting with the most deeply nested loops
- The transitions define a set of particular paths (expn in # of conditionals)
- Since the a finite abs domain is not needed at this point, greater precision can be used
- Analyze the loop in the usual manner
- Rather than using the closure set to represent the loop, a stronger abstraction can be used
  - e.g. for uncovering “immaterial invariants”

## Immaterial Invariants



## Improvements: General Case

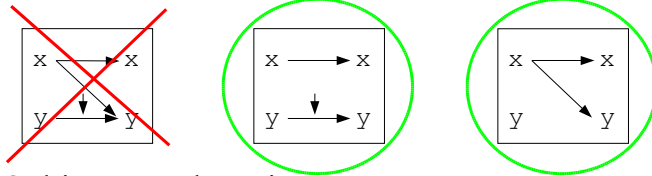
- Select a subgraph of the SCC s.t. the component is still a SCC if the subgraph is replaced with one or more edges
- Analyze the sub graph
- Compute abstractions for traces
  - Starting at an inbound edge
  - Exiting at an outbound edge
- Possible criterion: minimal cut
- Side effect: counter examples become more localized

## PTIME Approximations

- Two PTIME approx. of closure set comp (*C.S.Lee*), predicate set of form:  $x \geq C$ ,  $x' \leq y$ ,  $x' \leq y-1$

- Quadratic

requires abstract transitions to be "fan-in" free



- Qubic, general version

- Precise results in practice

## Conclusion

- Presented an extension of predicate abstraction to transition predicate abstraction
- Automated method for verification of liveness properties under (strong+weak) fairness assumptions
- Improved closure set computation:
  - Better scalability in SCCs
  - Handles non-monotone descent and
  - Captures immaterial size relations over sub SCCs
  - PTIME approximations



## Author Index

- Akashi, Miyoko, 1  
Andersen, Jesper, 80  
Avery, James, 95
- Bohr, Nina, 109
- Castagna, Giuseppe, 79
- Elsborg, Ebbe, 80
- Filinski, Andrzej, 118  
Frederiksen, Carl Christian, 155  
Frisch, Alain, 79
- Hagiya, Masami, 18  
Henglein, Fritz, 80, 139  
Hosoya, Haruo, 79  
Hu, Zhenjiang, 24, 38, 111
- Jones, Neil D., 14
- Takehi, Kazuhiko, 38, 111  
Kakutani, Yoshihiko, 150
- Ketema, Jeroen, 124
- Lawall, Julia, 36  
Liu, Dongxi, 38
- Matsuzaki, Kiminori, 30  
Mogensen, Torben Ægidius, 54  
Moriyama, Akimasa, 111  
Muller, Gilles, 36
- Nakano, Keisuke, 63  
Nissen, Michael, 50
- Olsen, Jørgen, 3
- Simonsen, Jakob Grue, 80, 124  
Skelboe, Stig, 11  
Stefansen, Christian, 80
- Takeichi, Masato, 7, 38, 111
- Wang, Hao, 38



After Work  
Copenhagen, Nyhavn