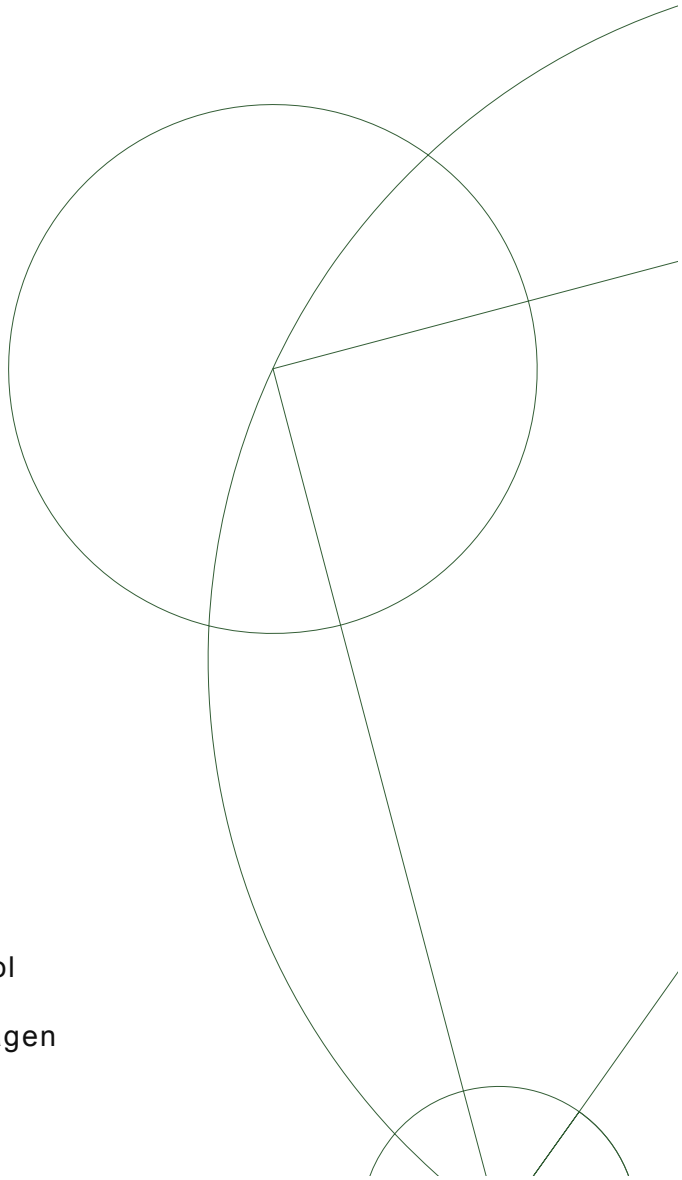
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Ph.D. Dissertation

Anders Starcke Henriksen

Adversarial Models for Cooperative Interactions



This thesis has been submitted to the PhD School

of The Faculty of Science, University of Copenhagen

Abstract

Existing models for concurrent and distributed systems are *cooperative* by nature. In a cooperative setting, all participants implicitly or tacitly work together towards a common goal. We propose to employ a fundamentally *adversarial* viewpoint instead. In an adversarial setting there is no a priori notion of common goal; instead, each participant may pursue individual goals unknown to the other participants. By extension, this means that every participant will behave as he pleases, unless there are special incentives for choosing certain behaviour. To formally capture these incentives, each participant enter into legally binding *contracts* that specify how each participant is rewarded or penalised for certain behaviour. Such an adversarial model is more robust in the presence of failures, because each participant has to account for the other ones defecting from the intended interaction.

With the adversarial perspective in mind, we study verification and certification of concurrent and distributed systems. Concretely, we propose two mathematical formalisms (one using traces and one using automata) based on *zero-sum, real-time games*. We define how a strategy (e.g., a program) conforms with a set of contracts. We also show how to compose strategies, in order to build a conforming strategy from substrategies. For certification, we introduce a framework called *verification-time monitoring* that allows the certification of communicating real-time processes using a standard method like Floyd-Hoare logic.

Lastly, we explore focused proof systems in two ways. Extending on work by Nigam and Miller, we show how to use a focused proof system for intuitionistic logic as a hosting framework for other proof systems. Their work exploited the structural properties of linear logic for the encodings of the object systems; our work shows that linearity is not needed – making focusing the main component for these encodings. We also show how to remove contraction from a focused system for classical propositional logic, which, to our knowledge, has not been done before.

Resumé (Danish abstract)

Eksisterende modeller for samtidige og distribuerede systemer er *kooperative* af natur. For et kooperativt scenarie gælder, at alle deltagere implicit eller stiltiende arbejder mod et fælles mål. Vi anbefaler at man i stedet anlægger et grundlæggede *adversativt* synspunkt. I et adversativt scenarie er der ikke noget på forhånd givet fælles mål; hver deltager søger i stedet individuelle, for de øvrige deltagere ukendte, mål. Følgeligt betyder dette, at hver deltager agerer autonomt, med mindre der er særlige encitamenter til at agere efter et bestemt mønster. For at indfange encitamenterne formelt, indgår deltagerne i juridisk bindende kontrakter. Disse specificerer, hvorledes hver deltager bliver enten belønnet eller straffet for bestemte former for ageren. En sådan adversativ model er mere robust i situationer, hvor der kan opstå fejl, fordi hver deltager er nødt til at tage højde for, at de øvrige deltagere ikke følger den tilsigtede interaktion.

Med det adversative perspektiv i tankerne, undersøger vi verifikation og certifikation af samtidige og distribuerede systemer. Konkret fremlægger vi to matematiske formalismer (en benyttende *traces* og en benyttende automater) baserede på nul-sums, realtids spil. Vi definerer, hvordan en strategi (f.eks. et program) konformerer mod en mængde af kontrakter. Vi viser derudover, hvordan strategier kan sættes sammen, og hvordan en konformerende strategi kan konstrueres udfra konformerende delstrategier. Til certifikation introducerer vi et paradigme, som vi kalder *verification-time monitoring*. Dette paradigme tillader en standardmetode som Floyd-Hoare logik at blive brugt til certifikation af kommunikerende realtidsprocesser.

Tilsidst undersøger vi *focused* bevissystemer på to måder. Vi bygger videre på arbejde af Nigam og Miller og viser hvordan et focused bevissystem for intuitionistisk logik kan bruges som værtssystem for andre bevissystemer. Deres arbejde udnyttede linær logiks strukturelle egenskaber til indkodningerne af objektsystemerne. Vores arbejde viser, at linearitet ikke er nødvendigt, hvilket gør focusing til den primære komponent. Vi viser også hvordan man fjerner *contraction* fra et focused bevissystem for klassisk propositions logik. Vi har ikke kendskab til, at dette er blevet gjort før.

Acknowledgements

First and foremost, I want to thank my main advisor Andrzej Filinski. He has been an inspiration with his high quality standards, and greater dedication and perseverance is hard to think of. Wading through scores of unfinished drafts, Andrzej's comments were always precise and insightful, and they helped improve this work immensely.

Thanks also goes to my co-advisor Carsten Schürmann, who helped with general matters, especially with organising my stay abroad. On that matter, I also want to thank Dale Miller who was a great host during my stay in Paris. Both in terms of academic guidance and general hospitality.

For handling administrative matters swiftly and efficiently, in such a way that they are barely noticed, I thank the group secretary of the APL-group at DIKU, Jette Giovanni Møller. I also thank the rest of my colleagues in the APL-group for creating a good working environment.

On a more personal matter I thank the 'lunch club' for creating a good social environment for me, both at work and after work. Especially Tom Hvitved for putting up with my endless stream of complaints in the final months.

The most heartfelt thanks goes to Irene, for providing comfort and love during my times of wild frustration, and for joining me on my stay abroad in Paris. Without her I would never have succeeded.

Anders Starcke Henriksen, Copenhagen 2011

Contents

Chapter 1

Introduction

1.1 Context

This work was done as part of the project *Trustcare: Trustworthy Pervasive Healthcare Services* [40]. The working title of this part was: *Foundations for certified code for concurrent and distributed processes.* Part of the project goals is that the systems considered should not only be restricted to communicating programs, but broad enough to encompass settings like formalised work processes, in order to better support changing demands. While investigating certification of such systems, we noticed that existing frameworks for orchestrating multiparty interactions did not adequately account for participants having different objectives. This deviation from the real world spurred us to rethink the foundations for program certification, especially in the context of distributed systems.

1.2 Introduction

This dissertation's title might look self contradicting at first, as 'adversarial' and 'cooperative' seem to be incompatible concepts. In this work, we seek to illustrate that purely cooperative scenarios are very rare, and that most real-world scenarios contain elements of adversariality, even though they might seem purely cooperative at first glance. Perhaps a more precise title would have been: *Adversarial models for (seemingly) cooperative interactions.*

This work is concerned with specification, verification and certification of concurrent and distributed *interaction*-based systems. An interaction happens when several *parties* perform certain *actions* affecting each other. Formal *specifications* have been applied to describe and analyse such interactions, ultimately to make sure they execute successfully. Examples of such specifications include communication protocols, service orchestration/choreographies, organisational workflows and program/code contracts. A common property amongst existing methods for modelling such interaction specifications is that they tend to assume (implicitly) that all parties *cooperate* towards a common goal, even though their reasons for wanting to obtain the goal might be different. This assumption of cooperation means that each party is expected to resolve ambiguities or underspecification, in a way that conforms with or supports the common goal.

In this work we argue that such a cooperative methodology, while easier to use (because of a stronger assumption about the participants), is not sufficient for most real-world examples, especially when wishing to support changes. We instead advocate

the use of a fundamentally *adversarial* model, in which each participant operates autonomously in accordance with the participant's own goals, and may at any time defect from the intended path of a specification, or even break it altogether, if this is more beneficial in terms of the participant's own goals. A participant may still follow a specification, if the penalty for breaking it, or the reward for following it, is big enough. Such a notion of cooperation is more robust, as it allows trusted and untrusted parties to cooperate uniformly.

The main challenge, when trying to adopt an adversarial methodology, is to cope with the fact that each participant can defect at any point, and therefore the specifications need to account for this behaviour, which in most cases is non-trivial to do. Furthermore, the extra information needed to find out what should happen in case of failures/defects is usually not present in current models, because such failures are assumed to be non-occurring. To get this extra information, one therefore needs to go back to the source of the specification. For example, in the context of program specification, partial correctness and "eventually" guarantees are not sufficient for adversarial applications; one could always loop in the partial case, and answer after a million years for the eventually guarantees. One needs to find out which original *deadlines* were in place and include them in the model. This further implies that an adversarial model needs to take time (at least in the form of deadlines) into account.

To be able to capture the competitive notion of adversarial specifications, we propose to use the well known concept of a *game* as an abstract basis for such specifications. Games are able to capture competition between different parties, called *players*, and the possibility of penalties and rewards in a familiar manner. From this abstract basis, we develop mathematical models for expressing adversarial specifications, in which each interaction specification is considered a *contract* between the parties, specifying the rules of their game. In general, each player enters into multiple contracts with potentially different other players, which means that it might be rational to lose one game, if this enables winning another game with higher stakes.

In addition to formalising interaction specifications, we also give a model for *implementations*, i.e., means of fulfilling specifications. Such an implementation could be a program, a process or a workflow. In general, each player tries to implement the specifications in a *sound* way; this corresponds to finding a *winning strategy* for the totality of all games this player is engaged in. Such a strategy must take the different rewards and penalties into account, and a sound one will never obtain larger penalties than rewards. In the modelled world this amounts to ensuring that high-assurance services are not relying on low-assurance services.

A desirable property when trying to develop formally verified or certified implementations is *compositionality*, meaning that a sound implementation can be built from sound subimplementations. We show how to compose implementations in the model, and we prove a theorem expressing compositionality as defined above. This works as long as the interaction between the subimplementations can be expressed as a contract, in which the implementations play opposite roles.

Having defined a semantic notion of soundness for implementations, we develop a syntactic method for proving soundness of implementations, which enables us to certify concrete programs. We use a special certification paradigm, which we call *verification-time monitoring*. The general idea is to formalise the semantic notion of soundness as a *test harness* program, for which error-free execution corresponds to soundness of the original implementation. This approach reduces the problem of specification soundness for a communicating program to prevention of error states in the test harness program,

which can be tackled by standard means. As an example, we show how to use an essentially standard Floyd-Hoare logic to certify real-time behaviour of a communicating process.

As part of the investigation of certification, we look at ways of representing proof systems. In this work we show how to use a focused intuitionistic proof system, called LJF, to host several different proof systems. That the system is intuitionistic is of special interest, as we think that an intuitionistic approach to Floyd-Hoare logic might give a better account for partial operators – although we leave that investigation to future work. As a side result of general interest, we show how to remove contraction from another focused system; the classical system LKF. This is interesting because contraction can lead to redundant proofs and because it is the only focused system without contraction we know of.

1.3 Contributions

The main contributions of this dissertation are:

1. We give a thorough treatment of the adversarial methodology. This treatment includes the basic terminology, together with the advantages and disadvantages of the methodology. We compare it with existing cooperative frameworks, and further elaborate on why we think that those approaches are insufficient. We present concrete examples to support this claim.

2. We develop two concrete mathematical models for adversarial interactions. Each model is conceptually a *zero-sum real-time* game between a number of players. The first model is based on traces, and the second is based on automata. Both models include a notion of conformance, which corresponds to a implementation fulfilling a set of specifications. For the trace-based model, we show how to compose implementations, and we prove a theorem that expresses how conformance is preserved during composition. This allows a complete implementation to be built incrementally. The automaton-based model is more executable, and we prove that conformance of an automata implementation implies conformance of its trace-based denotation. As work-in-progress, we sketch how to extend the model with support for linear resources as a primitive concept, allowing clearer models of physical goods and a dynamically changing communication topology.

3. We develop a simple concrete language to express implementations and specifications. Then we show how to certify conformance in concrete cases, using a method which we call verification-time monitoring. Certification is performed by translating the semantic conformance relation to a syntactic test harness program, for which error-free execution corresponds to conformance. This allows standard techniques, like Floyd-Hoare logic, to be used for certifying time-sensitive properties.

4. We show that an intuitionistic focused proof-system can be used to host several different proof-systems, extending on a previous result for focused linear logic. This demonstrates that linearity is not essential, and that focusing is the main ingredient for making this approach work. As part of a further investigation into focused proof-systems, we show how to remove contraction from a classical fo-

cused proof-system, which, to our knowledge, has not been done before due to the focusing part.

1.4 Overview of the dissertation

The dissertation is organised into two main parts and a conclusion.

1.4.1 Part I: Adversarial models of distributed scenarios

Chapter 2: contains the treatment of the adversarial methodology, including comparison to a cooperative methodology and examples illustrating the concepts.

Chapter 3: develops concrete mathematical models for the adversarial methodology. This includes definitions of both implementations and specifications. Furthermore, we define a notion of conformance and show how compositionality of implementations can be obtained.

Chapter 4: develops a certification paradigm called verification-time monitoring, and shows how to use it for certification in our model.

Chapter 5: shows preliminary work on extending the basic model with linear resources, allowing modelling of physical goods and dynamic communication setups.

1.4.2 Part II: Aspects of focusing

Chapter 6: contains a modified version of a tech-report [37], demonstrating how to use the focused system LJF to encode several different proof-systems.

Chapter 7: contains a manuscript for a paper, describing how to remove contraction from the propositional fragment of the focused system LKF for classical logic.

1.4.3 Part III: Conclusion and future work

Chapter 8: contains the conclusion, summary and directions for future work.

Part I

Adversarial models of distributed scenarios

Chapter 2

An adversarial approach to cooperation

The core goal of this thesis is a methodology for verifying and certifying concurrent and distributed interaction patterns and programs. For sequential programs, verification and certification are reasonably well understood and based on Floyd-Hoare logic [30, 45]. When verifying or certifying concurrent programs, standard pre- and post-conditions have to be generalised from input-output specifications to *communication* specifications. Input then corresponds to what is sent *to* a process and output to what is sent *from* a process. Furthermore, the specification in general needs access to the entire communication history. While this extension is technically quite challenging, and has sparked a great deal of research, it does not prompt a fundamental change in the specification paradigm. For distributed systems, the field is much more fragmented, and even the definition of distribution is not clearly agreed upon.

We classify distributed systems along two dimensions. The first dimension is the *physically* distributed systems. For such systems each component is assumed to be physically separated from the others. This separation is usually captured by an underlying imperfect communication model. In such a model, messages may take time to deliver, they may be modified during transfer, or they may be lost altogether. This type of distributed systems have already been studied in the past. The second category is the *logically* distributed systems. For such systems, each component is controlled by potentially different entities. Such entities might have different, possibly conflicting, agendas. From a verification and certification viewpoint, this administrative segregation implies that each component could fail independently of the others. Also, failures are much more likely to be Byzantine; messages can not only be lost or delayed in transmission, but may be incorrect (perhaps even deliberately so) to begin with. An example, of a logically but not physically distributed system, is mobile code, where a user downloads an application from an external source and runs it together with the user's other applications. In such a situation, the new code will often enjoy fast, reliable communication with its new environment, but because it was developed by another party, it is not guaranteed to be well-behaved.

To our knowledge, logically distributed systems have not been extensively studied from the viewpoint of verification, and, in particular, certification. In fact, we have not found a single example, where this distinction played a critical role. In this work, we argue that such distributed systems often exist in reality, and in order to handle them in the context of program verification or program certification, one has to apply a

fundamentally different specification paradigm.

2.1 The cooperative world

We start this section by describing three application domains. We will focus on these domains when analysing the foundation for formal specifications for logically distributed systems.

Programming-by-contract

Programming-by-contract (PBC), also called design-by-contract [58], is a method for facilitating modular construction of software by means of formal specifications. The prototypical scenario is a code producer tasked with the development of a program satisfying some specification. The development task is broken down into several modules, each with a formally specified interface, typically with *pre-* and *postconditions* for entering and leaving the module's code. Several different programmers can now work on different modules, and then if all developed modules satisfy their specifications, the full program will also satisfy its specification. Part of the appeal of PBC stems from its compositional nature, and is a good example of how formal specifications can be applied to programs to help ensure desirable properties. We consider a simple example:

Example 2.1.1. A programmer is tasked with the implementation of a module for calculating the fourth root of a non-negative number. As a help, a square root module is provided. To apply the PBC method, the square root module has the following pre- and postconditions (assuming input is taken from x and output given as in y):

$$\text{pre: } \{x \geq 0\} \quad \text{post: } \{y \cdot y = x\}.$$

The fourth root module has similar conditions:

$$\text{pre: } \{x \geq 0\} \quad \text{post: } \{y \cdot y \cdot y \cdot y = x\}.$$

It is now up to the implementer to find a correct implementation satisfying these constraints [1].

Communication protocols

In a communication protocol scenario, several parties seek to perform some set of actions in a predefined way. The protocol captures the expected interactions of each party. As a concrete scenario, we consider the order negotiation part of a buyer/seller example. The example comes from work on formalising business protocols by Carbone et al. [13], and stems from a use-case in the WS-CDL primer [87]:

Example 2.1.2 (Simple business protocol)**.** The protocol specifies how a buyer obtains delivery details for a purchase. The expected interaction is as follows:

1. Buyer asks Seller, through a specified communication channel, to offer a quote (the item to buy is fixed).

[1] A careful reader will notice that nothing is forcing the output of the square root module to be non-negative, and therefore the square root module cannot be used twice – we elaborate on this later.

2. Seller replies with a quote.

3. Buyer answers seller with either acceptance of the quote or rejection of the quote. If Buyer rejects the quote, the protocol terminates. If Buyer accepts, then Seller sends a confirmation to Buyer and forwards the address of the Buyer to the Shipper.[2]

4. Shipper sends the delivery details to Buyer, and the protocol terminates.

Two extensions to this simple interaction are also proposed:

- If the quote is too high, Buyer may ask for another quote until it receives a satisfactory quote.
- If Buyer does not reply within 30 seconds after Seller presents a quote, then Seller will abort the transaction. Once Seller decides to do so, even if a message arrives from Buyer later, it is deemed invalid.

Organisational workflows

Organisational workflows are a means of formally specifying how a specific work task should be performed, especially in the presence of multiple actors. Such workflows capture general requirements and actual implementation of those requirements. We present two concrete examples, both taken from recent work elsewhere in the TrustCare project. The first example is based on the work by Hildebrandt, Mukkamala and Slaats [43]:

Example 2.1.3 (Trade union)**.** The parties in the scenario are a Danish trade union, *Landsorganisationen i Danmark* (LO), which is the main organisation for Danish trade unions, and *Dansk Arbejdsgiverforening* (DA), the main organisation for Danish employer organisations. The interaction captured is that a trade union can create an employee complaint case, on the request of a member. The creation of the case must be followed by a meeting between the trade union, LO, and DA, arranged by LO. The expected interaction is as follows:[3]

1. The union creates a case and notifies both LO and DA.

2. After the case is created, LO can and must arrange a meeting between a union case worker, a LO case worker and a DA case worker. The meeting should be arranged in agreement between LO and DA (the union is not involved in this negotiation):

   (a) LO always proposes meeting dates to DA first.

   (b) DA should accept, but can also propose new dates to LO. If DA proposes new dates LO should accept, but can also again propose new dates. This could in principle go on forever.

3. After the meeting has been arranged, it must be held (organised by LO).

[2] The original example specified that Seller sends a channel of Buyer to Shipper. We do not want to force a potential model to include channel forwarding, and have replaced it with the neutral notion of an *address*.

[3] We have omitted some of the parts, which we do not need for this presentation, specifically the steps concerning registering data in a system.

(a) No meeting can be held while LO and DA are negotiating a meeting date. Once a date has been agreed upon, a meeting should be held on the date agreed.

The second example is based on work by Lyng, Hildebrandt and Mukkamala [42, 55]

Example 2.1.4 (Chemotherapy treatment)**.** This example captures some parts of a chemotherapy treatment at a hospital, specified as part of a *clinical practice guideline*. The expected interaction is as follows: [4]

1. The doctor calculates the therapeutic dose of chemotherapy. The dose is registered on a 'flowchart' (medical term with no relation to the computer science term) and transferred to the controlling pharmacist as a prescription.

2. The controlling pharmacist checks the doctor's calculation, and registers the information on a working slip. The working slip is used by a pharmacist assistant when preparing the drug.

3. The prepared drug is checked by the controlling pharmacist, to make sure the mixture matches the patient information, and then the drug is transferred to the treatment room.

4. The responsible nurse and another authorized person (nurse or doctor) checks the prepared drug, taking both content and patient information into account.

5. The patient is administered the drug.

If any check fails, the previous actors are asked to verify a state and possibly redo a calculation. Note that it is not specified what the different 'checks' actually verify – we return to this later.

For each of these domains, standard modelling techniques, while technically correct, are all to some extent based on the assumption that every party in the scenario behaves in such a way that benefits a common goal. This common goal is usually some non-formalised or implicit properties. In the PBC case, this goal could be that the final program must work correctly, it must be efficient and it must be easy to maintain. For the business protocol example, a common goal could be that the Buyer receives the delivery details. For the trade union example, a common goal could be that a meeting is held, and for the treatment example, a common goal could be that the patient received correct treatment. We shall consider what could happen in these scenarios if this assumption of *cooperation* was inaccurate.

In the PBC scenario, the specifications formalise certain aspects of the common goal. The problem is that this common goal is not formalised. Directly, we can see that a single programmer can do a "bad-faith" implementation, e.g. implementing the square root module in such a way that the negative root is returned, or worse; by implementing a partial correctness specification through a trivial infinite loop. More subtly, there is no a priori mechanism for ensuring that the programmers use the 'best' implementation of a specification, or even a good one. Depending on how each programmer is paid (e.g. development time, lines of code), they might actually prefer a suboptimal solution.

[4] Again we leave out some parts of the original description.

Suboptimal might be in terms of running time but also in terms of code quality. If several programmers follow this trend, the problem might be further amplified, and in the end the final program might be useless; unacceptably slow or non-maintainable. While these problems could in principle be addressed by stronger specifications, yet another problem is the inclusion of third-party libraries or modules. In the traditional PBC setting, a module is absolved from blame, if a called submodule fails. However, if the programmer is free to pick his own libraries, it is possible to implement a nominally correct module, but by using flawed or buggy libraries the final program might not work as intended.

In the communication-protocol example, a deliberately malicious Seller could send the delivery details very late, perhaps waiting for other buyers to order the same item, so that the Seller might collect a discount on e.g. bulk shipping. Similarly, a malicious Buyer could continuously ask the Seller for new quotes with no intention of ever buying anything, e.g. if the Buyer was acting for a competing seller. There is also the possibility that the Shipper does not send the delivery details, either because of a failure or because of a more important (e.g. higher price) request for the same item. It is not clear, what effect such a Shipper failure has on Buyer and Seller – is it the Seller's fault for choosing a poor Shipper, or is Seller free from blame, because Seller did not fail directly?

In the trade union example, it seems plausible to assume that the employer organisation (DA) might not want to run a work related complaint case. Yet it is possible for DA to delay a meeting; by proposing infeasible meeting dates, or by purposely delaying a response. By delaying a meeting, it would be harder to investigate the matter or the employee might drop the case altogether.

In the chemotherapy treatment example, there is not a direct reason, why any party would on purpose try to invalidate the expected interaction. At the surface, however, is seems like several seemingly redundant checks are being performed. Those checks are in place, because people make mistakes, even though they are directly focused on the common goal of treating the patient. Still, everyone might have a side goal of minimising their own effort (not necessarily because they are lazy, but in order to get more work done), so some checks might not be needed, leading to a "suboptimal" solution. Another problem is that because it is not specified what the checks should ensure, there is no guarantee that every thing, which should be checked, is actually checked.

In the end, we see that the assumption of cooperation is critical for the application of the models to the real world, as only a single rogue or fallible part is enough to make the models inaccurate or inadequate, and in the worst case 'make the house of cards topple over'.

The main problem is that in a logically distributed system, where each party is a separate administrative entity, the assumption of cooperation is too strong. Firstly, there are those cases where the different parties are in direct competition, and both may be assumed to behave actively *hostilely* towards each other. The trade union mentioned above is such a scenario. That it works at all in practice must be either because DA acts irrationally, or (more plausibly) that there are additional constraints and incentives to follow the intended interaction.

Secondly, in several other cases, the parties might not be in direct competition, but they might have goals that are only partly compatible, as illustrated in Fig. 2.1. In such a setting, each party often focuses on their side and therefore does things in their own way, which can then lead to problems when compounded.

When the goals are only partly compatible, the original interaction specification is almost always a compromise, and the compromise is usually not reflected anywhere in the model. The intermediate specifications in a PBC context can be seen as such

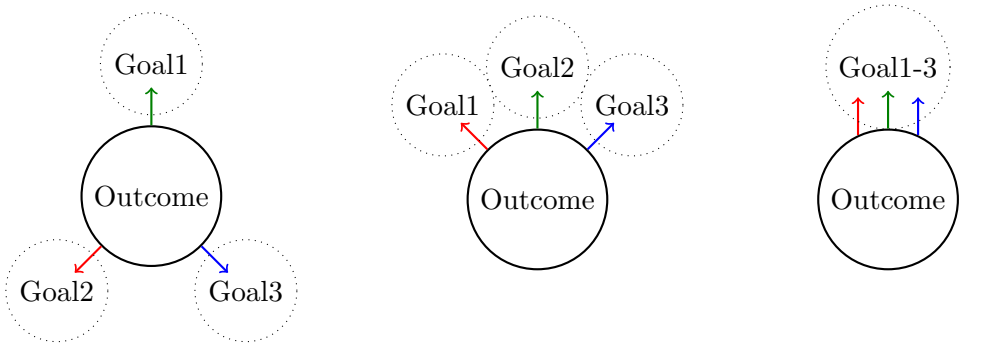


Figure 2.1: Different goal settings.

compromises; it is not clear from those specifications alone, how they where created: some programmers might have preferred an easy to implement specification, while others might have preferred harder to implement specifications (such specifications might pay more). In a bidding scenario it is usually an advantage to make the last offer, so in the business protocol example Seller would be at a disadvantage, which might have come up as a compromise, because there were more sellers available for Buyer. Even for the chemotherapy treatment there might have been a compromise, e.g. the original doctor could have checked the pharmacist's work himself, but instead a nurse can do it – probably because the doctor's time is more valuable.

In a system where these compromises are not reflected anywhere, potential changes to the specifications are hard to do; as they require a new model where a new compromise is obtained. This makes it difficult to do any non-local changes to the work processes because we cannot guarantee they will behave in accordance with the different actor's expectations.

Thirdly, there are those cases where there are no obvious entities with significantly different goals (in our opinion those cases are rare). We argue that even those cases benefit from a competitive viewpoint. The main concern is that failures can always happen, regardless of how committed to the common goal the different entities are. This means that a robust model still has to take into account that the different parties do not end up always following the rules, perhaps unintendedly. A way of viewing this is that all the cooperative parties are ultimately working together against a common adversary, say, *Nature*, illustrated as working against a current in Fig. 2.2. Such a viewpoint facilitates worst case analyses, because if Nature can win, it will eventually, as coined by the well known *Murphy's law: Anything that can go wrong will go wrong.* Considering the chemotherapy treatment example, we can view the doctor as acting rationally when determining the prescription. But when the prescription is communicated, Nature takes over and the nurse does not see, what is intended.

Instead of sweeping all these problems under the big catch all assumption of cooperation, we propose to tackle them head on, with a fundamentally *adversarial* methodology.

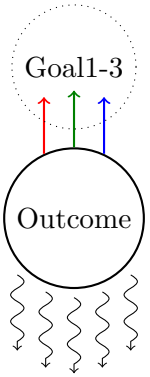


Figure 2.2: Working against a common adversary.

2.2 The adversarial methodology

In an adversarial setting, each party is an autonomous entity with its own set of goals. Those goals are not necessarily compatible with the goals of the other entities. The entities enter into *contracts* with each other in order to further their own goals. Those contracts act as specifications for the interactions of the entities, but each entity might defect from the intended meaning or even break a contract, if it is more profitable in terms of the goals. To allow any commitment, each contract may offer rewards or assign penalties to different entities. As a result, an entity will honour a contract when the reward for following it, or the penalty for breaking it, is high enough. In general, it is also not enough to reflect blame for a failure on to another entity; instead, each contracting entity must assume *responsibility* by quantifying the penalty it is willing to provide in case of failure. We note that this notion of quantified performance also makes sense in a non-physically distributed setting, because a quantified monetary guarantee is more realistically applicable in many cases than ideal mathematical correctness. Furthermore, it also allows certified components to be integrated directly with non-certified components, while still providing a (maybe different) quantified guarantee. We now consider several of the previous examples from an adversarial viewpoint.

In the PBC example, when each programmer is equipped with his own goals, we must realise that they might produce suboptimal code, if they are not rewarded/penalised accordingly; e.g. the post condition must explicitly forbid that the square root module returns the negative root. Furthermore, the risk for breaking a specification must be clear at all times, because it could be broken at any time, whether deliberately or inadvertently (e.g. not bothering to account for corner cases). Therefore, we must quantify the risk of breach for each module, and make sure that we do not let a module with a high penalty for failure depend on a module with a low penalty for failure (e.g. a third-party module). By quantifying the penalties and rewards, we move away from the traditional distinction between a module and its environment. Instead of considering the environment a single monolithic entity, we consider the environment a composite entity consisting of multiple different parts, and failure from one of those parts does not automatically void the module's commitments with the other parts.

In the communication protocol example, both Buyer, Seller and Shipper are endowed with their own goals. Buyer would now have to realise that Seller could delay the delivery details, and should therefore insist on imposing some deadline. Similarly, Seller could

insist on some penalty for Buyer, if Buyer continuously asks for new quotes. Most importantly is however that Shipper can fail independently from Seller. In many cases it will be unacceptable for Buyer to contact Shipper in case of failures, because Buyer does not know the agreements between Shipper and Seller. In those cases, Seller must assume responsibility for the delivery details, and in case of failure pay the penalty to Buyer directly. If Seller is properly organised, the agreement with Shipper ensures that the penalty owed to Buyer is sufficiently covered by penalties obtained from Shipper.

The trade union example is similar to the communication protocol example. When LO and DA have divergent goals, each party has to ensure that the other does not exploit the specification, e.g. delaying the process on purpose, by writing appropriate deadlines into the contract and assigning proper penalties.

In the chemotherapy treatment example, we consider the different actors (doctor, nurse, pharmacist) to be separate entities with their own goals. The implications are that we must analyse the actions of each actor in terms of their goals. Consider for instance the checks: in a setting with personal goals we can not directly assume that the nurse (or doctor or pharmacist) will perform the optimal check, when we do not know what optimal means. We therefore need to analyse the reasons why the checks are needed (they could be part of a legal requirement or to save resources), and make sure that the reasons are reflected in such a way that their potential absence can be analysed.

A key insight about organisational workflows (even if declaratively specified using constraints) is that they should not be used as the ultimate top-level specification, because such specifications are more accurately captured using legally binding contracts. Instead, they should be viewed as means of satisfying these contracts as well as possible, i.e., we view such workflows as implementations or intermediate-level specifications.

In addition to capturing the different scenarios more accurately, the adversarial approach also provides additional benefits. Firstly, the approach is more robust in terms of failure, because every specification can be broken at any time. We say that the focus is on the 'unhappy' and not the 'happy' path. In apparently cooperative settings, a common adversary (Nature, laws, the state, ...) focuses the attention of the cooperative participants on potential errors; e.g. if the law enforcement could come on a surprise visit at any time, a company would probably be more concerned with following the rules. Secondly, the adversarial notion of assuming responsibility for success allows a greater degree of compositionality, as an entity only needs to know its own contracts and contractors, and not any potential subcontractors. Furthermore, an entity can reason compositionally about its commitments using quantified penalties and rewards. This allows, for example, the creation of a more reliable (higher penalty for failure) service, using several redundant, but less reliable, services. Thirdly, the quantified penalties and rewards also allow optimisations and analyses of, for instance, work processes, allowing an entity to compare two implementations and pick the one which ensures the largest reward or the least penalty; yet another reason for considering workflows as implementations of specifications and not as the specifications themselves. This is especially useful in the context of dynamic changes: a change in an implementation could be analysed in terms of the existing penalties and rewards, but also a changed contract could be analysed together with the unchanged contracts. Lastly, the adversarial viewpoint includes the cooperative viewpoint as a special case, namely when each entity has the common goal as a private goal. The adversarial approach can therefore be seen as a generalisation of the cooperative approach.

As we mentioned earlier, existing models and frameworks are apparently cooperative.

In the following we discuss reasons why the adversarial methodology has not yet been considered in detail.

When trying to design, e.g., a new model for interaction, one can look at existing models and try to represent the same examples and scenarios. Because the adversarial viewpoint is not only a new model, but a new methodology, this approach does not work directly. When a situation is already modelled cooperatively, information about, e.g., different goals, is lost, as it is not captured by the model. This implies that in order to obtain the benefits of an adversarial methodology, one has to go back to the original domain experts and re-analyse the scenario; thus making such an approach more work intensive than another cooperative model. This new analysis might also uncover new problems not captured before, again providing extra work compared to another cooperative model. As an example, consider the checks in the chemotherapy treatment case. When considered cooperatively, there is no problem in including an informally or vaguely specified action called 'check', because by definition the actors perform the relevant checks. In an adversarial setting, we need to consider the reasons behind the check and make sure that the actors do the 'right thing', if not by themselves, then because of the penalties and rewards. We note that, in order to find out what these checks actually consists of, we had to go back to a domain expert[5], because was not specified in the cooperative model.

Another problem with an adversarial approach is that certain aspects, which are not formalised in a cooperative model, have to be formalised in an adversarial model. These new aspects are often challenging to model formally. An example could be how to model rewards for developing a good implementation in the PBC example, as it can be hard to express exactly what a good implementation is. Another example is how to model a medical assessment. Some penalties and rewards are also especially hard to quantify, as they might have a controversial effect. The prototypical example is the price of human life, but also actions that have an unclear future effect, e.g. actions that damage the environment. In general, there is no golden rules or silver bullets here. A good penalty and reward structure is not immediately clear; we will later present a simple structure. However, future research might come up with better structures – a good structure might also be dependent on the concrete scenario. The demand for having more properties formalised is also a challenge, because 'normal' people already have a hard time employing formal methods.

The wide span use of cooperative models also illustrates that in some sense they 'work': they prevent several types of errors and mistakes, so why is another approach needed? The situation is similar to that of the end-to-end argument in system design [88]. Briefly, the end-to-end argument states that while a reliable packet transfer medium reduces the frequency of application level failures, it cannot replace an application level error check, as errors might happen while data is read from disk or divided into packages. This not to say that the reliable medium is useless, as it still prevents lots of full file retransmissions by only resending failed packets. Transformed to the cooperation setting, we think, that the cooperative models are very useful for catching common errors, which can be dealt with quickly. But they cannot replace the robustness of an adversarial model. Lastly, in addition to the conceptual challenges of the adversarial methodology discussed above, there are additional technical challenges; e.g. real-time deadlines. We discuss these challenges later.

[5]Karen Marie Lyng, personal communication, Aug. 2011.

2.3 Modelling adversarial scenarios

In this section, we informally present the different components in our adversarial model. Hopefully, this should make the formal definitions in the next chapter easier to understand. Additionally, we give a game-based way of understanding an adversarial contract, which will form the basis for our concrete model in the next chapter.

The fundamental actors in the adversarial setting are the *principals*. Each principal represents an autonomous legal entity. Such principals could be persons, companies or organisations. Each of these principals has its own set of goals, and these goals are in general not known by the other principals.

Principals do not interact directly. Instead, each principal can control a number of *agents*. These are actors without any goals, e.g. machines. Agents can interact with each other by performing *actions*. Those actions can be based on other actions previously *observed* by the agent. Both data transmission and physical operations are expressed in terms of actions. There might be different physical restrictions between different pairs of agents; two agents might have a fast connection, while others might not even be connected.

Principals enter into *contracts* with each other, concerning interaction specified in terms of actions. Each contract is a legally binding agreement on the outcomes of sequences of interactions. Contracts are the main abstraction for e.g. formal specifications, legal agreements or clinical practice guidelines. Each contract can assign both *penalties* and *rewards* to the participating principals, e.g., monetary benefits. Contracts, being mutual agreements, are inherently zero-sum, in that a party can not receive more than the other parties pay. In general, each agent can help fulfil several contracts, but several agents might also work together to fulfil a single contract. In delegation scenarios, it is possible for principals to negotiate contracts concerning actions of agents that are not controlled by any of the contracting principals.

To do well in its contracts, a principal must devise a *strategy* for its agents to follow; each agent follows a part of the strategy called a *tactic*. The tactic specifies which actions the agents should perform, when they observe certain other actions. A strategy encompasses the entire contract portfolio of a principal, and can be viewed as an implementation of the specification induced by the contracts. Examples of such strategies and tactics include program code or instructions for performing work tasks. When delegating, a principal must make sure that its strategy takes into account that the agents performing the tasks follow the (unknown) strategy of their controller. Therefore, the principal cannot depend on a specific behaviour of those agents.

Adversarial or competitive settings have been extensively studied in the context of *game theory* [99, 57]. We believe that viewing specifications and contracts from the viewpoint of game-theory leads to a more intuitive model, and that it should be possible to draw from the experience of the game-theoretic field. Concretely, we view the principals as players, a contracting connection between principals corresponds to a game, and the particular contract to the rules of the game. The actions correspond to the moves in the games, and the penalties and rewards correspond to assigning pay-offs to different outcomes of the games. Strategies correspond directly to the game-theoretic notion.

Chapter 3

Adversarial models of interaction

In this chapter we define two formal models for contracts and strategies. One is an extensional model based on traces and one is an intensional model based on automata. The intensional model is naturally less expressive as the extensional model, however, all automaton-based contracts can be given a trace-based denotation. We prove that automaton conformance implies conformance of the trace-based denotations. The material in this chapter is an extension of earlier work in form of a paper [38] and an internal tech report [49].

This chapter is organised as follows: first we define the general communication concepts including traces; then we define trace-based specifications called contracts and trace-based implementations called tactics, including a notion of composition and conformance; then we define the automaton version of contracts and automatons, with a notion of conformance; and lastly we prove that automaton-based conformance implies trace-based conformance. We finish the chapter with related and future work.

3.1 Communication setup

The principals in the model are the autonomous legal entities.

Definition 3.1.1. *Principals are taken from an unspecified set of principals,* **Principal**. *We use p to range over principals.*

In a specific scenario the relevant principals will not be this big set, but a smaller and finite subset. The relevant principals will also be dependent on the viewpoint. When presenting the examples, we do not wish to specify the exact principals involved; e.g. we would like to refer to the Doctor without specifying a particular doctor. What we do is to present the example in terms of *roles* (e.g. Buyer, Doctor, ...) and then we assume that for a specific scenario those roles can be instantiated with proper principals.

Example 3.1.2 (Continuing 2.1.2)**.** In the business protocol example, the roles considered are Buyer, Seller, and Shipper. Each role would be instantiated to a particular principal; e.g. Buyer could be Copenhagen University, Seller could be Amazon, and Shipper could be UPS. We assume that we have picked such concrete principals and then continue the example using Buyer, Seller and Shipper as principals. Taking different viewpoints into account, Buyer should not need to know Shipper, and Shipper only needs to know the delivery address of Buyer, not its identity. Therefore, from the viewpoint of Buyer, the relevant principals are:

$$P_{\mathsf{Buyer}} = \{\mathsf{Buyer}, \mathsf{Seller}\}.$$

Similarly, the relevant principals from the viewpoint of Seller and Shipper, respectively, are:

$$P_{\mathsf{Seller}} = \{\mathsf{Buyer}, \mathsf{Seller}, \mathsf{Shipper}\} \text{ and } P_{\mathsf{Shipper}} = \{\mathsf{Seller}, \mathsf{Shipper}\}.$$

Our use of roles and principals are similar to the use of *roles* for *Distributed Dynamic Condition Response Structures* (DDCR) [41], which have been used to model the organisational workflows we considered in the last chapter.

Example 3.1.3 (Continuing 2.1.3 & 2.1.4)**.** In the trade-union example, the principals, (roles) are precisely the roles given in the formalisation by Hildebrandt, Mukkamala and Slaats [43]:

$$P = \{\mathsf{Union}, \mathsf{LO}, \mathsf{DA}\}.$$

In this case, Union is a role, because it can stand for any union, but LO and DA are specific principals.

For the chemotherapy treatment example, as we mentioned earlier, it can be advantageous to include Nature as a special principal, which can be seen as working against the rest of the principals:

$$P = \{\mathsf{Doctor}, \mathsf{Nurse}, \mathsf{Pharmacist}, \mathsf{Assistant}, \mathsf{Nature}\}.$$

As mentioned before, each principal controls a set of agents and each agent is controlled by at most one principal.

Definition 3.1.4. *An agent, a, is taken from a set of agents,* **Agent**. *Each principal, p, has an associated set of agents, A_p, and two different principals, p, p', have disjoint set of agents $A_p \cap A_{p'} = \emptyset$.*

The communication protocol example can have a simple agent structure:

Example 3.1.5 (Continuing 2.1.2)**.** Buyer, Seller, and Shipper each has a single agent, performing all communication:

$$A_{\mathsf{Buyer}} = \{\mathsf{a}_{\mathsf{Buyer}}\}, \quad A_{\mathsf{Seller}} = \{\mathsf{a}_{\mathsf{Seller}}\}, \quad A_{\mathsf{Shipper}} = \{\mathsf{a}_{\mathsf{Shipper}}\}.$$

The agent setup is static, that is, we do not specify in the model how agents are created or removed. It should, however, be possible to create a generic agent contract, in which a principal needing another agent sends the desired tactic to another principal, representing either a 'free' agent or an 'agency'. The principal then executes the tactic on a corresponding agent. There is, of course, no guarantee that the tactic would be executed faithfully by an agent for hire, but then the agent contract must specify appropriate penalties.

3.2 Communication concepts

In this section we present the main communication concepts used in the model.

Time

To verify and certify real-time scenarios, our model needs a notion of time. In particular, as previously mentioned, *deadlines* are crucial in almost all settings. Our concrete time model will be one, where the time domain is continuous, point-based and quantifiable. A continuous domain, instead of a discrete domain eliminates the need to fix a global time precision. In a discrete setting, if each time step is 1 second, we can not directly move to a setting, where each step is 1/10 second for some of the interactions. It is important to note that nothing in the model needs a truly continuous time-domain; it is only for the modelled examples we use it. We use a point-based domain instead of an interval-based one, because it is more clear what it means to be within a deadline. Quantitative time, as opposed to qualitative time, means that the actions are not only ordered by time, but that there is also a quantifiable time difference between each action. We use such a domain, because we want to be able to capture deadlines.

To make the time domain concrete, we use the real numbers ($\mathbb{R}$), but we could just as well have used the rationals. To make the presentation clear, we separate the use of time into two domains: a domain of *time points* and a domain of *time differences*. Both domains are modelled as real numbers, but we only use certain operations on each domain, e.g. we cannot add two time points, but we can add a time difference to a time point.

Definition 3.2.1. *The set of time points* **Time** *(ranged over by t) and the set of time differences* **TimeD** *(ranged over by d) are both defined to be the real numbers:*

$$\mathbf{Time} := \mathbb{R} \qquad\qquad \mathbf{TimeD} := \mathbb{R}.$$

We will make use of the following operations:

- $\leq \subseteq \mathbf{Time} \times \mathbf{Time}$*, is the comparison between two time points;*
- $\leq \subseteq \mathbf{TimeD} \times \mathbf{TimeD}$*, is the comparison between two time differences (the use of the same symbol should not pose any problems);*
- $+ : \mathbf{Time} \times \mathbf{TimeD} \to \mathbf{Time}$*, is the addition of a time difference to a time point;*
- $- : \mathbf{Time} \times \mathbf{Time} \to \mathbf{TimeD}$*, is the time difference between two time points.*

We often need to stipulate strictly positive time differences and therefore define:

$$\mathbf{TimeD}_+ := \{d \in \mathbf{TimeD} \mid d > 0\}.$$

Time points and time differences are both used when modelling the concrete scenarios. In the examples, time differences are usually expressed in seconds, so that e.g. $3s = 3, 4m = 4 \cdot 60 = 240$ and $1ms = 0.001$. Similarly, we let time points be relative to a specific time.

Example 3.2.2 (Cont. 2.1.2)**.** The extra timing requirement in the business protocol example could be captured by a time difference of 30 seconds:

$$d_{\mathsf{QuoteDeadline}} = 30s$$

So if the quote from seller arrived at 15:30:45, a reply at 15:31:00 would be on time and a reply at 15:31:30 would be too late:

$$t_{\mathsf{QuoteTime}} = 15{:}30{:}45, \quad t_{\mathsf{OnTime}} = 15{:}31{:}00, \quad t_{\mathsf{Late}} = 15{:}31{:}30.$$

Note that we have omitted the date part of the time points to make them less verbose, because all time points are assumed to be on the same date.

Actions and links

All communication in the model takes place in form of *actions*. An action is initiated by a *sender*, and then observed by a *receiver*. In our model, an action can never be blocked, meaning that it is entirely up to the sender when actions happen. A receiver can choose to ignore an action, but not prevent its occurrence.

Disallowing the blocking of actions is different from the standard version of several process calculi (e.g. CSP [46]), but still a fairly common approach, taken for instance by input/output automata [54] or any asynchronous calculi. We have two main reasons for disallowing blocking. Firstly, we only want to reason about the actually communicated actions and not about potential refusals of actions. Such refusals would for instance be hard to document in practice. Secondly, blocking reception of unexpected input is not robust in a setting where the communication partners are fallible. We want all errors or unexpected behaviour to be able to occur, so that it can be assigned a penalty.

Each communication action takes place along *links*. A link is a directed, one-to-one communication medium between two agents. A link represents an idealised medium: when an action is performed, it is always observed (but not necessarily processed) after a certain time. Each link has an associated *latency*, which is the time it takes from an action being performed and until it is observed. In a data-transmission setting, the time that the action is performed is the time when the first bit is being sent, and the time that the action is observed is the time when the last bit is received. For simplicity, we assume that each link has a capacity of one, meaning that only one action is being transmitted at a time.

Even though we have chosen a fairly simple communication setup, it is possible to encode more complicated setups using special agents and contracts. A blocking medium could be implemented by a protocol with explicit acknowledgements. In the same way, an unreliable medium could be specified with a contract that allows the agents to drop some of the messages. In such a way, we can also quantify the assurance we have on a particular medium. Similarly, one-to-many or many-to-one links could also be specified with special agents. For now, we consider the link setup to be static; in Chapter 5 we suggest a refinement of links to *resources*, which makes for a more dynamic communication setup.

Actions are parametrised by *values* (e.g. the data in a packet, the amount paid in a transfer, the medicine administered). An action can then be modelled as the link used paired with a value, transmitted from sender to receiver. We allow the values of each link to come from a different value domain, which we call the *alphabet* of that link. We consider each value domain abstract at the moment, but a concrete instance of the model would fix the domain to a specific domain (e.g. finite sequences of bits).

Definition 3.2.3. *Links, l, are taken from some unspecified set of links,* **Link**. *For each link l we assign a domain of values,* $\mathbf{Value}_l$*, corresponding to the actions along that link. Each link has a single unique sender and receiver:* $\mathrm{send}(l) \in \mathbf{Agent}, \mathrm{recv}(l) \in \mathbf{Agent}$. *There may be multiple links, usually with different alphabets, between each pair of agents. The links where an agent is the receiver/sender are called the* input/output *of that agent:*

$$\mathrm{inp}(a) = \{l \mid \mathrm{recv}(l) = a\}$$
$$\mathrm{outp}(a) = \{l \mid \mathrm{send}(l) = a\}$$

Similar to the situation for principals and roles, links are a concrete connection between two agents. When considering the contract, it is be useful to employ *contract*

templates, which specify the interaction in terms of *channels*, also called logical links. Channels are independent of, which agents are available at contracting time, but will be instantiated to concrete links before run time. We will use α in the examples to range over channels. The usage of channels gives a mildly dynamic system, in which the principals can negotiate contracts (templates) without considering the executing agents directly. We do not model the instantiation of channels with links, but, as mentioned, Chapter 5 suggests a refinement of links, giving a more dynamic system.

Example 3.2.4 (Cont. 2.1.2 & 3.1.2)**.** In the business protocol example we consider the following channels (using $\mathsf{B}, \mathsf{S}, \mathsf{H}$ for Buyer, Seller, and Shipper respectively):

$$\{\alpha_{\mathsf{B}\to\mathsf{S}}, \alpha_{\mathsf{S}\to\mathsf{B}}, \alpha_{\mathsf{S}\to\mathsf{H}}, \alpha_{\mathsf{H}\to\mathsf{S}}, \alpha_{\mathsf{H}\to\mathsf{B}}\},$$

The channel $\alpha_{\mathsf{B}\to\mathsf{S}}$ is a channel from Buyer to Seller. This channel is then instantiated with the link $l_{\mathsf{AB}\to\mathsf{AS}}$ with the following value domain:

$$\mathbf{Value}_{l_{\mathsf{AB}\to\mathsf{AS}}} = \{\mathsf{RequestQuote}, \mathsf{Accept}, \mathsf{Reject}\},$$

having Buyer's agent as sender and Seller's agent as receiver:

$$\mathrm{send}(l_{\mathsf{AB}\to\mathsf{AS}}) = \mathsf{a}_{\mathsf{Buyer}}, \quad \mathrm{recv}(l_{\mathsf{AB}\to\mathsf{AS}}) = \mathsf{a}_{\mathsf{Seller}}.$$

Similarly, $\alpha_{\mathsf{S}\to\mathsf{B}}$ can be instantiated with $l_{\mathsf{AS}\to\mathsf{AB}}$ with the following value domain:

$$\mathbf{Value}_{l_{\mathsf{AS}\to\mathsf{AB}}} = \{\mathsf{OrderConfirmation}\} \cup \{\mathsf{QuoteResponse}(n) \mid n \in \mathbb{N}\},$$

and the opposite sender and receiver. The rest of the channels follow the same pattern. The input/output for e.g. $\mathsf{a}_{\mathsf{Buyer}}$ ends up being:

$$\begin{aligned}\mathrm{inp}(\mathsf{a}_{\mathsf{Buyer}}) &= \{l_{\mathsf{AS}\to\mathsf{AB}}, l_{\mathsf{AH}\to\mathsf{AB}}\}\\ \mathrm{outp}(\mathsf{a}_{\mathsf{Buyer}}) &= \{l_{\mathsf{AB}\to\mathsf{AS}}\}\end{aligned}$$

As described above, each link has an associated time between an action is requested and it is actually performed. Formally, we associate a *latency* with each link expressing how quickly actions can be performed.

Definition 3.2.5. *The function* $\mathrm{lat} : \mathbf{Link} \to \mathbf{TimeD}_+$ *assigns a positive latency to each link.*

Such a latency assignment represents physical constraints, and is given as part of the scenario; in most cases it can be set sufficiently low, and then the contracts can specify other restrictions (e.g. maximal response time or minimal time between requests) on top of that.

Example 3.2.6 (Cont. 2.1.2)**.** In the protocol example we assign a latency of 1 second to the links $l_{\mathsf{B}\to\mathsf{S}}$ and $l_{\mathsf{S}\to\mathsf{B}}$:

$$\mathrm{lat}(l_{\mathsf{B}\to\mathsf{S}}) = \mathrm{lat}(l_{\mathsf{S}\to\mathsf{B}}) = 1s,$$

expressing that at most 1 message per second can be sent.

Note that even though two links share the same latency, it does not mean that they have to be in sync. It only expresses that between an action being performed and received, there must be at least a time difference of the latency, not that actions are only allowed on multiples of the latency.

3.3 Trace-based model

In this section, we define the trace-based model for contracts and strategies. We start by defining *traces* themselves. A trace is a way of representing a communication history. It captures which actions have happened and at what time. All traces are finite and contain actions from a fixed set of links.

Formally, a trace assigns values to the given links at different times, while respecting the latencies of the individual links. We keep the time points sorted in strict ascending order to easier access parts of the trace.

Definition 3.3.1 (Trace)**.** *A trace over a (finite) set of links, L, is a mapping*[1] *from the links into lists of time points and values:*

$$\sigma \in \mathbf{Trace}_L := \prod_{l \in L} (\mathbf{Time} \times \mathbf{Value}_l)^*,$$

satisfying that for all $l \in L$ where $\sigma(l) = [(t_1, v_1), \ldots, (t_n, v_n)]$:

$$\forall i < j.\ t_j - t_i \geq \mathrm{lat}(l).$$

We write $[\,]_{\mathrm{tr}}$ for the trace that maps all links into empty lists.

We have chosen to divide the different links into separate lists, as we think this gives the clearest definitions; an alternative formulation of a trace would be to use a single list, with partial mappings for each time point, i.e. $(\mathbf{Time} \times \prod_{l \in L}(\mathbf{Value}_l + \{\varepsilon\}))^*$. Note the way we represent partial maps: instead of writing $f : A \rightharpoonup B$, we write $f : A \to B + \{\varepsilon\}$, because we believe this is a more clear notation. However, we do not write explicit injections, so we often write $f(a) = \varepsilon$ instead of $f(a) = \mathrm{in}_2(\varepsilon)$.

In addition to the actual actions, the designated end time for a given trace is also important (e.g. with respect to deadlines). We therefore define traces where all time points are less than a given time point.

Definition 3.3.2. *Given a set of links, L, and a time point, t, we define the set of all traces up to and including t as:*

$$\mathbf{Trace}_L^t := \{\sigma \in \mathbf{Trace}_L \mid \forall l \in L.\ \forall (t', v) \in \sigma(l).\ t' \leq t\}.$$

The notation $(t, v) \in \mathrm{ts}$ refer to list membership.

We show an example trace:

Example 3.3.3 (Trace)**.** Given links l_1 and l_2 with latency $\mathrm{lat}(l_1) = 1.0$ and $\mathrm{lat}(l_2) = 0.3$, and value domains $\mathbf{Value}_{l_1} = \{\mathsf{a}, \mathsf{b}\}$ and $\mathbf{Value}_{l_2} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$, the following:

$$\sigma = \left\{ l_1 \mapsto [(3.1, \mathsf{a}), (6.5, \mathsf{b}), (8.0, \mathsf{a})], l_2 \mapsto [(4.5, \mathsf{z}), (4.8, \mathsf{z}), (5.5, \mathsf{x})] \right\}$$

is a legal trace $\sigma \in \mathbf{Trace}_{\{l_1, l_2\}}$. If the time point for the last value for l_2 had been 5.0, the recovery time for l_2 would have been violated, and therefore σ would not have been a legal trace. Every time point greater than or equal to 8.0 can be used as end time, e.g. $\sigma \in \mathbf{Trace}^{8.0}_{\{l_1, l_2\}}$ and $\sigma \in \mathbf{Trace}^{20.0}_{\{l_1, l_2\}}$.

[1] We use $\prod$ to refer to the dependent function space. That is, when $f \in \prod_{x \in A} B_x$, then $f : A \to \cup_{x \in A} B_x$, satisfying $f(a) \in B_a$.

Traces are used to define contracts and strategies; we introduce several operators to simplify the presentation:

Definition 3.3.4. *The following operations are associated with traces:*

- *For* $ts \in (\mathbf{Time} \times \mathbf{Value}_l)^*$, *we have* $ts_{\leq t} = [(t', v) \mid (t', v) \in ts \wedge t' \leq t]$, *and similarly for* $ts_{< t}$. [2]
- *The time restriction extends to traces:* $\sigma_{\leq t}(l) = \sigma(l)_{\leq t}$, *and* $\sigma_{< t}(l) = \sigma(l)_{< t}$.
- *If* $L \subseteq L'$ *and* $\sigma \in \mathbf{Trace}_{L'}$ *then* $\sigma_{|L} \in \mathbf{Trace}_L$ *is defined by restriction of the domain.*
- *The first/last time point for a given trace* $\sigma \in \mathbf{Trace}_L$ *is defined in the following way:*

$$\begin{aligned} \mathrm{first}(\sigma) &= \min\{t \mid \exists l \in L.\ (t, v) \in \sigma(l)\} \\ \mathrm{last}(\sigma) &= \max\{t \mid \exists l \in L.\ (t, v) \in \sigma(l)\} \end{aligned}$$

- *When* $L_1 \cap L_2 = \emptyset$ *then* $\cup : \mathbf{Trace}_{L_1} \times \mathbf{Trace}_{L_2} \to \mathbf{Trace}_{L_1 \cup L_2}$, *is the union, defined in the following way:*

$$(\sigma \cup \sigma')(l) = \begin{cases} \sigma(l) & \text{if } l \in L_1, \\ \sigma(l) & \text{if } l \in L_2. \end{cases}$$

Again we show examples of the definitions:

Example 3.3.5. Given the trace σ from Ex. 3.3.3, we have that:

$$\begin{aligned} \sigma_{\leq 5.0} &= \big\{l_1 \mapsto [(3.1, \mathsf{a})], l_2 \mapsto [(4.5, \mathsf{z}), (4.8, \mathsf{z})]\big\} \\ \sigma_{|l_1} &= \big\{l_1 \mapsto [(3.1, \mathsf{a}), (6.5, \mathsf{b}), (8.0, \mathsf{a})]\big\} \\ \mathrm{first}(\sigma) &= 3.1 \\ \mathrm{last}(\sigma) &= 8.0 \end{aligned}$$

Now given:

$$\sigma' = \big\{l_3 \mapsto [(3.5, \mathsf{o}), (12.5, \mathsf{o})]\big\}$$

for some l_3 with $\mathbf{Value}_{l_3} = \{\mathsf{o}\}$, we have that:

$$\begin{aligned} \sigma \cup \sigma' = \big\{&l_1 \mapsto [(3.1, \mathsf{a}), (6.5, \mathsf{b}), (8.0, \mathsf{a})], \\ &l_2 \mapsto [(4.5, \mathsf{z}), (4.8, \mathsf{z}), (5.5, \mathsf{x})], \\ &l_3 \mapsto [(3.5, \mathsf{o}), (12.5, \mathsf{o})]\big\} \end{aligned}$$

Each of the operations in Def. 3.3.4 preserves the designated end time of the traces.

Proposition 3.3.6. *The following holds:*

- *If* $\sigma \in \mathbf{Trace}_L^t$ *then* $\sigma_{\leq t'} \in \mathbf{Trace}_L^t$, *and similarly for* $\sigma_{< t'}$.

[2] The notation $[f(x) \mid x \in \mathrm{xs} \wedge P(x)]$ is a list comprehension known from functional programming languages, it is defined the same way as for sets, but it respects the order of the list.

- *If* $\sigma \in \mathbf{Trace}_L^t$ *then* $\sigma_{|L'} \in \mathbf{Trace}_{L'}^t$.
- *If* $\sigma \in \mathbf{Trace}_{L_1}^t$ *and* $\sigma' \in \mathbf{Trace}_{L_2}^t$ *then* $\sigma \cup \sigma' \in \mathbf{Trace}_{L_1 \cup L_2}^t$.

Proof. Follows directly from the definitions. □

Having defined traces and their operations, we move to the definition of contracts.

3.3.1 Trace-based contracts

A contract assigns rewards and penalties to certain interaction sequences. Interactions are formalised as traces, and therefore each contract denotes a function on traces that for a given trace specifies a *verdict*. In the context of runtime monitoring, such a function is also called a *monitor* [51]. We require several properties:

1. Contracts must be deterministic: from a given interaction trace there should only be a single verdict.
2. The penalty and reward assignment should be final: when an assignment has been specified, no later actions should be able to change it.
3. The assignment of rewards must be zero-sum: a principal can not receive more as a reward than the others paid as penalties. In this way we can consider the penalties and rewards a linear resource, similar to actual money.

As domain for pay-offs, we have chosen the real numbers, because we do not wish to fix a specific minimal currency (e.g. to account for micro-transactions). To capture the last property we define verdicts:

Definition 3.3.7. *A n-ary verdict is either not-finished or a pay-off vector:*

$$v \in \mathbf{Verdict}_n := \{\bot\} + \mathbb{R}^n,$$

satisfying for $v = (k_1, \ldots, k_n)$*:*

$$\sum_{i=1}^{n} k_i = 0.$$

As mentioned, contracts should be deterministic, which is modelled by viewing contracts as a function that assigns verdicts to traces. The traces we consider include an end point, which is expected to lie after the actions in the trace. This end point is used to model that even though no actions are happening, the contract can still evolve into a final state (e.g. in the case a deadline passes).

Definition 3.3.8. *A (trace-based) contract between n parties regarding a finite set of links, L, is a function:*

$$c \in \mathbf{Contract}_L := \prod_{t \in \mathbf{Time}} (\mathbf{Trace}_L^t \to \mathbf{Verdict}_n),$$

satisfying a monotonicity condition:

$$\forall \sigma \in \mathbf{Trace}_L^{t'}.\ t \leq t' \Rightarrow \big(c(t)(\sigma_{\leq t}) = \bot \vee c(t)(\sigma_{\leq t}) = c(t')(\sigma)\big)$$

The monotonicity condition ensures exactly the second requirement; assignment of pay-offs is final. While the contract function is a nice semantic notion, it is harder to write concrete contracts. Nevertheless, we present a simple example, to illustrate the definition:

Example 3.3.9. We consider the following contract between two parties:

> Whenever a number, n, is sent on link l_1, then $n+2$ must be sent on link l_2 within $10s$. If not, the first party receives a pay-off of 1. Note that neither l_1 or l_2 need to involve agents directly controlled by the contracting parties.

The contract is defined below:

$$c(t,\sigma) = \begin{cases} (1,-1) & \text{if } \exists (t',n) \in \sigma(l_1).\ t' + 10\text{s} \leq t \\ & \quad \wedge\ \neg(\exists (t'',m) \in \sigma(l_2).\ t' < t'' < t' + 10\text{s} \wedge m = n+2); \\ \bot & \text{otherwise.} \end{cases}$$

This contract specifies the several requests can be submitted before answers are received, however, in such a case the order of the requests are not respected by the responses. Later we see a contract where multiple requests are explicitly forbidden.

Bilateral contracts ($n = 2$) are easier to compose, and should be sufficient for most examples. In the rest of this work we therefore consider such contracts only. The set of verdicts can therefore be simplified to either non-finished or a single real number:

Definition 3.3.10. *The set of bilateral verdicts (from the viewpoint of the first party) is:*

$$\mathbf{Verdict} := \{\bot\} + \mathbb{R}.$$

Ex. 3.3.9 can easily be modified to a bilateral setting, by changing the pay-off vector $(1,-1)$ into the number 1.

3.3.2 Trace-based tactics

A principal's strategy, also called its *implementation*, is the collection of I/O behaviours each agent is expected to exhibit. We refer to each specification of behaviour as the *tactic* for that agent. In a programming setting, the tactics correspond to the program code.

Each agent observes the actions on its input links, and then performs actions on its output links. We require the input links to be disjoint from the output links. This model for agents means that the only way agents can react to something, is if they observe it on one of their input links. This implies that the only way principals can affect the execution of the agents is either through the tactic or through an explicit link from another of the principal's agents. In general, it is not possible to change the running code, unless it is explicitly modelled as input. As mentioned earlier, agents do not have any goals themselves or any knowledge of the contracts of their principal.

As we described earlier, each link has an associated latency. The model has to take the latencies into account, so that an action in a trace cannot be influenced by the observations made during the latency time. In the model, this corresponds to requiring that an action at time t on link l can only depend on observations made before $t - \text{lat}(l)$.

The formal semantic model for tactics is similar to the semantic model for contracts. We formalise the tactic as a function mapping traces over input links to traces over output links.

Definition 3.3.11. *A (trace-based) tactic with input L_{in} and output L_{out}, where $L_{\text{in}} \cap L_{\text{out}} = \emptyset$, is a function:*

$$\tau \in \mathbf{Tactic}_{L_{\text{in}} \to L_{\text{out}}} := \prod_{t \in \mathbf{Time}} (\mathbf{Trace}^t_{L_{\text{in}}} \to \mathbf{Trace}^t_{L_{\text{out}}}),$$

satisfying a monotonicity condition:

$$\forall \sigma, \sigma' \in \mathbf{Trace}^t_{L_{\text{in}}}.\ \left(t' \leq t \wedge \sigma_{\leq t'} = \sigma'_{\leq t'}\right) \Rightarrow \\ \forall l \in L_{\text{out}}.\ \tau(t)(\sigma)(l)_{\leq t' + \text{lat}(l)} = \tau(t)(\sigma')(l)_{\leq t' + \text{lat}(l)}.$$

The monotonicity condition is similar to the one for contracts. When the input traces match up until a certain time point, we require the output traces for each link to match up until that time point plus the latency of the link, because it would take at least a time period equal to the latency to react. Similar to contracts, we do not plan to write that many concrete tactics, but we show a simple one as an example:

Example 3.3.12. The tactic, τ, which repeatedly reads a number n on link l_1 and then outputs $n + 2$ one time unit later on link l_2 is defined in the following way:

$$\tau(t)(\{l_1 \mapsto [(t_1, n_1), \ldots, (t_k, n_k)]\}) = \{l_2 \mapsto [(t_1 + 1.0, n_1 + 2), \ldots, (t_k + 1.0, n_k + 2)]\},$$

where $t_k + 1.0 < t$. If the latency of l_1 and l_2 are equal and less than 1.0, then this tactic satisfies the monotonicity condition; otherwise the function must be changed, such that the generated trace respects the latencies.

In addition to the intuition that reaction takes time, the monotonicity condition also allows a definition of what it means for two tactics to operate in parallel. This corresponds to what happens, when the principal runs all his agents together, which in turn gives meaning to the principal's strategy. We call this type of parallel composition an *external* parallel composition, because it can compose any tactics, regardless of what language they are written in (not even the same) – in fact the languages themselves might even include their own notion of *internal* parallel composition.

A parallel composition of two tactics is shown in Fig. 3.1. As illustrated in the figure, when combining two tactics, output from one tactic may be used for input to the other tactic (links in $L_{\text{in1}} \cap L_{\text{out2}}$), and vice versa (links in $L_{\text{in2}} \cap L_{\text{out1}}$). Formally this internal communication can be captured by a fixed-point. We must prove that the desired fixed point exists and is unique, the result depends on the monotonicity of the tactics in two ways: for uniqueness, we need that output from a tactic at a time is determined, when the input up to that time point is known. For existence we need the latencies, so that each tactic cannot 'speed up' the other one. The properties of the fixed point is captured by the following proposition:

Proposition 3.3.13. *Given two tactics, τ_1 and τ_2 with input/output links $L_{\text{in1}}/L_{\text{out1}}$ and $L_{\text{in2}}/L_{\text{out2}}$, where $L_{\text{in1}} \cap L_{\text{in2}} = L_{\text{out1}} \cap L_{\text{out2}} = \emptyset$. Let*

$$L_{\text{in12}} = L_{\text{in1}} \cup L_{\text{in2}},$$
$$L_{\text{out12}} = L_{\text{out1}} \cup L_{\text{out2}}.$$

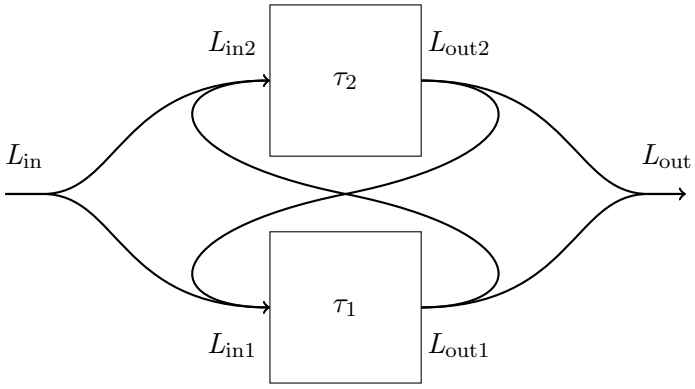


Figure 3.1: External parallel composition of two tactics.

Now for all $t \in \mathbf{Time}$ *and* $\sigma \in \mathbf{Trace}^t_{L_{\text{in12}} \setminus L_{\text{out12}}}$ *there exists a unique* $\sigma' \in \mathbf{Trace}^t_{L_{\text{out12}}}$ *such that the following two equations hold:*

$$\tau_1(t)((\sigma \cup \sigma')_{|L_{\text{in1}}}) = \sigma'_{|L_{\text{out1}}},$$
$$\tau_2(t)((\sigma \cup \sigma')_{|L_{\text{in2}}}) = \sigma'_{|L_{\text{out2}}}.$$

The function

$$\Phi : \prod_{t \in \mathbf{Time}} (\mathbf{Trace}^t_{L_{\text{in12}} \setminus L_{\text{out12}}} \to \mathbf{Trace}^t_{L_{\text{out12}}})$$

that for each time and trace assignes the uniquely defined σ' *is monotone when interpreted as a tactic.*

Proof. In the following let $m = \min\{\text{lat}(l) \mid l \in L_{\text{out12}}\}$. We start by proving uniqueness, so assume that there exists σ'_1 and σ'_2, such that:

$$\tau_1(t)((\sigma \cup \sigma'_1)_{|L_{\text{in1}}}) = (\sigma'_1)_{|L_{\text{out1}}},$$
$$\tau_2(t)((\sigma \cup \sigma'_1)_{|L_{\text{in2}}}) = (\sigma'_1)_{|L_{\text{out2}}},$$
$$\tau_1(t)((\sigma \cup \sigma'_2)_{|L_{\text{in1}}}) = (\sigma'_2)_{|L_{\text{out1}}},$$
$$\tau_2(t)((\sigma \cup \sigma'_2)_{|L_{\text{in2}}}) = (\sigma'_2)_{|L_{\text{out2}}}.$$

We wish to prove that $\sigma'_1 = \sigma'_2$, so assume for contradiction $\sigma'_1 \neq \sigma'_2$. This means that there must exist t' such that:

$$(\sigma'_1)_{<t'} = (\sigma'_2)_{<t'} \wedge (\sigma'_1)_{\leq t'} \neq (\sigma'_2)_{\leq t'}.$$

Now because m is positive we have that:

$$(\sigma'_1)_{\leq t'-m} = (\sigma'_2)_{\leq t'-m},$$

and from monotonicity of τ_1 and τ_2 we get:

$$\tau_1(t)((\sigma \cup \sigma'_1)_{|L_{\text{in1}}})_{\leq t'} = \tau_1(t)((\sigma \cup \sigma'_2)_{|L_{\text{in1}}})_{\leq t'},$$
$$\tau_2(t)((\sigma \cup \sigma'_1)_{|L_{\text{in2}}})_{\leq t'} = \tau_2(t)((\sigma \cup \sigma'_2)_{|L_{\text{in2}}})_{\leq t'}.$$

These gives that $(\sigma'_1)_{\leq t'} = (\sigma'_2)_{\leq t'}$, which is a contradiction.

We now consider existence. Now given t and σ we wish to construct σ'. We start by defining the function $F : \mathbf{Trace}^t_{L_{\text{out}12}} \to \mathbf{Trace}^t_{L_{\text{out}12}}$ as follows:

$$F(\sigma') = \tau_1(t)((\sigma \cup \sigma')_{|L_{\text{in}1}}) \cup \tau_2(t)((\sigma \cup \sigma')_{|L_{\text{in}2}}).$$

Now it is easy to see that for any t' and σ':

$$\sigma'_{\leq t'} = F(\sigma')_{\leq t'} \Rightarrow F(\sigma')_{\leq t'+m} = F^2(\sigma')_{\leq t'+m}.$$

Now we wish to construct a fixed-point for F: if $F([\,]_{\text{tr}}) = [\,]_{\text{tr}}$ then we are done, otherwise let $t'' = \text{first}(F([\,]_{\text{tr}}))$. Now we have that:

$$([\,]_{\text{tr}})_{< t''} = F([\,]_{\text{tr}})_{< t''}.$$

Which again means that:

$$([\,]_{\text{tr}})_{\leq t''-m} = F([\,]_{\text{tr}})_{\leq t''-m}.$$

A simple proof by induction now shows that:

$$F^{n+1}([\,]_{\text{tr}})_{\leq t''+m\cdot n} = F^{n+2}([\,]_{\text{tr}})_{\leq t''+m\cdot n}.$$

Now with:

$$N = \left\lceil \frac{t - t''}{m} \right\rceil,$$

we have that $F^{N+1}([\,]_{\text{tr}})$ is a fixed point of F, because $t'' + m \cdot N \geq t$. Now with $\sigma' = F^{N+1}([\,]_{\text{tr}})$ the two equations follow.

That Φ is monotone follows directly from monotonicity of the two tactics. □

We can use this proposition to define parallel composition:

Definition 3.3.14 (External parallel composition)**.** *Given two tactics, τ_1 and τ_2 with input/output links $L_{\text{in}1}/L_{\text{out}1}$ and $L_{\text{in}2}/L_{\text{out}2}$, where $L_{\text{in}1} \cap L_{\text{in}2} = L_{\text{out}1} \cap L_{\text{out}2} = \emptyset$. Let*

$$L_{\text{in}12} = L_{\text{in}1} \cup L_{\text{in}2},$$
$$L_{\text{out}12} = L_{\text{out}1} \cup L_{\text{out}2}.$$

The parallel composition

$$\tau_1 \parallel \tau_2 : \prod_{t \in \mathbf{Time}} (\mathbf{Trace}^t_{L_{\text{in}12} \setminus L_{\text{out}12}} \to \mathbf{Trace}^t_{L_{\text{out}12} \setminus L_{\text{in}12}})$$

is now defined by:

$$(\tau_1 \parallel \tau_2)(t)(\sigma) = \Phi(t)(\sigma)_{|(L_{\text{out}12} \setminus L_{\text{in}12})}$$

where Φ is defined as in Prop 3.3.13.

3.3.3 Strategies and conformance

The strategy of a principal is an assignment of tactics to agents. Each agent is assigned exactly one tactic, and the effect of the strategy is the parallel composition of the tactics, which can be viewed as a single tactic denotation. We formalise a strategy as a map from the principal's agents to tactics.

Definition 3.3.15. *A strategy for a principal, p, with agents A_p is a mapping:*

$$\Sigma : \prod_{a \in A_p} \mathbf{Tactic}_{\mathrm{inp}(a) \to \mathrm{outp}(a)}.$$

satisfying that when $\Sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$ then

$$\tau = \tau_1 \parallel \ldots \parallel \tau_n,$$

is well-defined, i.e., that the links set of the agents satisfy the disjointness condition of parallel composition. When $\tau \in \mathbf{Tactic}_{L_{\mathrm{in}} \to L_{\mathrm{out}}}$ we write $\Sigma \in \mathbf{Strategy}_{L_{\mathrm{in}} \to L_{\mathrm{out}}}$.

In general a strategy seeks to comply with several contracts at once, called a contract *portfolio*. Using the real-valued pay-offs, we simply add the pay-offs to get the pay-off for the contract portfolio. With the notion of pay-off for a portfolio, we can define what it means to implement a portfolio in a sound way. The notion of soundness we consider is that a well-behaved implementation should always receive enough rewards to cover any incurred penalties. E.g. consider a task that is delegated to a subcontractor. A proper implementation must ensure that if the subcontractor fails, the penalty received from the subcontractor must be enough to cover any penalties incurred by the main contractor. As a special case sound strategies should never 'fail first', i.e. if the strategy violates a contract, one of its subcontractors must already have violated its contract. In terms of pay-offs, this notion corresponds to ensuring that the total accumulated pay-off is non-negative at all times. If a strategy ensures this property, we say that it *conforms* with the portfolio:

Definition 3.3.16. *A strategy, $\Sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\} \in \mathbf{Strategy}_{L_{\mathrm{in}} \to L_{\mathrm{out}}}$ conforms to a portfolio $c_1, \ldots, c_m$, written $\models \Sigma : c_1, \ldots, c_m$ iff*

$$\forall t \in \mathbf{Time}, \sigma \in \mathbf{Trace}^t_{(L_{\mathrm{in}} \cup L_1 \cup \ldots \cup L_n) \setminus L_{\mathrm{out}}}. \ \sum_{i=1}^{n} \partial(c_i(t)((\sigma \cup \tau(t)(\sigma_{|L_{\mathrm{in}}}))_{|L_i})) \geq 0,$$

where c_i is a contract with links L_i, and

$$\begin{aligned} &\tau = \tau_1 \parallel \ldots \parallel \tau_n, \\ &\partial : \mathbf{Verdict} \to \mathbb{R}, \\ &\partial(\bot) = 0, \quad \partial(k) = k. \end{aligned}$$

Our definition of conformance expresses that the accumulated pay-off must be non-negative at all times, but it is possible to encode other pay-off scenarios using additional pseudo-contracts. E.g. a contract could start out by giving us a pay-off of 100, meaning the strategy could pay for some subcontractors in anticipation of eventual reward from the main client. Another contract could continuously give a pay-off of –1 every month, requiring that some minimal income is generated.

Contracts are always specified from the viewpoint of the first part. Sometimes we want to consider the same contract but from the viewpoint of the second part. With bilateral contracts this is easy: by inverting the pay-offs in a contract, the obligations and permissions are effectively reversed. This is captured by the notion of a *dual* contract:

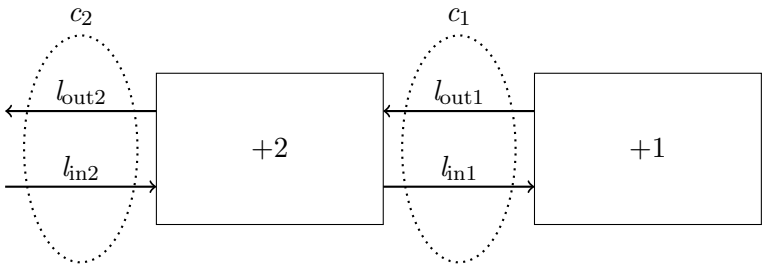


Figure 3.2: Agents in the +2 example.

Definition 3.3.17. *Given a contract c we define the dual contract $\overline{c}$ to be:*

$$\overline{c}(t)(\sigma) = \begin{cases} \bot & \text{if } c(t)(\sigma) = \bot, \\ -c(t)(\sigma) & \text{otherwise.} \end{cases}$$

To exemplify strategies and conformance, we consider a programming example.

Example 3.3.18. We consider the implementation of a +2 service. Instead of writing a +2 tactic directly, we invoke a +1 tactic twice to illustrate subcontracting. Note that in this case the computation is trivial, but this example generalises to an arbitrary function f, and the calculation of $f \circ f$, or a more complex subcontracting setting. The relevant links are:

- $l_{\text{in2}}, l_{\text{out2}}$ the input and output links from the +2 service.
- $l_{\text{in1}}, l_{\text{out1}}$ the input and output links from the +1 service.

The setup is illustrated in Fig. 3.2. Each contract has the corresponding service provider as the first party; thus the +2 tactic is the first part in c_2 and the second part in c_1 (first part in $\overline{c_1}$). For simplicity, each link has a latency of 1s, and the value domain for each link is the integers.

The +2 service contract, c_2, expresses that when a number n is received on l_{in2}, then $n + 2$ should be sent on l_{out2} within 10s, otherwise the +2 service provider receives a pay-off of –1. Similarly, the +1 service contract, c_1, expresses that when a number n is received on l_{in1}, then $n + 1$ should be sent on l_{out1} within 3s, otherwise the +1 service provider receives a pay-off of –1.

The +2 service must implement a tactic, τ_2 to run on agent a_2. The tactic must be 'conditionally' correct, which means that, if someone (a +1 service provider) is willing to fulfil c_1, then τ_2 can fulfil c_2. The conditional correctness can expressed by conformance with respect to the original contract and the dualised contract:

$$\models \{a_2 \mapsto \tau_2\} : c_2, \overline{c_1}.$$

Note that this also covers cases where the +1 service provider fails. In Chapter 4 we show how to prove this conformance.

For verification and certification purposes it is highly desirable that the model allows some level of compositionality, i.e., we wish to be able to create a full conforming strategy by composing substrategies. The general setup is as follows: We partition the agents into

several sets, and we then prove that each partition conforms with some of the contracts. We then get directly (a theorem proved in the following) that by combining all the partitions, we get a conforming strategy for all the contracts. This approach works even when the agents are allowed to interact across the partitions. But in order to treat each partition in isolation, we abstract their interaction as a contract denotation. In these contracts the principal plays both parts, and therefore it will not have any effect on the outside contracts, as they are only used to capture the intended interaction. But from the viewpoint of one trying to verify or certify a certain set of tactics, they are no different from the rest of the contracts. We revisit the previous example:

Example 3.3.19 (Continuing Ex. 3.3.18)**.** Instead of an external +1 service provider, the +2 service provider can implement the +1 tactic itself. Concretely, the provider must implement a tactic τ_1 for agent a_1, which conforms with c_1:

$$\models \{a_1 \mapsto \tau_1\} : c_1.$$

Now the combined strategy $\Sigma = \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2\}$ should conforms with the external contract c_2. This is however not trivial, but will be proved in following.

The main compositionality theorem expresses that: if we have a strategy that conforms with a set of contracts, and another strategy that conforms with some other contracts, including the dual of the original contracts. Then when combining the tactics from both strategies, they conform against those contracts that did not have a dual, expressing a form of *external compositionality.*

Theorem 3.3.20 (External compositionality)**.** *Given strategies for non-overlapping agents:*

$$\Sigma_1 \in \mathbf{Strategy}_{L_{\text{in1}} \to L_{\text{out1}}}, \quad \Sigma_2 \in \mathbf{Strategy}_{L_{\text{in2}} \to L_{\text{out2}}}$$

and contracts $c_1, \ldots, c_n$, $c'_1, \ldots, c'_{n_1}$, $c''_1, \ldots, c''_{n_2}$ *where:*

$$\models \Sigma_1 : c_1, \ldots, c_n, c'_1, \ldots, c'_{n_1},$$
$$\models \Sigma_2 : \overline{c_1}, \ldots, \overline{c_n}, c''_1, \ldots, c''_{n_2}.$$

The set of internal links are $L_{\text{int}} = (L_{\text{in1}} \cup L_{\text{in2}}) \cap (L_{\text{out1}} \cup L_{\text{out2}})$*. If the internal links are disjoint from all links of the contracts* $c'_1, \ldots, c'_{n_1}, c''_1, \ldots, c''_{n_2}$ *(*$L_{\text{int}} \cap L'_i = \emptyset$ *and* $L_{\text{int}} \cap L''_i = \emptyset$*) then*

$$\models \Sigma_1 \cup \Sigma_2 : c'_1, \ldots, c'_{n_1}, c''_1, \ldots, c''_{n_2}$$

Proof. To avoid tedious notational clutter, we only show the proof of the special case, where both strategies consist of a single tactic, ($\Sigma_1 = \{a_1 \mapsto \tau_1\}$ and $\Sigma_2 = \{a_2 \mapsto \tau_2\}$) and $n_1 = n_2 = n = 1$. So assume that we have tactics:

$$\tau_1 \in \mathbf{Tactic}_{L_{\text{in1}} \to L_{\text{out1}}}, \tau_2 \in \mathbf{Tactic}_{L_{\text{in2}} \to L_{\text{out2}}},$$

and contracts:

$$c \in \mathbf{Contract}_L, c' \in \mathbf{Contract}_{L'}, c' \in \mathbf{Contract}_{L''}.$$

Furthermore assume that:

$$\models \{a_1 \mapsto \tau_1\} : c, c', \tag{3.1}$$
$$\models \{a_2 \mapsto \tau_2\} : \overline{c}, c''. \tag{3.2}$$

We want to show that:

$$\models \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2\} : c', c''.$$

The disjointness assumption is: $L' \cap L_{\text{int}} = \emptyset$ and $L'' \cap L_{\text{int}} = \emptyset$. To prove conformance we assume $t \in \mathbf{Time}$ and $\sigma \in \mathbf{Trace}^t_{(L' \cup L'' \cup L_{\text{in}}) \setminus L_{\text{out}}}$. We now need to show that (with $\tau = \tau_1 \parallel \tau_2$):

$$\partial(c'(t)((\sigma \cup \tau(t)(\sigma_{|L_{\text{in}}}))_{|L'})) + \partial(c''(t)((\sigma \cup \tau(t)(\sigma_{|L_{\text{in}}}))_{|L''})) \geq 0.$$

Now let $\sigma' = \Phi(t)(\sigma_{|L_{\text{in}}})$, where Φ is defined as in Prop 3.3.13. By definition we therefore have $\sigma'_{|L_{\text{out}}} = \tau(t)(\sigma_{|L_{\text{in}}})$.

The general idea of the proof is to use both assumptions, and therefore define traces σ_1, σ_2 in such a way that the output of the tactic called on $\sigma_1 \cup \sigma_2$ is the same as on σ. We start by considering σ_1; we want to define it to be equal to σ (resp. σ') whenever possible. There are three different cases to consider:

1. If a link is output from τ_2 we define σ_1 to be equal to σ'.
2. If a link is not an output link but instead in either L', L'' or L_{in} then we define σ_1 to be equal to σ.
3. If neither of the above, we set the value of that link to an arbitrary value.

These considerations result in the following definition of $\sigma_1 \in \mathbf{Trace}^t_{(L \cup L' \cup L_{\text{in1}}) \setminus L_{\text{out1}}}$:

$$\sigma_1(l) = \begin{cases} \sigma'(l) & \text{if } l \in L_{\text{out2}}, \\ \sigma(l) & \text{if } l \in L' \cup L'' \cup L_{\text{in}}, \\ ts_l & \text{otherwise,} \end{cases}$$

where ts_l is an arbitrary value (e.g. $[\,]$). In a similar manner, we can define $\sigma_2 \in \mathbf{Trace}^t_{(L \cup L'' \cup L_{\text{in2}}) \setminus L_{\text{out2}}}$ in the following way:

$$\sigma_2(l) = \begin{cases} \sigma'(l) & \text{if } l \in L_{\text{out1}}, \\ \sigma(l) & \text{if } l \in L' \cup L'' \cup L_{\text{in}}, \\ ts_l & \text{otherwise.} \end{cases}$$

Now we wish to prove that:

$$\tau_1(t)((\sigma_1)_{|L_{\text{in1}}}) = \tau_1(t)((\sigma \cup \sigma')_{|L_{\text{in1}}}),$$

which comes down to showing:

$$(\sigma_1)_{|L_{\text{in1}}} = (\sigma \cup \sigma')_{|L_{\text{in1}}}.$$

So let $l \in L_{\text{in1}}$, there are two cases:

- $l \in L_{\text{out}2}$: Here we have
$$\sigma_1(l) = \sigma'(l) = (\sigma \cup \sigma')(l).$$
- $l \notin L_{\text{out}2}$: Here $l \notin L_{\text{int}}$, so $l \in L_{\text{in}}$ and therefore:
$$\sigma_1(l) = \sigma(l) = (\sigma \cup \sigma')(l).$$

Similarly we can show that:

$$\tau_2(t)((\sigma_2)_{|L_{\text{in}1}}) = \tau_2(t)((\sigma \cup \sigma')_{|L_{\text{in}1}}).$$

Now we have by the equations in Prop 3.3.13:

$$\begin{aligned}\sigma' &= \tau_1(t)((\sigma \cup \sigma')_{|L_{\text{in}1}}) \cup \tau_2(t)((\sigma \cup \sigma')_{|L_{\text{in}2}}) \\ &= \tau_1(t)((\sigma_1)_{|L_{\text{in}1}}) \cup \tau_2(t)((\sigma_2)_{|L_{\text{in}2}}) = \sigma'_1 \cup \sigma'_2,\end{aligned}$$

using σ'_1, σ'_2 as shorthands.

Now we wish to prove that:

$$c'(t)((\sigma \cup \sigma'_{|L_{\text{out}}})_{|L'}) = c'(t)((\sigma_1 \cup \sigma'_1)_{|L'}),$$

which comes down to showing that:

$$(\sigma \cup \sigma'_{|L_{\text{out}}})_{|L'} = (\sigma_1 \cup \sigma'_1)_{|L'}.$$

So let $l \in L'$ be given. From the disjointness assumption we get that $l \notin L_{\text{int}}$. We consider different cases:

- $l \notin L_{\text{out}}$: Here, $l \notin L_{\text{out}1}$ and $l \notin L_{\text{out}2}$ (because $l \notin L_{\text{int}}$). We get:
$$(\sigma \cup \sigma'_{|L_{\text{out}}})(l) = \sigma(l) = \sigma_1(l) = (\sigma_1 \cup \sigma'_1)(l).$$
- $l \in L_{\text{out}}$: Here there are two subcases:
  - $l \in L_{\text{out}1}$: Here:
  $$(\sigma \cup \sigma'_{|L_{\text{out}}})(l) = \sigma'(l) = \sigma'_1(l) = (\sigma_1 \cup \sigma'_1)(l).$$
  - $l \in L_{\text{out}2}$: Here:
  $$(\sigma \cup \sigma'_{|L_{\text{out}}})(l) = \sigma'(l) = \sigma_1(l) = (\sigma_1 \cup \sigma'_1)(l).$$

We write $v_1 = c'(t)((\sigma_1 \cup \sigma'_1)_{|L'})$. Similarly we can prove that:

$$c''(t)((\sigma \cup \sigma'_{|L_{\text{out}}})_{|L''}) = c''(t)((\sigma_2 \cup \sigma'_2)_{|L''}) = v_2.$$

As a last equality, we want to show that:

$$c(t)((\sigma_1 \cup \sigma'_1)_{|L}) = c(t)((\sigma_2 \cup \sigma'_2)_{|L}),$$

which amounts to showing that:

$$(\sigma_1 \cup \sigma'_1)_{|L} = (\sigma_2 \cup \sigma'_2)_{|L}.$$

So let $l \in L$. We consider cases:

- $l \in L_{\text{out}1}$: Here we have:

$$(\sigma_1 \cup \sigma_1')(l) = \sigma_1'(l) = \sigma'(l) = \sigma_2(l) = (\sigma_2 \cup \sigma_2')(l).$$

- $l \in L_{\text{out}2}$: Here we have:

$$(\sigma_1 \cup \sigma_1')(l) = \sigma_1(l) = \sigma'(l) = \sigma_2'(l) = (\sigma_2 \cup \sigma_2')(l).$$

- $l \notin L_{\text{out}1}$ and $l \notin L_{\text{out}2}$: Here we have:

$$(\sigma_1 \cup \sigma_1')(l) = \sigma_1(l) = \sigma(l) = \sigma_2(l) = (\sigma_2 \cup \sigma_2')(l).$$

We refer to this verdict as v.

Now from 3.1 we get that:

$$\partial(v) + \partial(v_1) \geq 0.$$

Similarly, we get that by 3.2 (taking the dual contract into account):

$$-\partial(v) + \partial(v_2) \geq 0.$$

Adding these two inequalities together we get:

$$\partial(v_1) + \partial(v_2) \geq 0,$$

which is exactly what we needed to show. □

This theorem allows us to finish the example from before:

Example 3.3.21 (Continuing Ex. 3.3.18)**.** From the two conforming substrategies:

$$\models \{a_1 \mapsto \tau_1\} : c_1,$$
$$\models \{a_2 \mapsto \tau_2\} : c_2, \overline{c_1},$$

we can use Theorem 3.3.20 to conclude:

$$\models \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2\} : c_2,$$

which was the result we wanted. Note that we did not have to consider the implementation of τ_1 for the conformance of τ_2 and vice versa. Therefore, we can change the implementation of e.g. τ_1 without having to redo the conformance proof for τ_2, provided that the new implementation of τ_1 still conforms with the original contract.

We summarise how to develop a conforming strategy in a modular fashion. The starting point is a contract portfolio consisting of the external contracts, as example we use c_1, c_2, c_3. Some of the contracts can easily be implemented by a single agent, whereas others might require several agents to work together. For this example we can assume that agent a_1 with τ_1 can implement c_1, and a_2 with τ_2 can implement c_2. To work together to fulfil a single contract, exactly one agent must be the main responsible for the contract, but the other agents can 'help' the main agent. The help is formalised with internal contracts, where the main agent is one party and the helping agent is the other party. In the example, a_1 and a_2 could cooperate to implement c_3, with a_1 as the

main responsible. Their internal communication contract is specified as c_{int}, and they play dual roles in that contract. All in all, the needed conformances are as follows:

$$\models \{a_1 \mapsto \tau_1\} : c_1, c_3, \overline{c_{\text{int}}},$$
$$\models \{a_2 \mapsto \tau_2\} : c_2, c_{\text{int}}.$$

From these conformances we get, by the compositionality theorem, that all three external contracts can be implemented by the two agents:

$$\models \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2\} : c_1, c_2, c_3.$$

Remark 3.3.22. Note that even though we have only looked at bilateral contracts, the result should also generalise to n-party contracts. The main generalisation is that instead of having a strategy conforming with the role of the first party in a contract, we need conformance with respect to a specific party. Say we have a 3-party contract, c_1 and a 4-party contract c_2. We can then express that a strategy, Σ, conforms with those contracts, when participating as second part in c_1 and third part in c_2:

$$\models \Sigma : [c_1]_{2/3}, [c_2]_{3/4}.$$

We write $[c]_{i/n}$ for participating as i'th party in an n-party contract. The compositionality theorem must be changed to: if substrategies participate as every party in some contracts, then those contracts can be eliminated. As example consider an external 2-party contract c. To implement that contract, the principal devises an internal 3-part contract c_{int} and then implements three tactics. The first tactic τ_1, is the main responsible for c, and participates as second part in c_{int}. The second tactic τ_2, participates as first part in c_{int}. The third tactic τ_3, participates as third part in c_{int}. The compositionality theorem must now ensure that the conformances:

$$\models \{a_1 \mapsto \tau_1\} : [c]_{1/2}, [c_{\text{int}}]_{2/3},$$
$$\models \{a_2 \mapsto \tau_2\} : [c_{\text{int}}]_{1/3},$$
$$\models \{a_3 \mapsto \tau_3\} : [c_{\text{int}}]_{3/3},$$

imply the desired conclusion:

$$\models \{a_1 \mapsto \tau_1, a_2 \mapsto \tau_2, a_3 \mapsto \tau_3\} : [c]_{1/2}.$$

3.4 Automaton model

In the last section, we defined an trace-based model for contracts and strategies. The trace-based model allows the definition of parallel composition and conformance fairly easily, but is it hard to implement concrete contracts. In this section we define an model for contracts and strategies based on automata.

We can view the trace-based model as an extensional model for general agent behaviour, while the automaton model is an intensional model for concrete agents. Additionally, the automaton-based model is easier to relate to the concrete syntactical model we introduce in Chapter 4.

As expected two models are not equivalent, contracts and strategies expressed in the automaton-based model can, however, be given a trace-based denotation. We can also define conformance for automaton-based strategies, and we prove that automaton-based conformance implies trace-based conformance of the denotations. This allows the trace-based compositionality theorem to be used for strategies defined using the automaton-based model.

3.4.1 Basic definitions

When contracts and tactics are considered as automata, they both have a notion of state and a transition function. The main idea is that the contract and the tactic automaton step as a function of the actions on the input links. But whereas the trace formulation looked at the entire history, the automaton formulation only looks at the current input, and all relevant history must be encoded in the state. This section includes the definitions needed to present the automaton model; in particular we need to define the notion of current input.

Definition 3.4.1. *An* action map *over a (finite) set of links, L, is a partial map from the links into values:*

$$m \in \mathbf{Amap}_L := \prod_{l \in L} (\mathbf{Value}_l + \{\varepsilon\}).$$

The set of all non-empty maps is:

$$\mathbf{Amap}_L^+ := \{m \in \mathbf{Amap}_L \mid \exists l.\ m(l) \neq \varepsilon\}$$

We show two example maps:

Example 3.4.2. Given the links, l_1, l_2 from example 3.3.3. Two actions maps are given below:

$$m_1 = \{l_1 \mapsto \mathsf{a}, l_2 \mapsto \varepsilon\}$$
$$m_2 = \{l_1 \mapsto \mathsf{b}, l_2 \mapsto \mathsf{z}\}$$

Both are non-empty.

To present the theory more clearly, we add the following operations for action maps:

Definition 3.4.3. *The following operations are associated with action maps:*

- $\emptyset_{\mathrm{m}}$ *is the empty map, i.e.,* $\emptyset_{\mathrm{m}}(l) = \varepsilon$.
- *For traces we have a partial cons-operation,* $\triangleright : (\mathbf{Time} \times \mathbf{Amap}_L) \times \mathbf{Trace}_L \to \mathbf{Trace}_L + \{\varepsilon\}$*, defined in the following way:* [3]

$$((t, m) \triangleright \sigma)(l) = \begin{cases} [(t, v)] +\!\!+\ \sigma(l) & \text{if } \exists v.\ m(l) = v, \\ \sigma(l) & m(l) = \varepsilon. \end{cases}$$

  The operator is defined whenever $t < \mathrm{first}(\sigma)$ *and*

$$\forall l \in L.\ m(l) \neq \varepsilon \wedge (t', v') \in \sigma(l) \Rightarrow t' - t \geq \mathrm{lat}(l).$$

- *When* $L_1 \cap L_2 = \emptyset$ *then* $\cup : \mathbf{Amap}_{L_1} \times \mathbf{Amap}_{L_2} \to \mathbf{Amap}_{L_1 \cup L_2}$*, is the union, defined in the following way:*

$$(m \cup m')(l) = \begin{cases} m(l) & \text{if } l \in L_1, \\ m(l') & \text{if } l \in L_2. \end{cases}$$

[3] We use $+\!\!+$ to concatenate two lists.

The cons-operation defined above can be used as basis for a pattern matching operation, as shown in the following proposition.

Proposition 3.4.4. *If $\sigma \neq [\,]_{\mathrm{tr}}$ then there exist unique non-empty $m \in \mathbf{Amap}_L^+$, t, and σ' such that:*

$$\sigma = (t, m) \triangleright \sigma'.$$

The unique t is exactly $\mathrm{first}(\sigma)$.

Proof. Given σ, we define:

$$t = \min\{t \mid \exists l, v.\ (t, v) \in \sigma(l)\};$$
$$m(l) = \begin{cases} v & \text{if } (t, v) \in \sigma(l), \\ \varepsilon & \text{otherwise;} \end{cases}$$
$$\sigma'(l) = \begin{cases} ts & \text{if } \exists v.\ \sigma(l) = [(t, v)] +\!\!+\ ts, \\ \sigma(l) & \text{otherwise.} \end{cases}$$

These satisfy that $\sigma = (t, v) \triangleright \sigma'$. That they are unique is easy to see. □

A couple of examples illustrate these definitions:

Example 3.4.5. Given the traces from Ex. 3.3.3 and the maps from Ex. 3.4.2, we have that:

$$(1.7, m_1) \triangleright \sigma = \big\{l_1 \mapsto [(1.7, \mathsf{a}), (3.1, \mathsf{a}), (6.5, \mathsf{b}), (8.0, \mathsf{a})], l_2 \mapsto [(4.5, \mathsf{z}), (4.8, \mathsf{z}), (5.5, \mathsf{x})]\big\}.$$

Given $m_3 = \{l_3 \mapsto \mathsf{o}\}$ we have that:

$$m_1 \cup m_3 = \{l_1 \mapsto \mathsf{a}, l_2 \mapsto \varepsilon, l_3 \mapsto \mathsf{o}\}.$$

Lastly, we see that σ can be split up in exactly one way with cons:

$$\sigma = (3.1, \{l_1 \mapsto \mathsf{a}, l_2 \mapsto \varepsilon\}) \triangleright \big\{l_1 \mapsto [(6.5, \mathsf{b}), (8.0, \mathsf{a})], l_2 \mapsto [(4.5, \mathsf{z}), (4.8, \mathsf{z}), (5.5, \mathsf{x})]\big\}.$$

Similar to the other trace operations, the cons operation preserves end time:

Proposition 3.4.6. *If $\sigma \in \mathbf{Trace}_L^t$ then (if it is defined) $(t', m) \triangleright \sigma \in \mathbf{Trace}_L^t$.*

Proof. This is immediately clear from the definition. □

With these basic definitions, we move on to the contracts.

3.4.2 Contract automata

A contract automaton has a set of states and a starting state. Furthermore, it has a transition function specifying how the state changes in reaction to input. The basic idea is that whenever something happens on one of the contract's links, the state changes. But as contracts should be able to specify deadlines and other time-based properties, the formalisation has to take timing into account. We model timing by also advancing the contract at certain fixed time points. This corresponds to adding a timer, which fires at a certain frequency, to each contract.

Definition 3.4.7. *A contract automaton, $\mathcal{C}$, for link set L, is a 5-tuple $(G, g_0, d_0, \rho_\mathrm{t}, \rho_\mathrm{i})$ where*

- *G is a (not necessarily finite) set of contract states;*
- *$g_0 \in G$ is the starting state;*
- *$d_0 \in \mathbf{TimeD}_+$ is the time between timer transitions;*
- *$\rho_\mathrm{t} : G \to G + \mathbb{R}$, is the function for timer transitions;*
- *$\rho_\mathrm{i} : G \times \mathbf{Amap}_L^+ \to G + \mathbb{R}$, is the function for input transitions.*

A contract has two transition functions; each considers the current state and returns either a new state or a pay-off, signalling that the contract has finished. The input transition function, ρ_i, takes a non-empty map of inputs and computes a new state (or pay-off) based on those. The timer transition function, ρ_t, is invoked every d_0 time units. The first timer transition takes place d_0 after the starting time. If the timer and the input happen at the same time, they are both executed, with the timer executing first. The timer transitions are used to encode deadlines in contracts. To encode a deadline, the contract implements a counter in the state, and then decrements this counter at every timer transition, until it reaches zero and a pay-off is assigned.

The input transition takes as input a collections of potentially several actions, considered to be happening at the same time point. When the time domain is the real numbers, two actions will very rarely happen at the same time. In some applications, the time domain may be discrete, so that all actions happening in an interval are considered simultaneous. An example is standard business contracts, which usually express constraints in terms of whole days. Here it is very likely that multiple actions occur at the same time point (same day). An alternative to the input transition taking a set of inputs would be to define a priority of each link and then have a separate input transition function for each link (e.g. $(\rho_\mathrm{t})_l : G \times \mathbf{Value}_l \to G + \mathbb{R}$). The priorities could, however, in general be different for each contract, and we think this would give a more complicated model. An interesting thing to consider is whether most contracts are *additive*: the order of actions between two consecutive timer steps do not matter. We leave these considerations for future work.

Example 3.4.8. As an example, we show a formalisation of the $+i$ contracts, c_i, from Ex. 3.3.18, given as $(G, g_0, d_0, \rho_\mathrm{t}, \rho_\mathrm{i})$ where:

$$
\begin{aligned}
G &= \{\mathsf{start}\} + \{\mathsf{run}(n,t) \mid n \in \mathbb{Z}, t \in \mathbf{Time}\};\\
g_0 &= \mathsf{start};\\
d_0 &= 1s;\\
\rho_\mathrm{t}(\mathsf{start}) &= \mathsf{start}\\
\rho_\mathrm{t}(\mathsf{run}(n,t)) &= \begin{cases} -1 & \text{if } t \leq 0.0,\\ \mathsf{run}(n, t-d_0) & \text{otherwise;}\end{cases}\\
\rho_\mathrm{i}(\mathsf{start}, m) &= \begin{cases} -1 & \text{if } m(l_{\mathrm{out}1}) \neq \varepsilon,\\ \mathsf{run}(n, t_{\mathrm{deadline}}) & \text{if } m(l_{\mathrm{in}1}) = n;\end{cases}\\
\rho_\mathrm{i}(\mathsf{run}(n,t), m) &= \begin{cases} 1 & \text{if } m(l_{\mathrm{in}1}) \neq \varepsilon,\\ \mathsf{start} & \text{if } m(l_{\mathrm{in}1}) = n+i,\\ -1 & \text{if } m(l_{\mathrm{in}1}) \neq n+i,\end{cases}
\end{aligned}
$$

where t_{deadline} is either 3.0 for c_1 or 10.0 for c_2. Note that compared to the, previously mentioned, trace-based +2 contract (Ex. 3.3.9) there are two differences. The first difference is that the invoker of service must not send extra requests, when there is a request under way. Similarly the service must not send any results when there are no requests. The second difference is that the deadline specified in this contract is not directly 10 seconds, but instead 11 timer transitions, corresponding to a deadline between 10 and 11 second.

Being an intensional model, contract automata are naturally more restricted than the trace-based contracts; a contract, which expresses that an action must happen at a specific time point, is only expressible in the trace-based model. The automaton model cannot express that some action must happen exactly at time π, for two reasons: firstly, contracts can only see which timer interval the actions occurred in, which seems more realistic in practice, and secondly, all timer transitions are relative to a starting time.

For trace-based contracts such a starting time is not even needed. As another example, the contract which lets the first player win if there is an even number of actions up to a given time point, can easily be specified as a trace-based contract. A similar automaton-based contract could, however, only count the actions in an interval, bounded both by a start time and an end time. To define a trace-denotation for a automaton contract, we therefore, need to specify the starting time for the automaton.

Before looking at the contract denotation, we present two auxiliary definitions. The first definition captures that while a contract automaton is running, the semantics must keep track of the contract state and the time point for next timer transition:

Definition 3.4.9 (Contract state)**.** *The running states of a contract automaton*

$$(G, g_0, d_0, \rho_{\text{t}}, \rho_{\text{i}})$$

are given by the set $G_{\text{r}} := (\mathbf{Time} \times G) + \mathbb{R}$*. When* $g_{\text{r}} = (t, g)$ *then the contract state is* g *and the time for next timer transition is* t*. When* $g_{\text{r}} = k$*, then the contract has ended with pay-off* k*.*

The next definition expresses how the running state evolves with input.

Definition 3.4.10 (Contract advancement)**.** *The stepwise advancement of a contract automaton is expressed as a function:*

$$\text{cstep} : G_{\text{r}} \times \mathbf{Time} \times \times \mathbf{Amap}_L \to G_{\text{r}},$$

defined in the following way:

$$\text{cstep}(k, t, m) = k$$

$$\text{cstep}((t_{\text{s}}, g), t, m) = \begin{cases} (t_{\text{s}}, g) & \text{if } t < t_{\text{s}} \wedge m = \emptyset_{\text{m}}, \\ k & \text{if } t < t_{\text{s}} \wedge \rho_{\text{i}}(g, m) = k, \\ (t_{\text{s}}, g') & \text{if } t < t_{\text{s}} \wedge \rho_{\text{i}}(g, m) = g', \\ k & \text{if } \rho_{\text{t}}(g) = k, \\ \text{cstep}((t_{\text{s}} + d_0, g'), t, m) & \text{if } \rho_{\text{t}}(g) = g'. \end{cases}$$

The function is well-defined, because in every recursive call t_{s} *grows with* d_0*, and the function terminates when* $t_{\text{s}} > t$*.*

We explain the defined function: when the running state of a contract is g_r, t is a time point and m is the input to the contract at time t; then $\mathrm{cstep}(g_\mathrm{r}, t, m)$ is the running state advanced up to and including the time point t. We see from the definition that if $g_\mathrm{r} = (t_\mathrm{c}, g)$ and $t_\mathrm{c} = t$ then the timer transition is handled before the input transition, as we specified previously.

A trace-based denotation of a contract automaton can now be formally defined:

Definition 3.4.11 (Contract automaton denotation)**.** *Given a contract automaton, $\mathcal{C}$, for links L. For a starting time $t_0 \in \mathbf{Time}$, the denotation $[\![\mathcal{C}]\!]_{t_0} : \mathbf{Contract}_L$ is given as:*

$$[\![(G, g_0, d_0, \rho_\mathrm{t}, \rho_\mathrm{i})]\!]_{t_0}(t_\mathrm{end})(\sigma) = f_\mathrm{c}(g_0, t_0 + d_0)(t_\mathrm{end})(\sigma_{\geq t_0})$$

where f_c is defined below:

$$\begin{aligned}
&f_\mathrm{c} : G_\mathrm{r} \to \prod_{t \in \mathbf{Time}} (\mathbf{Trace}_L^t \to \mathbf{Verdict}),\\
&f_\mathrm{c}(g_\mathrm{r})(t)([\,]_\mathrm{tr}) = \begin{cases} \bot & \text{if } \mathrm{cstep}(g_\mathrm{r}, t, \emptyset_\mathrm{m}) = g'_\mathrm{r}, \\ k & \text{if } \mathrm{cstep}(g_\mathrm{r}, t, \emptyset_\mathrm{m}) = k, \end{cases}\\
&f_\mathrm{c}(g_\mathrm{r})(t)((t', m) \triangleright \sigma) = f_\mathrm{c}(\mathrm{cstep}(g_\mathrm{r}, t', m))(t)(\sigma).
\end{aligned}$$

That this denotes a contract is proven in Prop. 3.4.13.

The intuition behind f_c is that it considers all elements in the trace in ascending time order. For every input map, the function cstep is used to advance the running state of the contract up to and including that input. When there is no more input, cstep is used to advance the contract up to and including the end time of the trace. A starting time of t_0 means that the automaton only considers actions after that time point. The incoming trace is, therefore, restricted to actions after t_0.

When a contract finishes with a pay-off because of a timer transition, any input that comes at a later time do not matter. We prove this as a simple lemma, which we use later:

Lemma 3.4.12. *If $\mathrm{cstep}(g_\mathrm{r}, t, \emptyset_\mathrm{m}) = k$ and $t \leq t'$ then $\mathrm{cstep}(g_\mathrm{r}, t', m) = k$ for any m.*

Proof. If $g_\mathrm{r} = k$, then the result follows directly, so assume $g_\mathrm{r} = (t_\mathrm{s}, g)$. Inspecting the definition, we see that only the fifth case in the definition of cstep can be hit. Therefore, we have that $t \geq t_\mathrm{s}$ and $\rho_\mathrm{t}(g) = k$. Because of the assumption, we therefore have that $t' \geq t_\mathrm{s}$ and the result follows. □

With this lemma we can prove that the automaton can be given a contract denotation:

Proposition 3.4.13. $[\![\mathcal{C}]\!]_{t_0}$ *(from Def. 3.4.11) denotes a contract.*

Proof. This amounts to showing that the function defined by f_c is monotone. So let $\sigma \in \mathbf{Trace}_L^t$ and $t \leq t'$ we need to show that:

$$\begin{aligned}
\forall \sigma \in \mathbf{Trace}_L^{t'}.\ & f_\mathrm{c}(g_0, t_0 + d_0)(t)((\sigma_{\leq t})_{\geq t_0}) = \bot \vee \\
& f_\mathrm{c}(g_0, t_0 + d_0)(t)((\sigma_{\leq t})_{\geq t_0}) = f_\mathrm{c}(g_0, t_0 + d_0)(t')(\sigma_{\geq t_0}).
\end{aligned}$$

It is easy to see that we can do induction on traces using the $\triangleright$ operator. So we wish to prove by induction on σ that for any g_r we have that:

$$f_\mathrm{c}(g_\mathrm{r})(t)(\sigma_{\leq t}) = \bot \vee f_\mathrm{c}(g_\mathrm{r})(t)(\sigma_{\leq t}) = f_\mathrm{c}(g_\mathrm{r})(t')(\sigma).$$

- $\sigma = [\,]_\mathrm{tr}$: Here either $f_\mathrm{c}(g_\mathrm{r})(t)([\,]_\mathrm{tr}) = \bot$, or $f_\mathrm{c}(g_\mathrm{r})(t)([\,]_\mathrm{tr}) = k$. In the first case we are done, in the latter case cstep is used, and using Lemma 3.4.12 we can get that $f_\mathrm{c}(g_\mathrm{r})(t')([\,]_\mathrm{tr}) = k$ as needed.

- $\sigma = (t'', m) \triangleright \sigma'$: Here either $t'' \leq t$ or $t'' > t$. In the first case we can use the induction hypothesis directly. In the latter case we have that $\sigma_{\leq t} = [\,]_\mathrm{tr}$ and therefore either $f_\mathrm{c}(g_\mathrm{r})(t)(\sigma_{\leq t}) = \bot$, or $f_\mathrm{c}(g_\mathrm{r})(t)(\sigma_{\leq t}) = k$. In the $\bot$ case we are done. In the other case by Lemma 3.4.12 we have that $\mathrm{cstep}(g_\mathrm{r}, t'', m) = k$, from which it follows that $f_\mathrm{c}(g_\mathrm{r})(t')(\sigma) = k$ as needed.

□

3.4.3 Tactic automata

Similar to contract automata, we now define tactic automata. The definition itself is very similar to the one for contracts, but instead of terminating with a pay-off, the tactic specifies which actions should be performed in each step.

Definition 3.4.14. *A tactic automaton, $\mathcal{T}$, with input/output links $L_\mathrm{in}/L_\mathrm{out}$, is a 5-tuple $(S, s_0, d_0, \delta_\mathrm{t}, \delta_\mathrm{i})$ where*

- *S is a (not necessarily finite) set of tactic states;*
- *$s_0 \in S$ is the starting state;*
- *$d_0 \in \mathbf{TimeD}_+$ is the time between timer transitions;*
- *$\delta_\mathrm{t} : S \to S \times \mathbf{Amap}_{L_\mathrm{out}}$ is the function for timer transitions;*
- *$\delta_\mathrm{i} : S \times \mathbf{Amap}^+_{L_\mathrm{in}} \to S \times \mathbf{Amap}_{L_\mathrm{out}}$ is the function for the input transitions.*

The definition is similar to the one for contract automata. The input transition, δ_i, specifies what happens when input is received. The timer transition, δ_t, specifies what happens at each timer tick. If input is received at the same time as a timer tick, the timer tick is handled first.

Each tactic can act every time there is an input. To keep the input for later use, the tactic has to save it in the state. The output of a tactic is a map that specifies what should be sent on each link. When receiving inputs fast or when timer transitions are performed quickly, the specified outputs might violate the latencies of the output links. In those cases the requested output is ignored; corresponding to a capacity of one for each link. If needed, an output buffer can be modelled in the state, or a special buffering agent can be added.

We show an example tactic automaton:

Example 3.4.15. A formalisation of the +2 tactic from Ex. 3.3.18, can be given as $(S, s, d_0, \delta_\mathrm{t}, \delta_\mathrm{i})$ where:

$$
\begin{aligned}
S &= \{\mathsf{start}, \mathsf{wait1}, \mathsf{wait2}\};\\
s &= \mathsf{start};\\
d_0 &= 1s;\\
\rho_\mathrm{t}(s) &= (s, \emptyset_\mathrm{m});\\
\rho_\mathrm{i}(\mathsf{start}) &= \begin{cases} (\mathsf{wait1}, \{l_{\mathrm{in}1} \mapsto n, l_{\mathrm{out}2} \mapsto \varepsilon\}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{in}2}) = n,\\ (\mathsf{start}, \emptyset_\mathrm{m}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{in}2}) = \varepsilon; \end{cases}\\
\rho_\mathrm{i}(\mathsf{wait1}) &= \begin{cases} (\mathsf{wait2}, \{l_{\mathrm{in}1} \mapsto n, l_{\mathrm{out}2} \mapsto \varepsilon\}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{out}1}) = n,\\ (\mathsf{wait1}, \emptyset_\mathrm{m}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{out}1}) = \varepsilon; \end{cases}\\
\rho_\mathrm{i}(\mathsf{wait2}) &= \begin{cases} (\mathsf{start}, \{l_{\mathrm{in}1} \mapsto \varepsilon, l_{\mathrm{out}2} \mapsto n\}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{out}1}) = n,\\ (\mathsf{wait1}, \emptyset_\mathrm{m}) & \text{if } \emptyset_\mathrm{m}(l_{\mathrm{out}1}) = \varepsilon; \end{cases}
\end{aligned}
$$

Note that the tactic uses a trivial timer transition function in this case. But one could easily imagine a setting, where the +2 provider receives a larger penalty, and therefore has to contact a backup +1 tactic, in case the first one fails to give an answer. In such a setting, the timer transition would be used to contact the backup service.

Similarly to contract automata, we can define a trace-based denotation for tactic automata. But as mentioned, latency has to be taken into account, and actions that do not respect the latency should be discarded. To express the denotation more clearly, we introduce a snoc operation for traces which takes the latencies into account:

Definition 3.4.16. *The operator,* $\circledast : \mathbf{Trace}_L \times (\mathbf{Time} \times \mathbf{Amap}_L) \to \mathbf{Trace}_L$ *is defined in the following way:*

$$
(\sigma \circledast (t, m))(l) = \begin{cases} \sigma(l) +\!\!+ [(t + \mathrm{lat}(l), v)] & m(l) = v \wedge \forall (t', v') \in \sigma(l).\ t' \leq t,\\ \sigma(l) & \textit{otherwise.} \end{cases}
$$

When $\sigma \circledast (t, m)$ the actions in m are attempted at time t. If the latencies are respected then an action on link l should be performed at time $t + \mathrm{lat}(l)$. Therefore to make sure that the latencies are respected, we check, whether all previous actions on the same link are performed before t. If this condition is not satisfied the attempted action is discarded. Similar to contract automata, we define the running state of a tactic, which is a state and a time point for next timer transition:

Definition 3.4.17 (Tactic state)**.** *The running states of a tactic automaton*

$$(S, s_0, d_0, \delta_\mathrm{t}, \delta_\mathrm{i})$$

is given by the set $S_\mathrm{r} := \mathbf{Time} \times S$*. When* $s_\mathrm{r} = (t, s)$*, then tactic state is* s *and the time for next timer transition is* t*.*

The stepwise advancement of a tactic is similar to the stepwise advancement of a contract, however, output has to be taken into account:

Definition 3.4.18 (Tactic advancement)**.** *The function*

$$\text{tstep} : S_\text{r} \times \mathbf{Time} \times \mathbf{Amap}_{L_\text{in}} \times \mathbf{Trace}_{L_\text{out}} \to S_\text{r} \times \mathbf{Trace}_{L_\text{out}},$$

expressing the stepwise advancement of the tactic automaton is defined in the following way:

$$\text{tstep}((t_\text{s}, s), t, m, \sigma) = \begin{cases} ((t_\text{s}, s), \sigma) & \text{if } t < t_\text{s} \wedge m = \emptyset_\text{m}, \\ ((t_\text{s}, s'), \sigma \circledast (t, m')) & \text{if } t < t_\text{s} \wedge \delta_\text{i}(s, m) = (s', m'), \\ \text{tstep}((t_\text{s} + d_0, s'), t, m, \sigma \circledast (t_\text{s}, m')) & \text{if } \delta_\text{t}(s) = (s', m'). \end{cases}$$

The function is well-defined, because in every recursive call t_s grows with d_0, and the function terminates when $t_\text{s} > t$.

When the running state of the tactic is s_r, t is a time point, m is input at time t and σ is a trace of already generated output, then $\text{tstep}(s_\text{r}, t, m, \sigma) = (s'_\text{r}, \sigma')$ means that the tactic is advanced up to and including t resulting in a new running state s'_r. Furthermore, the output generated up to t is added to the trace σ, resulting in the new output trace σ'. The input trace is used to known the latencies of the previously generated output, and by using the snoc operation defined above, we make sure that the output latencies are respected. With these two definitions, we can define a trace-based denotation of a tactic automaton:

Definition 3.4.19 (Tactic automata denotation)**.** *Given a tactic automaton*

$$(S, s_0, d_0, \delta_\text{t}, \delta_\text{i})$$

with input/output links L_in/L_out. For a starting time $t_0 \in \mathbf{Time}$, the denotation $[\![(S, s_0, d_0, \delta_\text{t}, \delta_\text{i})]\!]_{t_0} : \mathbf{Tactic}_{L_\text{in} \to L_\text{out}}$ is given as:

$$[\![(S, s_0, d_0, \delta_\text{t}, \delta_\text{i})]\!]_{t_0}(t_\text{end}, \sigma) = f_\text{t}((t_0, s_0), [\,]_\text{tr})(t_\text{end}, \sigma_{\geq t_0})$$

where f_t is defined below:

$$f_\text{t} : S_\text{r} \times \mathbf{Trace}_{L_\text{out}} \to \prod_{t \in \mathbf{Time}} (\mathbf{Trace}^t_{L_\text{in}} \to \mathbf{Trace}^t_{L_\text{out}}),$$
$$f_\text{t}(s_\text{r}, \sigma)(t)([\,]_\text{tr}) = \sigma'_{\leq t} \text{ when } \text{tstep}(s_\text{r}, t, \emptyset_\text{m}, \sigma) = (s'_\text{r}, \sigma'),$$
$$f_\text{t}(s_\text{r}, \sigma)(t)((t', m) \triangleright \sigma') = f_\text{t}(\text{tstep}(s_\text{r}, t', m, \sigma))(t)(\sigma').$$

That this denotes a tactic is proven in Prop. 3.4.21.

The function f_t is similar to f_c. If reads inputs one by one and then uses tstep to advance the tactic up to the input times. The already generated output is passed as a parameter, and when there are no more input the output is returned as a result. Because tstep do not take the end time into account, the resulting trace is truncated to the end time. Similar to contracts the starting time is handled by restriction of the input trace.

Even though a tactic is called with two different input at different times, the generated output will still match up to the smallest of the input times plus the latency. That is, all timer transitions performed before the input, must return the same output. This is captured by the following lemma:

Lemma 3.4.20. *If* $\mathrm{tstep}(s_\mathrm{r}, t_1, m_1, \sigma) = (s_\mathrm{r}^1, \sigma_1)$, $\mathrm{tstep}(s_\mathrm{r}, t_2, m_2, \sigma) = (s_\mathrm{r}^2, \sigma_2)$ *then for every* $l \in L_\mathrm{out}$ *we have that:*

$$(\sigma_1)_{< \min(t_1,t_2) + \mathrm{lat}(l)} = (\sigma_2)_{< \min(t_1,t_2) + \mathrm{lat}(l)}.$$

Proof. From the definition of tstep we can see that the only elements that can be different in each case are the ones added from the input transition. And because of the definition of $\circledast$, these are added at a time point, t', satisfying that $t' \geq \min(t_1, t_2) + \mathrm{lat}(l)$, and therefore the traces match up to (not including) $\min(t_1, t_2) + \mathrm{lat}(l)$. □

With this lemma we can prove that tactic automata can be given a trace-based denotation:

Proposition 3.4.21. $[\![\cdot]\!]$. *(from Def. 3.4.19) denotes a tactic.*

Proof. This amounts to showing that the function defined by f_t is monotone. So let $\sigma_1, \sigma_2 \in \mathbf{Trace}^t_{L_\mathrm{in}}$, $t' \leq t$ and $(\sigma_1)_{\leq t'} = (\sigma_2)_{\leq t'}$. We need to show that:

$$\begin{aligned} \forall l \in L_\mathrm{out}.\ & f_\mathrm{t}((t_0, s_0), [\,]_\mathrm{tr})(t)((\sigma_1)_{\geq t_0})(l)_{\leq t' + \mathrm{lat}(l)} \\ & = f_\mathrm{t}((t_0, s_0), [\,]_\mathrm{tr})(t)((\sigma_2)_{\geq t_0})(l)_{\leq t' + \mathrm{lat}(l)}. \end{aligned}$$

The proof is similar to Prop. 3.4.13, but we need to take both input traces into account. The induction hypothesis becomes:

$$\forall l \in L_\mathrm{out}.\ f_\mathrm{t}(s_\mathrm{r}, \sigma)(t)(\sigma_1)(l)_{\leq t' + \mathrm{lat}(l)} = f_\mathrm{t}(s_\mathrm{r}, \sigma)(t)(\sigma_2)(l)_{\leq t' + \mathrm{lat}(l)}.$$

- $\sigma_1 = \sigma_2 = [\,]_\mathrm{tr}$: This case is trivial.
- $\sigma_1 = [\,]_\mathrm{tr} \wedge \sigma_2 = (t_2, m_2) \triangleright \sigma_2'$: Here $t' < t_2$ and from Lemma 3.4.21 (with $t_1 = t$) we get that for any l the traces returned by tstep match until (not including) $t_2 + \mathrm{lat}(l)$, which means they match up to and including $t' + \mathrm{lat}(l)$. And because all output generated from σ_2' comes after $t_2 + \mathrm{lat}(l)$, we get the result.
- The other case with $\sigma_2 = [\,]_\mathrm{tr}$ is symmetric.
- $\sigma_1 = (t_1, m_1) \triangleright \sigma_1' \wedge \sigma_2 = (t_2, m_2) \triangleright \sigma_2'$: Here are two subcases, depending on whether $t_1 \leq t'$ and $t_2 \leq t'$ or $t_1 > t'$ and $t_2 > t'$. Note that the case where $t_1 \leq t'$, and $t_2 > t'$ the symmetric case cannot occur.
  - $t_1 \leq t' \wedge t_2 \leq t'$: Here $m_1 = m_2$, $t_1 = t_2$ and the result follows from the induction hypothesis.
  - $t_1 > t' \wedge t_2 > t$: Here we use that all output on a link, l, generated from σ_1' must come after $t_1 + \mathrm{lat}(l)$ and similarly from σ_2' output must come after $t_2 + \mathrm{lat}(l)$. And therefore the traces match until $t' + \mathrm{lat}(l)$ as needed.

□

3.4.4 Automaton conformance

In this section we give a definition of conformance based only on the automaton formulation of contracts and tactics. We prove that this conformance for tactic and contract automata implies trace-based conformance for their trace-based denotations. The idea behind automaton conformance is a special conformance relation for tactic states and

contract states. A relation is a conformance relation if it continuously ensures that the accumulated pay-off is non-negative, regardless of the incoming actions.

In the definition of conformance we wish to use a trace to capture all input and output that have been specified by now performed yet. To pick out the next actions that should be performed, we introduce an auxiliary notion of splitting a trace at a certain time point.

Definition 3.4.22. *Given a trace $\sigma \in \mathbf{Trace}_L$ and a time point t, we define the trace splitting function* $\mathrm{split}(\sigma, t) \in \mathbf{Amap}_L \times \mathbf{Trace}_L$ *in the following way:*

$$\begin{aligned}\mathrm{split}(\sigma, t) &= (m, \sigma') \text{ where}\\ m(l) &= \begin{cases} v & \text{if } (v,t) \in \sigma(l),\\ \varepsilon & \text{otherwise;}\end{cases}\\ \sigma'(l) &= \sigma(l)_{>t}\end{aligned}$$

The operation $\mathrm{split}(\sigma, t) = (m, \sigma')$ returns the actions at time t as m and then removes them from the trace σ. The resulting trace is returned as σ'. Using this operation, we remove from the trace all actions that have been performed, so that the next actions that must be performed, are the first ones in the trace.

With this definition, we can define automaton conformance:

Definition 3.4.23. *Given a tactic automaton $(S, s_0, d_0, \delta_\mathrm{t}, \delta_\mathrm{i})$ with input/output links $L_\mathrm{in}/L_\mathrm{out}$, and contract automata: $\mathcal{C}_1, \ldots, \mathcal{C}_n$ regarding link sets $L_1, \ldots, L_n$. In the following $i \in \{1, \ldots, n\}$, and $\mathcal{C}_i = (G^i, g_0^i, d_0^i, \rho_\mathrm{t}^i, \rho_\mathrm{i}^i)$.*

We say that a relation $R \subseteq \mathbf{Time} \times S_\mathrm{r} \times \prod_i G_\mathrm{r}^i \times \prod_{t \in \mathbf{Time}} (\mathbf{Trace}^t_{L_\mathrm{in} \cup L_\mathrm{out} \cup \bigcup_i L_i})$ is a conformance relation if it holds that whenever $(t_\mathrm{now}, s_\mathrm{r}, (g_\mathrm{r})_i, t_\mathrm{end}, \sigma) \in R$ then $t_\mathrm{now} \leq t_\mathrm{end}$ implies

$$\begin{aligned}&\text{let } (m, \sigma') = \mathrm{split}(\sigma, t_\mathrm{now}),\\ &\quad ((t', s'), \sigma'') = \mathrm{tstep}(s_\mathrm{r}, t_\mathrm{now}, m_{|L_\mathrm{in}}, \sigma'_{|L_\mathrm{out}})\\ &\quad (g'_\mathrm{r})_i = \mathrm{cstep}((g_\mathrm{r})_i, t_\mathrm{now}, m_{|L_i})\\ &\quad \sigma''' = \sigma'_{|(L_\mathrm{in} \cup \bigcup_i L_i) \setminus L_\mathrm{out}} \cup \sigma''_{< t_\mathrm{end}}\\ &\text{in } \sum [k \mid k \in (g'_\mathrm{r})_i] \geq 0 \wedge\\ &\quad (\min(\mathrm{first}(\sigma'''), t', \{t \mid (t, g) \in (g'_\mathrm{r})_i\}), (t', s'), (g'_\mathrm{r})_i, t_\mathrm{end}, \sigma''') \in R;\end{aligned}$$

We say that the tactic conforms with the contracts, starting at time $t_0 \in \mathbf{Time}$, if for any $t \in \mathbf{Time}$ and $\sigma \in \mathbf{Trace}^t_{(L_\mathrm{in} \cup \bigcup_i L_i) \setminus L_\mathrm{out}}$ there exists such a conformance relation, R, satisfying that:

$$(t_0, (t_0, s_0), (t_0 + d_0^i, g_0^i)_i, t, \sigma_{\geq t_0}) \in R.$$

We explain the idea behind the definition. When $(t_\mathrm{now}, s_\mathrm{r}, (g_\mathrm{r})_i, t_\mathrm{end}, \sigma) \in R$ the components corresponds to the following: t_now is the time for the next action in the trace or the time for the next timer transition of either the tactic or a contract; s_r and $(g_\mathrm{r})_i$ are the running states of the tactic and the contracts respectively; t_end is the designated end time for the trace; σ is a trace that contains all actions at or after t_now. σ contains both, parts of the input trace, and output that the tactic has comittet to, but not yet performed (due to latency).

In each step of such a conformance relation, we extract the actions performed at the current time point (t_{now}) from the trace σ. The action map m contains the actions at t_{now}. Both the tactic and the contracts are advanced up to t_{now} using their corresponding stepping function (tstep and cstep). The splitted trace (σ') and the output from the tactic (σ'') are combined to form the trace σ''', which contains the actions that are going to be performed in the future (later than t_{now}).

Now being a conformance relation amounts to having a uniform accumulated non-negative pay-off, which is checked by summerising the pay-off of all finshed contracts in every step. The system can change state either by processing input or by a timer transition, therefore the next time something can happen after t_{now}, is the minimum time amongst actions in the trace and timer transition times. The current time used in the next step is defined exactly as this minimum. The stepping process continues until the current time passes the designated end time of the trace (t_{end}).

We show an example of how to implement this conformance relation, and how to use it, in a concrete case in Chapter 4. To conclude this section, we show a theorem that relates the automaton-based model to the trace-based model. That is, we show that automaton conformance Def. 3.4.23 implies trace conformance (Def. 3.3.16) for the trace denotations (Def. 3.4.11 and Def. 3.4.19).

Before being able to prove the main theorem, we need lemmas that relate tstep and cstep to the splitting function. The first lemma shows that if we split the trace at the earliest time that something can happen, then calling f_{t} on the original trace and calling it on a trace, where we have used tstep up to that time, yields the same result except from the output at the splitting time point:

Lemma 3.4.24. *Given* $\sigma \in \mathbf{Trace}^{t_{\text{end}}}_{(L_{\text{in}} \cup L) \setminus L_{\text{out}}}$, $s_{\text{r}} = (t_{\text{t}}, s)$ *and given that the following relations hold:*

$$t_{\text{now}} \leq \text{first}(\sigma), \quad t_{\text{now}} \leq t_{\text{t}}, \quad t_{\text{now}} \leq t_{\text{end}}.$$

If

$$\begin{aligned} \text{split}(\sigma, t_{\text{now}}) &= (m, \sigma'), \\ \text{tstep}(s_{\text{r}}, t_{\text{now}}, m_{|L_{\text{in}}}, \sigma'_{|L_{\text{out}}}) &= (s'_{\text{r}}, \sigma'') \end{aligned}$$

then

$$f_{\text{t}}(s_{\text{r}}, \sigma_{|L_{\text{out}}})(t_{\text{end}})(\sigma_{|L_{\text{in}}}) = (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright f_{\text{t}}(s'_{\text{r}}, \sigma''_{\leq t_{\text{end}}})(t_{\text{end}})(\sigma'_{|L_{\text{in}}}).$$

Proof. We have that $\sigma_{|L_{\text{out}}} = (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma'_{|L_{\text{out}}}$. We look at cases depending on whether $\sigma_{|L_{\text{in}}}$ is empty or not:

- $\sigma_{|L_{\text{in}}} = [\,]_{\text{tr}}$: Here $\sigma'_{|L_{\text{in}}} = [\,]_{\text{tr}}$ and $m_{|L_{\text{in}}} = \emptyset_{\text{m}}$. Now we look at subcases depending on whether $t_{\text{now}} < t_{\text{t}}$ or $t_{\text{now}} = t_{\text{t}}$:
  - $t_{\text{now}} < t_{\text{t}}$: Here $\sigma'' = \sigma'_{|L_{\text{out}}}$ and $s'_{\text{r}} = s_{\text{r}}$ which gives that:

    $$\sigma_{|L_{\text{out}}} = (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma''.$$

    The result now follows from the definition.

- $t_{\text{now}} = t_{\text{t}}$: Here $\delta_{\text{t}}(s) = (s', m')$, $s'_{\text{r}} = (t_{\text{now}} + d_0, s')$ and $\sigma'' = \sigma'_{|L_{\text{out}}} \circledast (t_{\text{now}}, m')$. Furthermore, we have that:

$$\begin{aligned} & \text{tstep}(s_{\text{r}}, t_{\text{end}}, \emptyset_{\text{m}}, \sigma_{|L_{\text{out}}}) \\ = \ & \text{tstep}(s'_{\text{r}}, t_{\text{end}}, \emptyset_{\text{m}}, \sigma_{|L_{\text{out}}} \circledast (t_{\text{now}}, m')) \\ = \ & \text{tstep}(s'_{\text{r}}, t_{\text{end}}, \emptyset_{\text{m}}, (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma''), \end{aligned}$$

  from which the result follows.

- $\sigma_{|L_{\text{in}}} = (t', m') \triangleright \sigma_1$: Here we look at subcases depending on whether $t_{\text{now}} < t'$ or $t_{\text{now}} = t'$:
  - $t_{\text{now}} < t'$: Here $\sigma'_{|L_{\text{in}}} = \sigma_{|L_{\text{in}}}$ and the result can be show with an argument similar to the one for the empty trace (splitting into two cases depending on $t_{\text{now}} < t_{\text{t}}$ or $t_{\text{now}} = t_{\text{t}}$).
  - $t_{\text{now}} = t'$: Here $\sigma'_{|L_{\text{in}}} = \sigma_1$ and $m' = m_{|L_{\text{in}}}$. We get that:

$$\begin{aligned} & f_{\text{t}}(s_{\text{r}}, \sigma_{|L_{\text{out}}})(t_{\text{end}})((t', m') \triangleright \sigma_1) \\ = \ & f_{\text{t}}(s_{\text{r}}, (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma'_{|L_{\text{out}}})(t_{\text{end}})((t_{\text{now}}, m_{|L_{\text{in}}}) \triangleright \sigma'_{|L_{\text{in}}}) \\ = \ & f_{\text{t}}(s'_{\text{r}}, (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma'')(t_{\text{end}})(\sigma'_{|L_{\text{in}}}), \end{aligned}$$

    from which the result follows.

□

A second lemma show a similar thing for cstep:

Lemma 3.4.25. *Given $\sigma \in \mathbf{Trace}_L^{t_{\text{end}}}$, $g_{\text{r}} = (t_{\text{c}}, g)$ and given that the following relations hold:*

$$t_{\text{now}} \leq \text{first}(\sigma), \quad t_{\text{now}} \leq t_{\text{c}}, \quad t_{\text{now}} \leq t_{\text{end}}.$$

If

$$\begin{aligned} \text{split}(\sigma, t_{\text{now}}) &= (m, \sigma'), \\ \text{cstep}(g_{\text{r}}, t_{\text{now}}, m) &= g'_{\text{r}} \end{aligned}$$

then

$$f_{\text{c}}(g_{\text{r}})(t_{\text{end}})(\sigma) = f_{\text{c}}(g'_{\text{r}})(t_{\text{end}})(\sigma').$$

Proof. The proof is similar to, but easier than, the proof of Lemma 3.4.24. □

These lemmas allow us to prove the main theorem:

Theorem 3.4.26. *Given an tactic automaton $\mathcal{T}$, with input/output links $L_{\text{in}}/L_{\text{out}}$, an agent, a, with the same links, and contract automata $\mathcal{C}_1, \ldots, \mathcal{C}_n$, regarding links $L_1, \ldots, L_n$. If the tactic automaton conforms with the contracts at time t_0 (Def. 3.4.23) then*

$$\models \{a \mapsto [\![\mathcal{T}]\!]_{t_0}\} : [\![\mathcal{C}_1]\!]_{t_0}, \ldots, [\![\mathcal{C}_n]\!]_{t_0}.$$

Proof. For simplicity we show the theorem for the special case where $n = 1$, meaning that there is only a single contract. The proof should generalise straightforwardly to the case for multiple contracts.

So assume that the tactic conforms with a single contract. There now exists a conformance relation R, which relates the starting states. We now want to prove the following: if

$$t_{\text{now}} = \min(t_{\text{t}}, t_{\text{c}}, \text{first}(\sigma)), \quad t_{\text{now}} \leq t_{\text{end}};$$

and

$$(t_{\text{now}}, (t_{\text{t}}, s), (t_{\text{c}}, g), t_{\text{end}}, \sigma) \in R;$$

then with $\sigma_{\text{in}} = \sigma_{|(L \cup L_{\text{in}}) \setminus L_{\text{out}}}$ and $\sigma_{\text{out}} = \sigma_{|L_{\text{out}}}$:

$$\partial(f_{\text{c}}(t_{\text{c}}, g)(t_{\text{end}})((\sigma_{\text{in}} \cup f_{\text{t}}((t_{\text{t}}, s), \sigma_{\text{out}})(t_{\text{end}})(\sigma_{|L_{\text{in}}}))_{|L_i})) \geq 0.$$

If we can prove this, the result follows because R, relates the starting states which is used in the trace denotation of the automata.

We prove this result by induction, using a size measure for $t_{\text{t}}, t_{\text{c}}, \sigma$. We use the induction hypothesis when either:

1. $t_{\text{end}} - t_{\text{t}}$ gets smaller with the constant d_0^{t} (the timer transition time of the tactic); or
2. $t_{\text{end}} - t_{\text{t}}$ is constant, but $t_{\text{end}} - t_{\text{c}}$ gets smaller with the constant d_0^{c} (the timer transition time of the contract); or
3. both differences stay the same, but the number of elements in σ gets smaller.

Formally, we do induction using the lexicographic ordering on

$$(\frac{t_{\text{end}} - t_{\text{t}}}{d_0^{\text{t}}}, \frac{t_{\text{end}} - t_{\text{c}}}{d_0^{\text{c}}}, |\sigma|).$$

So let

$$(t_{\text{now}}, (t_{\text{t}}, s), (t_{\text{c}}, g), t_{\text{end}}, \sigma) \in R;$$

with $s_{\text{r}} = (t_{\text{t}}, s)$ and $g_{\text{r}} = (t_{\text{c}}, g)$. Because $t_{\text{now}} \leq t_{\text{end}}$ we can unfold R. Now using Lemma 3.4.24 on the equalities we get from R:

$$f_{\text{t}}(s_{\text{r}}, \sigma_{|L_{\text{out}}})(t_{\text{end}})(\sigma_{|L_{\text{in}}}) = (t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright f_{\text{t}}(s'_{\text{r}}, \sigma''_{\leq t_{\text{end}}})(t_{\text{end}})(\sigma'_{|L_{\text{in}}}).$$

Now put as shorthands:

$$\sigma_{\text{t}} = f_{\text{t}}(s_{\text{r}}, \sigma_{|L_{\text{out}}})(t_{\text{end}})(\sigma_{|L_{\text{in}}}),$$
$$\sigma'_{\text{t}} = f_{\text{t}}(s'_{\text{r}}, \sigma''_{\leq t_{\text{end}}})(t_{\text{end}})(\sigma'_{|L_{\text{in}}}).$$

By the definition of splitting, using the equality above, we see that:

$$\begin{aligned} & \text{split}(\sigma_{|(L_{\text{in}} \cup L) \setminus L_{\text{out}}} \cup \sigma_{\text{t}}, t_{\text{now}}) \\ = \ & \text{split}(\sigma_{|(L_{\text{in}} \cup L) \setminus L_{\text{out}}} \cup ((t_{\text{now}}, m_{|L_{\text{out}}}) \triangleright \sigma'_{\text{t}}), t_{\text{now}}) \\ = \ & (m, \sigma'_{|(L_{\text{in}} \cup L) \setminus L_{\text{out}}} \cup \sigma'_{\text{t}}). \end{aligned}$$

So now we can use Lemma 3.4.25 and conclude that:

$$f_{\mathrm{c}}(g_{\mathrm{r}})(t_{\mathrm{end}})(\sigma_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup \sigma_{\mathrm{t}}) = f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})(\sigma'_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup \sigma'_{\mathrm{t}}).$$

Now we look at two cases depending on whether the minimum is smaller or larger than t_{end} (in the following let $s'_{\mathrm{r}} = (t'_{\mathrm{t}}, s')$ and $g'_{\mathrm{r}} = (t'_{\mathrm{c}}, g')$):

- $\min(\mathrm{first}(\sigma'''), t'_{\mathrm{t}}, t'_{\mathrm{c}}) > t_{\mathrm{end}}$: here $t'_{\mathrm{t}} > t_{\mathrm{end}}$, $t'_{\mathrm{c}} > t_{\mathrm{end}}$ and $\sigma''' = [\,]_{\mathrm{tr}}$. Furthermore, one can see that $\sigma'_{\mathrm{t}} = [\,]_{\mathrm{tr}}$, from which it follows that:

  $$f_{\mathrm{c}}(g_{\mathrm{r}})(t_{\mathrm{end}})(\sigma_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup \sigma_{\mathrm{t}}) = f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})([\,]_{\mathrm{tr}}).$$

  Now we have that if $f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})([\,]_{\mathrm{tr}}) \neq \bot$ then $f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})([\,]_{\mathrm{tr}}) = g'_{\mathrm{r}}$ and therefore the result follows.

- $\min(\mathrm{first}(\sigma'''), t'_{\mathrm{t}}, t'_{\mathrm{c}}) \leq t_{\mathrm{end}}$: in this case we can use the induction hypothesis, because either $t'_{\mathrm{t}} = t_{\mathrm{t}} + d_0^{\mathrm{t}}$ or $t'_{\mathrm{c}} = t_{\mathrm{c}} + d_0^{\mathrm{c}}$ or the first elements in σ''' are removed. By IH we get (using that $\sigma'''_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} = \sigma'_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}}$ and $\sigma'''_{|L_{\mathrm{out}}} = \sigma''_{\leq t_{\mathrm{end}}}$):

  $$\partial(f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})((\sigma'_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup f_{\mathrm{t}}(s'_{\mathrm{r}}, \sigma''_{\leq t_{\mathrm{end}}})(t_{\mathrm{end}})(\sigma'_{|L_{\mathrm{in}}}))_{|L})) \geq 0.$$

  And from the equations above we get:

  $$\begin{aligned}
  &\partial(f_{\mathrm{c}}(g_{\mathrm{r}})(t_{\mathrm{end}})(\sigma_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup \sigma_{\mathrm{t}})) \\
  &= \partial(f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})(\sigma'_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup \sigma'_{\mathrm{t}})) \\
  &= \partial(f_{\mathrm{c}}(g'_{\mathrm{r}})(t_{\mathrm{end}})((\sigma'_{|(L_{\mathrm{in}}\cup L)\setminus L_{\mathrm{out}}} \cup f_{\mathrm{t}}(s'_{\mathrm{r}}, \sigma''_{\leq t_{\mathrm{end}}})(t_{\mathrm{end}})(\sigma'_{|L_{\mathrm{in}}}))_{|L})) \geq 0.
  \end{aligned}$$

  As needed to conclude the induction proof.

□

This theorem binds the two models together, and it ensures that we can use the automaton model for concrete tactics and contracts.

3.5 Related and future work

In this section we consider related and future work.

3.5.1 Related work

Verification for concurrent and distributed systems is an extensively studied area. Our focus in this work, however, is on logically distributed systems and adversarial composition of multi-principal systems. Therefore, we do not do a complete survey of existing models for concurrency and distribution, but only discuss a small portion of this work.

Models of concurrency. Models for concurrent systems can be grouped in two categories according to Cleaveland and Smolka [16]: the *intensional* models and the *extensional* models. Intensional models describe what systems do, e.g. in terms of states and transitions between those. Extensional models describe the behaviour of the systems from the view of an external observer. That is, first a notion of observation is defined, and then systems are represented in terms of the observations that can be made

about them. The automaton-based model presented here is an example of an intensional model, whereas the trace-based model is an extensional model.

Well known intensional models include the various forms of process calculi (e.g. CSP [46], CCS [63], the π-calculus [64]). The processes in these calculi are built up from a set of primitives and operators e.g. parallel composition. Communication is by message passing on channels.

Compared to the process calculi, our communication model is based on different assumptions. In the process calculi, processes do not communicate (at least in the synchronous version), until both sender and receiver are ready. In our model a sender can always send a message on a non-busy link, and it is up to the contracts to decide whom to blame, in case the message is not acted upon. Another difference is that the automaton model presented here does not have concrete primitives or operators. We do, however, specify a concrete language in Chapter 4.

Another well-known intensional formalism is I/O automata [54], where the communicating processes are automata. Instead of a specific language, each automaton has an abstract set of states and a transition relation expressing how the state change. This is very similar to our automaton model, and the timed versions of I/O automata [10, 50] served as inspiration for our concrete automaton model.

Extensional models specify the behaviour in terms of the observations one can make of a system. A classical way of capturing these time-dependent observations is by means of a *trace*, an example hereof is the trace-based semantics of CSP [46]. Our trace-based model is more abstract, as we do not have any concrete process language. We also distinguish between an input trace and an output trace. Another extensional model is event structures [102], which is a more denotational approach. Each agent is specified in terms of sets of events, using a consistency predicate and an enabling relation. Event structures also forms the basis for *Dynamic condition response graphs* [41], which, as part of the TrustCare project, has been applied to the two of the previously mentioned examples (Ex. 2.1.3 and Ex. 2.1.4).

Distributed systems. Several models are able to capture physically distributed systems, but we have not found any focusing on logically distributed systems. Frameworks for physically distributed systems include the Distributed Join-Calculus [31, 32]. In the Distributed Join-Calculus, processes belong to a location, which again belong to a physical site. These locations can move between sites, and the model also includes a notion crash of a site, which causes all associated locations to fail. The ambient calculus [14] is another example of a physically distributed system. In the ambient calculus, processes run inside *ambients*; that can be viewed as locations. Furthermore, processes can change location at runtime. Our model can express physical separation though assignment of latencies. As the latencies are fixed, there is, however, no notion of mobility at runtime.

Verification. There are several formalisms used for verifying these formal models. The formalisms can be simulation based or specification based. Simulation-based formalisms model the intended behaviour of a process with another (perhaps non-deterministic) process. The objective of verification is to prove that the behaviour of the process in question is included in the behaviour of the goal process. This inclusion can be formalised in many ways: bisimulation (e.g. for CCS [62]) is a commonly used method; behavioural containment (e.g. for the input output timed automata [10]) is another method.

The specification-based formalisms use a separate language of specifications (in contrast to the simulation-based formalisms, where the specification is a "reference implementation"). Each specification expresses the intended behaviour of a system, and are usually not executable in any way. Verification is the task of proving that a process conforms with its specification. Our model is a specification-based formalism, in which the specifications are called contracts and processes are agents. There are several classes of specifications. Some specifications are based on different logics (e.g. temporal logics [84]). Other approaches can be based on type-systems; session-types [47] is a commonly used type-based framework. Type-based approaches are usually decidable, which makes the verification much easier. However, because of decidability, some properties cannot be captured. Lastly, there are also approaches based on program logics, that is a set of syntactic rules for expressing properties of the programs. Examples of such systems based on syntactic rules include early work by Owicki and Gries [75] and more recent work by Hooman [48]. In this chapter we have only presented the formal mathematical model, and there is no special support for verifying conformance. In Chapter 4 we present a method for proving conformance based on a syntactic system. This puts the method in the same category as the work by Hooman, but instead of requiring a special custom built program logic, we can apply standard Floyd-Hoare logic. We elaborate more on this in Chapter 4.

Formal business processes. In the area of formalising business processes, there have been great deal of work. Formal workflows have been considered by several researchers. Two well-known formalisms are the industry standard WS-BPEL [4] and the academically developed YAWL [95]. In comparison to our model, these workflow-based methodologies take the workflow as the specification, and seek to find implementations or programs that follow them. We view workflows as means for satisfying certain constraints (e.g. guidelines, rules or laws), which make them closer to an implementation and not a specification. Lately, declarative workflows [96, 97] have emerged. Such workflows are closer to the actual constraints and can be seen as first steps towards getting the actual requirements formalised, and not just a single implementation of them.

For dealing with interactions in these business processes, formal methods based on web service choreographies have been investigated. A choreography is a global view of an interacting system, and the international standard for such choreographies is the Web services choreography description language (WS-CDL) [101], and a formal model is given by Busi et al. [11]. An appeal of the choreography-based approach is that a global view can be automatically transformed into a local view for each participant, as shown by Cabone et al. [12]. Compared to our approach, this approach is fundamentally cooperative, as the goal of each participant is to fulfil the goal specified as the global view. Our method is fundamentally adversarial, which we believe is needed for a realistic account for such interacting systems.

Game theory. Related to our game-theoretic interpretation of specifications, is the theory of game semantics [2], and especially the application of game semantics for software verification [1]. In game semantics a logical formula or programming-language type determines a game-like protocol for interacting with values of that type. Our model uses games in a traditional economic sense [99], to model incentive-driven behaviour.

3.5.2 Future work

The work presented here have several natural directions for future work:

Model refinement. The presented models could be refined, without changing them in any major way. One thing to look at could be inclusion of n-ary contracts in the composition theorem (see Remark 3.3.22). It would also be interesting to study the automaton model more: it should be possible to, e.g., define a parallel composition for automaton tactics and actually prove a compositionality theorem for the automaton model as well. In earlier work [38], we did just that, albeit for a simpler model.

Model extension. In addition to minor changes to the models there are several extensions that would be interesting to consider. Perhaps the most interesting direction is an extension to a more dynamic system, in which the different components can be created, destroyed and modified at runtime. Several directions exist: the first, and probably easiest, direction is to consider how to formalise the notion of roles and channels, we have used in the examples. One way would be to operate with another level of contract called a *contract template*. Such a template is specified in terms of roles and channels, which can be instantiated with concrete principals and links just before runtime. Another direction is to consider a dynamic communication topology where links can be created, destroyed and moved during runtime. We consider this in Chapter 5. Lastly, also contracts can also be dynamic. An interesting idea is to use a meta-contract, which specifies the rules for creating new contracts. Such a meta-contract contains a sort of interpreter for object-contracts. When a given contract is accepted by all parties, it becomes part of this meta-contract. Along the same line one, could also consider simulating creation of new agents though a special contract with an 'agent factory'.

Another direction for extension of the model is to consider alternative pay-off scenarios. A problem in some cases is that it can be hard to come up with a fixed price for bad events. In other cases it might not make sense to add penalties, and another pay-off domain might be better. An alternative could be a domain where some penalties and rewards can be added, and others cannot.

A last direction is how to handle physical resources. In our models, all actions can always be performed, but if actions represent events using physical resources, the resources may not always be available. Consider the example of transferring money. It does not make sense to model a physical cash transfer as an action, because all actions contain only data. Instead, we introduce a special bank principal. All actors wishing to exchange money then have contracts with this bank. Each contract specifies what the balance is, and how to interact with the account. When a specific principal then wishes to pay for something, he contacts the bank and requests a transfer. If the balance is big enough, then the bank performs the actual transfer, by updating both contracts. In turn, this enables the receiver of the transfer to perform his own transfers. Bank contracts do not have to be linear in their use of money, it is perfectly valid to include e.g. interest in the contracts. This idea can be generalised to all resource types, not only money, by using a resource manager principal similar to the bank principal. This resource manager ensures that only those transfers that make sense in the real world, actually happens. In Chapter 5 we present preliminary work on a model with first-class resources instead of these resource manager principals.

Model related directions. The models presented here are just mathematical functions. They act as the foundation for specifying interaction scenarios. In is interesting to consider how to map existing high-level languages into this model, both as tactics and as contracts. This might make it easier to express concrete scenarios, and it might be possible to get properties directly by translation, e.g. that the trace-denotation is latency respecting.

Another direction to investigate is contract analysis. In some settings we might not have a conforming strategy, because we can not prevent some errors. In those cases it might be possible to consider probabilistic models, e.g. if we know the probabilities of different strategies by our opponents, then we can analyse how our tactic fares in the average case. However, care has to be taken when the probabilities depend on each other, e.g. if the probability of a single bank defaulting is p, then one might think that the probability of n banks defaulting is p^n, but in reality the banks are connected, and therefore the real probability is much higher.

It would furthermore be very interesting to investigate the connection to traditional game theory for contract analysis. Game theory has its own notion of strategies and concepts like dominant strategy, which might provide inspiration for how to compare and evaluate several strategies for the same set of contracts.

Lastly, it is also interesting to look at general methods for showing conformance of tactics. We show one approach in Chapter 4, but there might be several other methods that can be used, especially if the tactics and contracts are written in high-level languages.

Practical application. Perhaps the most important part of future work is the application of the models to real-life problems or scenarios. We already considered some examples in Chapter 2, but as mentioned before; to give a proper implementation of the adversarial paradigm, one needs to go back to the domain experts to obtain the information that was lost in the translation to a cooperative model.

Such practical applications might include real-time programs, interaction specifications, protocols, workflows, business processes etc. In particular it is very important to investigate how business processes can be implemented as strategies, and how the underlying rules, can be implemented as contracts. In such practical examples, it would also be interesting to investigate how many errors or inconsistent behaviours that could be discovered by an adversarial analysis, and whether they could be solved in practice.

Chapter 4

Certification

In this chapter we investigate a model for certifying concrete programs with respect to the model presented in Chapter 3. The layout of this chapter is as follows: we start by giving a short background section on certification, then present the basic idea behind our certification approach. The approach is based on expressing both tactics and contracts in a small coroutine language, whose syntax and semantics we present. We then show how to use programs in that language to codify conformance, and a Floyd-Hoare logic, which we use as a basis for a concrete certification approach. Lastly, we show a small proof-of-concept implementation of the certification framework and demonstrate it with a simple tactic and two contracts. We finish with a discussion of related and future work.

4.1 The certification paradigm

A certificate, in the context of certified code, is a mechanised proof, witnessing that the piece of code satisfies some property, such as correctness (the code computes the right function) or safety (the code does not perform any forbidden actions). Such certificates are not to be confused with cryptographic certificates, in which an authority documents that a piece of code has not been altered or corrupted since it was signed. Such a signing process does not say anything about the code itself.

4.1.1 Background

Certified code is primarily used in the context of *proof-carrying code* (PCC) [67]; a second possible usage is discussed below. PCC is a paradigm, in which an untrusted piece of executable code can be equipped with a formal, machine-checkable proof of safety. There are several appealing features of the PCC paradigm:

- When the code is formally certified, there is no need for runtime checks or sandboxing.
- The task of checking a safety proof is significantly smaller (and decidable) than the task of constructing such a safety proof; therefore, the strain put on the code consumer is smaller compared to other code verification approaches.
- A proof for a specific program can be generated once by the code producer, and then independently checked by several consumers, meaning that the hard parts are only done once.

- The code producer usually has access to more invariants about the code, e.g. he might know that it is written in a statically typed language, and can then exploit the type information to generate a safety proof. (The feasibility of the type approach were demonstrated for a type-safe subset of C [68, 69], and for a large fragment of Java [17].)

An variant of PCC is called *foundational proof-carrying code*(FPCC) [6]. In FPCC the machine semantics and the safety theorem are also mechanised. This eliminates the need to trust the soundness of the proof system used. In the original formulation of FPCC, the machine semantics and the safety theorem were mechanised in the same logic, needing a higher-order logic, and making the proofs quite complex. Later Crary and Sarkar [18] showed a foundational approach using an object logic (LF [35]) to specify the machine semantics, and a metalogic (Twelf [81]) to formalise the safety theorem. Another advantage of the foundational approach is that the code producer can employ his own method for checking safety, as long as he proves his method correct. An example could be a safe type system: the code producer defines a type system and then gives a machine checkable proof that all programs that are well-typed are safe. This proof can then be checked once, and the code producer need only demonstrate typeability of the programs, rather than thier explicit safety. Another idea would be to implement a safe interpreter, and prove that the interpreter program is safe for any input program. There is actually an entire spectrum of methods, depending on whether they have a large program-dependent part (e.g., a full soundness proof for a single program) or a large program-independent part (e.g., a safe interpreter or something in the middle like a type-system). This spectrum was investigated further by the author in his master's thesis [36].

Outside the PCC world, certified code is rarely used. But we believe that certified code in a wider sense could have other applications besides PCC. One application could be for documentation; one can imagine that when work procedures or workflows become more and more electronic, formal certificates can be seen as licensing requirements, expressing that the processes conform with the rules and laws. Another application is modification of a running system, e.g. because the underlying work processes change. In such a case it might be easier to make sure that the modified system is error-free, if the proof that the old system is error-free is still available, because parts of the old proof could be reused. A barrier that prevents certified code from being more widely used is that in order to support a certification-based method, all programs or processes have to be formally proven correct for the paradigm to work, which in practice is often not realistic (e.g. the code might not be available). We believe that the economic based model allows certified and uncertified components to be mixed, thereby lowering the barrier for application of a certification-based methodology.

Certified code for sequential programs is a relatively well-understood field. However moving to a concurrent and in particular distributed setting, in which the programs are generalised to real-time communicating processes, comes with additional technical challenges. Examples include how to certify that these processes satisfy complex timing constraints, and how to handle asynchrony between different processes (asynchrony is often occurring in the distributed delegation settings). In the presence of concurrency, a clear advantage of a certification based approach, as opposed to traditional verification based approaches (e.g. timed automata), is that the methods applied do not have to be decidable. The proof obligation is on the code producer, which has the power to modify the program if it cannot be certified. The proof check, however, should always

be decidable and easy. In the next section we introduce a general methodology to tackle the challenges of certification, without the need for specialised logics (e.g. temporal logics).

4.1.2 Verification-time monitoring as certification paradigm

Traditionally, when doing certification, one defines a language for programs (e.g. machine code), then develops a logic in which it is possible to express the properties one wishes to certify (also called the safety policy), and lastly develops a sound proof system to express proofs for concrete programs. In a foundational approach, there is an extra step, in which the semantics of the program language is formalised so that the soundness of the proof system can also be machine checked. The general goal is to take a semantic property, e.g. that the program does not index an array out of bounds, and then define a syntactic judgement that can be checked independently of the program semantics.

We wish to follow the same method, viewing contracts and conformance as the safety policy. The main challenge is, therefore, how to devise a syntactic method for proving conformance. The rest of this section gives a high-level description of our approach.

We want to reuse as much of the standard Floyd-Hoare paradigm as possible. Floyd-Hoare logic is more than 40 years old [30, 45], and is well-understood and generally accepted. Concretely, we propose a paradigm, which we call *verification-time monitoring* due to the similarities with run-time monitoring.

In verification-time monitoring, we exploit a property of our safety policy (the conformance relation), namely that *failures are finite*, which means that if our implementation can fail (attain a negative accumulated pay-off), then there is a finite sequence of timed inputs, which demonstrates this failure. On the other hand, this also means that if there is no finite sequence of inputs that results in a failure, then the implementation is safe. With this property we can write a *test harness*: a program which simulates a communication environment around the implementation, and tries to get the implementation to fail. If a failure is found, the test harness returns an error. Therefore, the problem of proving that the implementation is safe is reduced to proving that the test harness never returns an error, and prevention of such error states in a program is a generally better understood problem.

In our concrete setting we can take the verification-time monitoring idea, and further refine it. First, we can view the test harness as consisting of two components: the contracts and a *supervisor*. The contracts monitor the inputs and outputs of the tested program and occasionally return verdicts. The supervisor is the glue which holds the entire system together. The general setup is a follows: the supervisor reads the inputs including timing information from a *test script*; those inputs are given to the program being tested, which in turn produces the additional outputs needed for the evaluation of the contracts. The supervisor tallies the pay-offs for the different contracts and fails if the accumulated pay-off becomes negative.

For certification, we abstract away from a concrete test script and prove that the supervisor never signals an error for any test script. Exploiting the similarities between tactics and contracts, we can view the entire system (tactic, contracts and supervisor) as a single sequential process. With this view, we can use well-known techniques, in particular Floyd-Hoare logic, for certification of this sequential process. In particular, we can employ an off-the shelf external theorem prover to check the bulk of the verification conditions. Based on this approach, it is therefore easy to get a sound proof-system. In some sense this approach is a variant of the foundational approach, the variation

being that the safety policy is formalised directly as an executable program and not in a special logic. To get a full foundational approach, we could mechanise the semantics of the target language as well.

To apply Floyd-Hoare logic, we require that tactics and contracts are written in the same (target) language. Note, however, that they could stem from different high-level source languages. If the supervisor is also written in that language (basically implementing the conformance relation of Def. 3.4.23), the entire program can then be annotated as a single whole, similar to how a normal program is proven correct with Floyd-Hoare logic. In essence, we program a simulation of that part of the system, which is relevant for a given principal. We then use standard techniques to prove that the simulation never reaches a certain state.

Lastly, we note that if provable safety is not required, the framework can also be used for ordinary testing by simulation, as one could just feed the supervisor with different test scripts and monitor the results. Similarly, one can also do verification by trying to abstract the state space across the entire execution. The state space will of course not be finite, both due to infinitary data, but also due to the interplay of time. The first reason could possibly be handled by standard techniques, but the other reason seems hard to do in practice. In the cases where it is hard to find a decidable verification algorithm, a certification-based approach could be used instead.

4.2 Language

In this section we present the common language we use for tactics, contracts and the supervisor. We start by giving a couple of criteria for the design of the language:

1. Simplicity: to make the development of the framework as easy as possible, we seek a simple language. As written before, we do not envision real-life tactics or contracts to be written directly in this language, but we still want to make it possible to express some basic examples in a reasonably natural style.

2. Proof-support: we want a language, where there is a well-understood way of proving that programs are error-free.

3. Suitable features: in order to model the simulation of both tactics and contracts, we need several features of the language:

   (a) Support for real-valued expressions, to model time.
   (b) A way of modelling the trace in the conformance relation.
   (c) A way of defining the tactic and the contracts independently of each other, while still allowing the supervisor part to execute both.

With these criteria taken into account, we have chosen a simple coroutine language, based on mutually recursive functions, where each function call is restricted to a tail-call. Each tail-call can be considered a jump with arguments, and the function abstraction is mostly a help in handling the scope of the variables. Because of the closeness to an imperative language, we can define a Floyd-Hoare logic, which can be used to prove error-free execution. We enforce a type system to help simplify the Floyd-Hoare logic, and we allow the definition of custom data types, which can be used to ensure that the state of the tactic and the contracts are hidden from each other, and to model the incoming trace in the conformance relation. The next sections formally define the language.

$$
\begin{array}{rll}
\mathrm{Var} \ni x, \quad \mathrm{Fvar} \ni f, \quad \mathrm{Tag} \ni k & & \text{(Variables, names \& tags)} \\
\mathbb{Z} \ni i, \quad \mathbb{R} \ni r & & \text{(Integers \& reals)} \\
\mathrm{Lit} \ni q & ::= \bar{i} \mid \bar{r} \mid \texttt{true} \mid \texttt{false} & \text{(Literals)} \\
\mathrm{Opr} \ni o & ::= \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots & \text{(Operators)} \\
\mathrm{Exp} \ni e & ::= x & \text{(Variables)} \\
& \mid k(e_1, \ldots, e_n) & \text{(Data type construction)} \\
& \mid q & \text{(Constants)} \\
& \mid o(e_1, \ldots, e_n) & \text{(Operators)} \\
\mathrm{Pattern} \ni p & ::= k(x_1, \ldots, x_n) & \text{(Pattern)} \\
\mathrm{Com} \ni c & ::= f(e_1, \ldots, e_n) & \text{(Tail-call)} \\
& \mid \texttt{case}\ e\ \texttt{of}\ p_1 \to c_1 \mid \cdots \mid p_n \to c_n & \text{(Case)} \\
& \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \text{(If-then-else)} \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ c & \text{(Let)} \\
& \mid \texttt{fail} & \text{(Failure)} \\
& \mid \texttt{done} & \text{(Success)} \\
\mathrm{Decl} \ni d & ::= f(x_1, \ldots, x_n) = c & \text{(Function declaration)} \\
\mathrm{Decls} \ni ds & ::= d_1 \cdots d_n & \text{(Declarations)}
\end{array}
$$

Figure 4.1: The syntax of the language.

4.2.1 Syntax

Below we present the syntax of the language. A program consists of a set of functions. We only allow static tail-calls for each function. The language has simple values (integers, reals and booleans) and recursive data types. The syntax of the language is given in Fig. 4.1, we do not, however, specify all possible operations, as it should be easy to see how to extend the language with those. We briefly comment on the constructions in the language. The language has both integer, real and boolean constants. In addition, a datatype expression can be created by applying a constructor to a list of arguments. Commands terminate with either a tail-call, failure or success. Both failure and success is used to terminate the program. In the case construction, we require all tags to be distinct, and similarly we require all variables in the same pattern or declaration to be distinct as well. A full program will generally be a set of declarations, with either a special start function or a command to evaluate from the start.

4.2.2 Dynamic semantics

In this section we describe evaluation of the different constructs in the language. The values in the language are either integers, reals, booleans or composite tagged values:

Definition 4.2.1 (Values and environments)**.** *The values are described by the following grammar:*

$$\mathcal{V} \ni \nu ::= \ q \mid k(\nu_1, \ldots, \nu_n)$$

$\boxed{\rho \vdash e \downarrow \nu}$

$$\frac{\rho(x) = \nu}{\rho \vdash x \downarrow \nu}$$

$$\frac{\rho \vdash e_1 \downarrow \nu_1 \qquad \cdots \qquad \rho \vdash e_n \downarrow \nu_n}{\rho \vdash k(e_1, \ldots, e_n) \downarrow k(\nu_1, \ldots, \nu_n)}$$

$$\frac{}{\rho \vdash q \downarrow q}$$

$$\frac{\forall i.\ \rho \vdash e_i \downarrow \nu_i \qquad o \vdash \nu_1, \ldots, \nu_n \downarrow \nu}{\rho \vdash o(e_1, \ldots, e_n) \downarrow \nu}$$

Figure 4.2: Evaluation of expressions. $i \in \{1, \ldots, n\}$.

Is is easy to see that the values are a subset of the expressions. Variables are bound to values from this set, so a variable environment is a finite mapping from variables to values:

$$\rho \in \text{Env} := \text{Var} \rightharpoonup_{\text{fin}} \mathcal{V}.$$

Expressions are evaluated into values. For the operators, we assume the presence of an evaluation judgement $o \vdash \nu_1, \ldots, \nu_n \downarrow \nu$, which expresses how to evaluate the different operators. The rules for addition could e.g. be:

$$\frac{}{+ \vdash \overline{i_1}, \overline{i_2} \downarrow \overline{i_1 + i_2}} \qquad \frac{}{+ \vdash \overline{r_1}, \overline{r_2} \downarrow \overline{r_1 + r_2}}$$

The evaluation of an expression is captured by a big-step judgement, $\cdot \vdash \cdot \downarrow \cdot \subseteq \text{Env} \times \text{Exp} \times \mathcal{V}$, defined by the rules in Fig. 4.2. The rules are straightforward: variables are evaluated to their value in the environment, a constructor is evaluated by evaluating all the arguments. Constants evaluate to themselves, and operators are evaluated using the judgement mentioned above.

The execution of a command is defined as an environment modifying small-step semantics. The judgement, $\cdot \vdash \langle \cdot \rangle \rightarrow \langle \cdot \rangle \subseteq \text{Decls} \times \text{State} \times \text{State}$, where $\text{State} = \text{Com} \times \text{Env}$ is defined by the rules in Fig. 4.3. A tail-call steps to the body of the corresponding function; the environment is created from the arguments to the function. A pattern-matching case finds the matching constructor, and then steps to the corresponding branch with the environment updated with the matched values. If-then-else steps to either the then branch or the else branch, depending on whether the boolean expression evaluates to true or false. A let-expression evaluates the expression to a value and binds the variable to that value. The fail and success constructions terminate the program and therefore do not step to anything.

4.2.3 Type system

In this section we present a type system that ensures that programs only fail by stepping to the `fail` tail-expression. The type system also ensures that every well-typed expression will have a well-defined value. This, in turn, greatly simplifies the Floyd-Hoare

$\boxed{ds \vdash \langle c, \rho\rangle \to \langle c', \rho'\rangle}$

$$\frac{\forall i.\ \rho \vdash e_i \downarrow \nu_i \qquad f(x_1, \ldots, x_n) = c \in ds}{ds \vdash \langle f(e_1, \ldots, e_n), \rho\rangle \to \langle c, [x_1 \mapsto \nu_1, \ldots, x_n \mapsto \nu_n]\rangle}$$

$$\frac{\rho \vdash e \downarrow k(\nu_1, \ldots, \nu_m) \qquad \begin{array}{c} p_i = k(x_1, \ldots, x_m) \\ \rho' = \rho[x_1 \mapsto \nu_1, \ldots, x_m \mapsto \nu_m] \end{array}}{ds \vdash \langle \texttt{case}\ e\ \texttt{of}\ p_1 \to c_1 \mid \cdots \mid p_n \to c_n, \rho\rangle \to \langle c_i, \rho'\rangle}$$

$$\frac{\rho \vdash e \downarrow \texttt{true}}{ds \vdash \langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \rho\rangle \to \langle c_1, \rho\rangle}$$

$$\frac{\rho \vdash e \downarrow \texttt{false}}{ds \vdash \langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \rho\rangle \to \langle c_2, \rho\rangle}$$

$$\frac{\rho \vdash e \downarrow \nu}{ds \vdash \langle \texttt{let}\ x = e\ \texttt{in}\ c, \rho\rangle \to \langle c, \rho[x \mapsto \nu]\rangle}$$

(No rules for `fail`, `done`.)

Figure 4.3: Execution judgement for command. $i \in \{1, \ldots, n\}$.

logic we consider later, as we do not have to account for partial expressions or partial assertions.

The types in the program will come from two categories: the basic types (the integers, reals and booleans) and the tagged types. To capture the name of a tagged type, we need a countable set of type names. To define data types, we add type declarations, and for simplicity, because we do not want to get into type inference, we modify function declarations to include typing information. The syntax of the type system is given in Fig. 4.4. A type is either one of the basic types or a data type. Each data type is defined with a set of constructors, and their declaration specify the types of the arguments for each tag. For simplicity, we again assume that all type names and tags in the type declarations are distinct.

$$\begin{array}{rll}
\text{Tname} \ni tn & & \text{(Type names)} \\
\text{Basic} \ni \beta & ::= \texttt{int} \mid \texttt{real} \mid \texttt{bool} & \text{(Basic types)} \\
\text{Type} \ni \tau & ::= \beta \mid tn & \text{(Types)} \\
\text{Tcon} \ni c & ::= k : \tau_1, \ldots, \tau_n & \text{(Type constructor)} \\
\text{Tdecl} \ni td & ::= \texttt{type}\ tn = c_1 \mid \cdots \mid c_n & \text{(Data type declaration)} \\
\text{Tdecls} \ni tds & ::= td_1 \cdots td_n & \text{(Data type declarations)} \\
\text{Decl} \ni d & ::= f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c & \text{(Annotated function declaration)}
\end{array}$$

Figure 4.4: The syntax of the type system.

Each variable is given a type, which is captured by a typing environment:

Definition 4.2.2. *A typing environment is as follows:*

$$\Gamma \in \text{Tenv} := \text{Var} \rightharpoonup_{\text{fin}} \text{Type}.$$

From a list of data type declarations, one can extract a type definition environment in a straightforward way:

Definition 4.2.3. *A type definition environment is as follows:*

$$\Theta \in \text{Tdef} := \text{Tname} \rightharpoonup_{\text{fin}} (\text{Tag} \rightharpoonup_{\text{fin}} \text{Type}^*).$$

Lastly, we can also create a function type environment from the annotated function declarations:

Definition 4.2.4. *A function type environment is a follows:*

$$\Psi \in \text{Tfenv} := \text{Fvar} \rightharpoonup_{\text{fin}} \text{Type}^*.$$

Because only tail calls are allowed, there is no return type, and the list contains the type of the parameters.

In essence, each function is just a label, and the function type environment specifies the type of the live variables.

Similar to how we assumed the presence of an evaluation judgement for operators, we assume that there is a typing judgement for operators $\vdash o : \tau_1, \ldots, \tau_n \to \tau$, expressing the type rules for each operator. E.g. the rules for addition would match the evaluation ones:

$$\frac{}{\vdash \texttt{+} : \texttt{int}, \texttt{int} \to \texttt{int}} \qquad \frac{}{\vdash \texttt{+} : \texttt{real}, \texttt{real} \to \texttt{real}}$$

With the data type environment and the typing environment, expressions can be given a type, captured by the judgement, $\cdot, \cdot \vdash \cdot : \cdot \subseteq \text{Tdef} \times \text{Tenv} \times \text{Exp} \times \text{Type}$, defined by the rules in Fig. 4.5. The type of a variable is given by the environment. To type a constructor, the type of the arguments must match the ones given in the type declaration, the result type is then the name of the type. Constants have their basic type. The operators use the operator typing judgement introduced above.

For the typing of commands, we need to include the function type environment in order to type the tail-calls. The typing judgement for commands $\cdot, \cdot, \cdot \vdash \cdot : \texttt{com} \subseteq \text{Tfenv} \times \text{Tdef} \times \text{Tenv} \times \text{Com}$, is defined by the rules in Fig. 4.6. A tail-call is well-typed, if the types of the arguments match the types declared at the definition. A case expression is more involved, first the expression must be typed with a data type, then each pattern must match a constructor, and lastly each branch is checked with the new variables bound to the types from the corresponding constructor. An if-then-else is well-typed, if the test has boolean type and both branches are well-typed. A let-expression updates the type environment with the type of the expression. A fail or done command is always well-typed.

An environment is well-typed, if it assigns correct values according to a type environment and a data type environment:

Definition 4.2.5. *An environment, ρ, is well-typed with respect to Θ and Γ, written $\Theta \vdash \rho : \Gamma$ iff*

$$\text{dom}(\Gamma) = \text{dom}(\rho) \wedge \forall x \in \text{dom}(\rho).\ \Theta, \Gamma \vdash \rho(x) : \Gamma(x).$$

$\boxed{\Theta, \Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Theta, \Gamma \vdash x : \tau}$$

$$\frac{\Theta(tn)(k) = (\tau_1, \ldots, \tau_n) \qquad \forall i.\ (\Theta, \Gamma \vdash e_i : \tau_i)}{\Theta, \Gamma \vdash k(e_1, \ldots, e_n) : tn}$$

$$\frac{}{\Theta, \Gamma \vdash \overline{i} : \texttt{int}} \qquad \frac{}{\Theta, \Gamma \vdash \overline{r} : \texttt{real}}$$

$$\frac{}{\Theta, \Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Theta, \Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\forall i.\ \Theta, \Gamma \vdash e_i : \tau_i \qquad \vdash o : \tau_1, \ldots, \tau_n \to \tau}{\Theta, \Gamma \vdash o(e_1, \ldots, e_n) : \tau}$$

Figure 4.5: Typing judgement for expressions. $i \in \{1, \ldots, n\}$.

$\boxed{\Psi, \Theta, \Gamma \vdash c : \texttt{com}}$

$$\frac{\Psi(f) = (\tau_1, \ldots, \tau_n) \qquad \forall i.\ \Theta, \Gamma \vdash e_i : \tau_i}{\Psi, \Theta, \Gamma \vdash f(e_1, \ldots, e_n) : \texttt{com}}$$

$$\frac{\Theta, \Gamma \vdash e : tn \qquad \forall i.\ \left\{ \begin{array}{c} p_i = k_i(x_i^1, \ldots, x_i^{m_i}) \\ \Theta(tn)(k_i) = (\tau_i^1, \ldots, \tau_i^{m_i}) \\ \Psi, \Theta, \Gamma[x_i^1 \mapsto \tau_i^1, \ldots, x_i^{m_i} \mapsto \tau_i^{m_i}] \vdash c_i : \texttt{com} \end{array} \right.}{\Psi, \Theta, \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \to c_1 \,|\, \cdots \,|\, p_n \to c_n : \texttt{com}}$$

$$\frac{\Theta, \Gamma \vdash e : \texttt{bool} \qquad \Psi, \Theta, \Gamma \vdash c_1 : \texttt{com} \qquad \Psi, \Theta, \Gamma \vdash c_2 : \texttt{com}}{\Psi, \Theta, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \texttt{com}}$$

$$\frac{\Theta, \Gamma \vdash e : \tau \qquad \Psi, \Theta, \Gamma[x \mapsto \tau] \vdash c : \texttt{com}}{\Psi, \Theta, \Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ c : \texttt{com}}$$

$$\frac{}{\Psi, \Theta, \Gamma \vdash \texttt{fail} : \texttt{com}} \qquad \frac{}{\Psi, \Theta, \Gamma \vdash \texttt{done} : \texttt{com}}$$

Figure 4.6: Typing judgement for commands. $i \in \{1, \ldots, n\}$.

Similarly, a list of declarations are well-typed if all functions are well-typed:

Definition 4.2.6. *A list of declarations, ds, is well-typed with respect to a data type environment, Θ, and a function type environment, Ψ, written $\Theta \vdash ds : \Psi$ iff*

$$\begin{aligned}&\forall(f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c) \in ds.\\ &\quad \Psi(f) = (\tau_1, \ldots, \tau_n) \wedge\\ &\quad \Psi, \Theta, [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash c : \texttt{com}.\end{aligned}$$

4.3 Implementation of verification-time monitoring

In this section we show how to implement the combined program in such a way that its error-free execution corresponds to conformance of a tactic. The general structure of the program is a single loop, for which an iteration corresponds to a step in the conformance relation of Def. 3.4.23. The full program is split into several components:

- The tactic component: simulates the transition functions of the concrete tactic. Basically Def. 3.4.14 implemented in the language. This component viewed in isolation denotes a tactic automaton, as shown later.
- The contract components: simulates the transition functions for concrete contracts. These components are implementations of the contract automata from Def. 3.4.7. Similar to the tactic component, each contract component in isolation denotes a contract automaton.
- The supervisor component: implements the test harness, i.e., this component performs the bookkeeping needed to implement the conformance relation. This includes deconstruction of the input trace, passing the incoming values to the tactics and contracts and collecting the accumulated pay-off. This component explicitly fails with `fail` if the accumulated pay-off becomes negative.

In the following sections we describe each component in detail. To exemplify the definitions we use Ex. 3.3.18 as a basis.

4.3.1 Tactic component

The tactic component comes with a special type definition of a type called *tactic state*. This type captures the state, S, for the tactic automaton. Furthermore, the component comes with two special functions, one for each transition map of a tactic automaton. The first function takes as input three things: an element of the tactic state type, an element representing the supervisor's state(explained later) and elements representing the actions in the input map. The second function takes as input an element of the tactic state type and an element representing the supervisor's state.

Simplified, the first function will have the following form:

$$\begin{aligned}&\text{tacI}(ss : \text{sState},\ ts : \text{tState}, l_1 : \text{act}_1, \ldots, l_n : \text{act}_n) =\\ &\qquad\vdots\\ &\quad \text{tRetI}(ss, ts', l'_1, \ldots, l'_m)\end{aligned}$$

where tRetI is assumed to have the following type: $(\text{sState}, \text{tState}, \text{act}'_1, \ldots, \text{act}'_m)$. In the function: ss is a variable containing the supervisor's state, which is passed unmodified

to the return function; ts, ts' is the tactic state in the last step and in the next step; $l_1, \ldots, l_n$ are the input actions and $l'_1, \ldots, l'_m$ are the output actions. The supervisor's state is considered abstract for the tactic component, but not for certification later. That the state is abstract means that the tactic component cannot decompose or construct a value of that type. This abstraction can be obtained by typing the tactic component without the definition of the concrete supervisor state. The different types of the actions, on the other hand, are not abstract: the tactic needs to be able to both decompose the input actions and construct the output actions, so we need to add the definition of the different action types, act_i, to the type declarations of the tactic. We do not fix exactly how actions look. but assume they include the empty action, giving a type declaration on the form:

$$\texttt{type}\ \text{act}_i = \text{Emp} \mid \ldots.$$

We write tds_{act} for these type declarations.

Similarly, the second function will have the following simplified form:

$$\begin{gathered}\text{tacT}(ss : \text{sState},\ ts : \text{tState}) = \\ \vdots \\ \text{tRetT}(ss, ts', l'_1, \ldots, l'_m)\end{gathered}$$

where tRetT has the same type as tRetI. Similar to the first function, ss and ts are the incoming states for the supervisor and the tactic; ts' is the new state for the tactic and $l'_1, \ldots, l'_m$ are the output actions.

In addition to the two functions, the tactic component must also include a value of type tState for the start state, and a real value for the time between timer transitions. We consider a tactic component example:

Example 4.3.1. Assuming an action definition of:

```
type act = Emp | V : int
```

The +2 tactic from Ex. 3.3.18 and Ex. 3.4.15 can be implemented in the following way:

```
type tState = S1 | S2 | S3

tacI(ss : sState, ts : tState, lo1 : act, li2 : act =
  case ts of
    S1 → case li2 of
      Emp → tRetI(ss,S1,Emp,Emp)
    | V(n) → tRetI(ss,S2,Emp,V(n))
  | S2 → case lo1 of
      Emp → tRetI(ss,S2,Emp,Emp)
    | V(n) → tRetI(ss,S3,Emp,V(n))
  | S3 → case lo1 of
      Emp → tRetI(ss,S3,Emp,Emp)
    | V(n) → tRetI(ss,S1,V(n),Emp)

tacT(ss : sState, ts : tState) = tRetT(ss,ts,Emp, Emp)
```

with `S1` as start state and 1.0 as timer transition time.

In order for the tactic component to denote a tactic automaton, we need to restrict what can be written in the two functions. The requirements come in two different categories: basic well-formedness requirements and semantic requirements. The basic requirements are:

1. The program must be well-typed.
2. Two functions, tacI, tacT, and a special type, tState, must be defined.
3. There must be a start state and a timer transition time.

The semantic requirements are the following:

1. The functions always terminates with a call to tRetI or tRetT respectively.
2. The argument *ss* is passed through the functions unmodified.

The precise formalisation of the basic requirements is given below.

Definition 4.3.2. *A list of declarations, ds, with typing environments, Θ, Ψ, and values $\nu_0 \in \mathcal{V}$, $r_0 \in \mathbb{R}$, is a well-formed tactic component for a tactic with input links $l_1, \ldots, l_n$ and output links $l'_1, \ldots, l'_m$, written $\vdash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{tac}$, iff it satisfies the following conditions:*

1. $\Theta' \vdash ds : \Psi'$.
2. $\exists k, \tau_1, \ldots, \tau_n.\ \Theta(\text{tState})(k) = (\tau_1, \ldots, \tau_n)$.
3. $\Theta', \emptyset \vdash \nu_0 : \text{tState}$.
4. $\exists c_{\text{ti}}.\ \text{tacI}(ss : \text{sState},\ ts : \text{tState}, l_1 : \text{act}, \ldots, l_n : \text{act}) = c_{\text{ti}} \in ds$.
5. $\exists c_{\text{tt}}.\ \text{tacT}(ss : \text{sState},\ ts : \text{tState}) = c_{\text{tt}} \in ds$.

where

$$\Psi' = \Psi[\text{tRetI} \mapsto (\text{sState}, \text{tState}, \text{act}_1, \ldots, \text{act}_n), \text{tRetT} \mapsto (\text{sState}, \text{tState}, \text{act}_1, \ldots, \text{act}_n)]$$

and Θ' is Θ updated with mappings for the actions in tds_{act}.

The first requirement is that the entire program is well-typed, when the types for the return functions are added. The second requirement specifies that the type for tactic states, tState, is defined. The third requirement expresses that the initial state is of type tState. The fourth and fifth requirements ensure that the two transition functions are well-defined. The semantic requirements are defined below:

Definition 4.3.3. *A well-formed tactic component, $\vdash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{tac}$, for a tactic with input links $l_1, \ldots, l_n$ and output links $l'_1, \ldots, l'_m$, is a well-behaved tactic component, written $\vDash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{wbTac}$, iff it satisfies the following two conditions:*

$$\begin{aligned}
&\forall \rho, \forall M \in \text{Tag} \rightharpoonup_{\text{fin}} \text{Type}^*.\\
&\quad \Theta'[\text{sState} \mapsto M] \vdash \rho : [ss \mapsto \text{sState}, ts \mapsto \text{tState}, l_1 \mapsto \text{act}_1, \ldots, l_n \mapsto \text{act}_n] \Rightarrow\\
&\quad \big(ds \vdash \langle c_{\text{ti}}, \rho\rangle \rightarrow^* \langle \text{tRetI}(e_{ss}, e_{ts}, e_{l'_1}, \ldots, e_{l'_m}), \rho'\rangle \wedge \rho' \vdash e_{ss} \downarrow \rho(ss)\big);
\end{aligned}$$

and

$$\begin{array}{l}\forall \rho, \forall M \in \mathrm{Tag} \rightharpoonup_{\mathrm{fin}} \mathrm{Type}^*.\\ \quad \Theta'[\mathrm{sState} \mapsto M] \vdash \rho : [ss \mapsto \mathrm{sState}, ts \mapsto \mathrm{tState}] \Rightarrow\\ \quad \big(ds \vdash \langle c_{\mathrm{tt}}, \rho\rangle \rightarrow^* \langle \mathrm{tRetT}(e_{ss}, e_{ts}, e_{l'_1}, \ldots, e_{l'_m}), \rho'\rangle \wedge \rho' \vdash e_{ss} \downarrow \rho(ss)\big),\end{array}$$

and Θ' is Θ updated with mappings for the actions in tds_{act}.

Well-behavedness ensures that when either of the two transition functions are called, then control steps to the corresponding return function and the first argument of the transition function is unchanged. Even though we only give a semantic definition, it should be easy to give a sound syntactic judgement ($\vdash \ldots :$ `wbTac`), ensuring that a tactic component is well-behaved. One way would be to forbid any operations using the first parameter, and to forbid function calls backwards in the program. When we have a well-behaved tactic component, we can translate it into an tactic automaton in the following way:

Definition 4.3.4. *Given a well-behaved tactic component $\vDash ds, \Theta, \Psi, \nu_0, r_0 :$ `wbTac`, we define a tactic automaton $(S, s_0, d_0, \delta_{\mathrm{t}}, \delta_{\mathrm{i}})$ in the following way:*

$$\begin{array}{rl} S = \{\nu \in \mathcal{V} \mid \Theta', \emptyset \vdash \nu : \mathrm{tState}\},\\ s_0 = \nu_0,\\ d_0 = r_0,\\ \delta_{\mathrm{t}}(s) = (s', m') \textit{ when}\\ ds \vdash \langle \mathrm{tacT}(\nu_{\mathrm{ss}}, s), \emptyset\rangle \rightarrow^* \langle \mathrm{tRetT}(e_{\mathrm{ss}}, e_{ps}, e_{l'_1}, \ldots, e_{l'_m}), \rho'\rangle,\\ \qquad \rho' \vdash e_{ps} \downarrow s',\\ \qquad \rho' \vdash e_{l'_1} \downarrow \nu'_1 \quad \cdots \quad \rho' \vdash e_{l'_m} \downarrow \nu'_m,\\ \qquad m' = [l'_1 \mapsto \nu'_1, \ldots, l'_m \mapsto \nu'_m];\end{array}$$

$$\begin{array}{rl} \delta_{\mathrm{i}}(s, m) = (s', m') \textit{ when}\\ \qquad m = [l_1 \mapsto \nu_1, \ldots, l_n \mapsto \nu_n],\\ ds \vdash \langle \mathrm{tacI}(\nu_{\mathrm{ss}}, s, \nu_1, \ldots, \nu_n), \emptyset\rangle \rightarrow^* \langle \mathrm{tRetI}(e_{\mathrm{ss}}, e_{ps}, e_{l'_1}, \ldots, e_{l'_m}), \rho'\rangle,\\ \qquad \rho' \vdash e_{ps} \downarrow s',\\ \qquad \rho' \vdash e_{l'_1} \downarrow \nu'_1 \quad \cdots \quad \rho' \vdash e_{l'_m} \downarrow \nu'_m,\\ \qquad m' = [l'_1 \mapsto \nu'_1, \ldots, l'_m \mapsto \nu'_m];\end{array}$$

where Θ' is Θ updated with mappings for the actions in tds_{act}, and ν_{ss} is an arbitrary value. We write $\ulcorner(ds, \Theta, \Psi, \nu_0, r_0)\urcorner$ for $(S, s_0, d_0, \delta_{\mathrm{t}}, \delta_{\mathrm{i}})$.

The idea behind the definition is to use values of type tState as the states in the automaton tactic. The transition functions of the automaton tactic are implemented with the two special functions, and to get the outputs we use the values that are passed to the return functions. We can show that this gives a tactic automaton:

Proposition 4.3.5. *If $\vDash ds, \Theta, \Psi, \nu_0, r_0 :$ `wbTac` is a well-behaved tactic component, then the translated tactic, $\ulcorner(ds, \Theta, \Psi, \nu_0, r_0)\urcorner$, (Def. 4.3.4) is a tactic automaton.*

Proof sketch. Because the component is well-formed, the types of the initial state and the transition functions are correct. What remains is to show that the transition functions are well-defined. From well-behavedness we know that the derivation exists, and because the language is deterministic, the result is well-defined. The last part is to make sure that execution cannot depend on the value v_{ss}. Because of the well-formedness typing, the value cannot be used, and therefore cannot affect the execution as needed. □

4.3.2 Contract components

The contract components are similar to the tactic components, the difference is the type of the two functions, as they have to correspond to the transition functions for contract automata.

Similar to the tactic component, each contract component defines a special type called the *contract state*. The type captures the contract automaton state, G. The first of the two functions will have the following simplified form:

$$\begin{aligned}\text{conI}(ss : \text{sState},\ cs : \text{cState}, l_1 : \text{act}_1, \ldots, l_n : \text{act}_n) =\\ \vdots\\ \text{cRetI}(ss, res)\end{aligned}$$

where cRetI has the following type: $(\text{sState}, \text{run})$. The result type, run, captures the disjoint sum in the transition function and is defined to be:

$$\texttt{type}\ \text{run} = \text{Run} : \text{cState}\ \texttt{|}\ \text{Done} : \texttt{real}$$

We write td_{run} for this type declaration. The variables used in the function are similar to the ones for the tactic component, ss is a variable containing the supervisor's state, which is again passed unmodified to the next function; cs is the contract state in last step and $l_1, \ldots, l_n$ is the input actions. It is important to note that the supervisor's state passed to the contract component can be different from the one passed to the tactic component, e.g. the contract one will contain the state of the tactic, and the tactic one the state of the contracts.

The second function is very similar, it has the following simplified form:

$$\begin{aligned}\text{conT}(ss : \text{sState},\ cs : \text{cState}) =\\ \vdots\\ \text{cRetT}(ss, res)\end{aligned}$$

where the only difference is the missing input actions.

We extend the example from before:

Example 4.3.6. The +2 contract, c_2, from Ex. 3.3.18 can be formalised very similarly to the c_1 contract from Ex. 3.4.8, and it can be implemented in the following way:

```
type cState2 = C2start | C2run : int, real

con2I(ss : sCon2, cs2 : cState2, li2 : act, lo2 : act) =
case cs2 of
  C2start →
```

```
    if lo2 ≠ Emp then con2RetI(ss,Done2(-1))
    else case li2 of
           Emp → con2RetI(ss,Run2(cs2))
         | V(n) → con2RetI(ss,Run2(C2run(n,20)))
| C2run (n,t) →
  if li2 ≠ Emp then con2RetI(ss,Done2(1))
  else case lo2 of
         Emp → con2RetI(ss,Run2(cs2))
       | V(n2) → if n + 2 = n2 then con2RetI(ss,Run2(C2start))
                 else con2RetI(ss,Done2(1))

con2T(ss : sCon2,cs2 : cState2) =
case cs2 of
  C2start → con2RetT(ss,Run2(cs2))
| C2run (n,t) →
    if t ≤ 0.0
    then con2RetT(ss,Done2(-1))
    else con2RetT(ss,Run2(C2run(n,t - 1)))
```

with `C1` as a start state and 1.0 as timer transition time.

Similar to the tactic component, in order for the contract components to denote contract automata, certain properties need to be fulfilled. Again there are two categories: basic requirements and semantic requirements.

Definition 4.3.7. *A list of declarations, ds, with typing environments, Θ, Ψ, and values $\nu_0 \in \mathcal{V}, r_0 \in \mathbb{R}$, is a well-formed contract component for a contract with links $l_1, \ldots, l_n$, written $\vdash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{contr}$, iff it satisfies the following conditions:*

1. $\Theta' \vdash ds : \Psi'$.
2. $\exists k, \tau_1, \ldots, \tau_n.\ (k, (\tau_1, \ldots, \tau_n)) \in \Theta(\text{cState})$.
3. $\Theta', \emptyset \vdash \nu_0 : \text{cState}$.
4. $\exists c_{\text{ci}}.\ \text{conI}(ss : \text{sState},\ cs : \text{cState}, l_1 : \text{act}, \ldots, l_n : \text{act}) = c_{\text{ci}} \in ds$.
5. $\exists c_{\text{ct}}.\ \text{conT}(ss : \text{sState},\ cs : \text{cState}) = c_{\text{ct}} \in ds$.

where

$$\Psi' = \Psi[\text{cRetI} \mapsto (\text{sState}, \text{run}), \text{cRetT} \mapsto (\text{sState}, \text{run})]$$

and Θ' is Θ updated with mappings for tds_{act} and td_{run}.

These syntactic requirements are very similar to the ones for tactics, the biggest difference is the return type of the transition functions. The semantic requirements are similar as well:

Definition 4.3.8. *A well-formed contract component, $\vdash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{contr}$, for a contract with links $l_1, \ldots, l_n$, is called a well-behaved contract component, written $\vDash$*

$ds, \Theta, \Psi, \nu_0, r_0 : \texttt{wbContr}$, iff it satisfies the following two conditions:

$$\begin{aligned}
&\forall \rho, \forall M \in \mathrm{Tag} \rightharpoonup_{\mathrm{fin}} \mathrm{Type}^*.\\
&\quad \Theta'[\mathrm{sState} \mapsto M] \vdash \rho : [ss \mapsto \mathrm{sState}, cs \mapsto \mathrm{cState}, l_1 \mapsto \mathrm{act}_1, \ldots, l_n \mapsto \mathrm{act}_n] \Rightarrow\\
&\quad \big(ds \vdash \langle c_{\mathrm{ci}}, \rho\rangle \rightarrow^* \langle \mathrm{cRetI}(e_{ss}, e_{res}), \rho'\rangle \wedge \rho' \vdash e_{ss} \downarrow \rho(ss)\big);
\end{aligned}$$

and

$$\begin{aligned}
&\forall \rho, \forall M \in \mathrm{Tag} \rightharpoonup_{\mathrm{fin}} \mathrm{Type}^*.\\
&\quad \Theta'[\mathrm{sState} \mapsto M] \vdash \rho : [ss \mapsto \mathrm{sState}, cs \mapsto \mathrm{cState}] \Rightarrow\\
&\quad \big(ds \vdash \langle c_{\mathrm{ct}}, \rho\rangle \rightarrow^* \langle \mathrm{cRetT}(e_{ss}, e_{res}), \rho'\rangle \wedge \rho' \vdash e_{ss} \downarrow \rho(ss)\big),
\end{aligned}$$

As for tactic components, it should be easy to give a sound syntactic judgement ($\vdash \ldots : \texttt{wbContr}$), ensuring that a contract components is well-behaved. With a well-behaved contract component, we can translate it into a contract automaton in the following way:

Definition 4.3.9. *Given a legal contract component $\vDash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{wbContr}$, we define a contract automaton $(G, g_0, d_0, \rho_{\mathrm{t}}, \rho_{\mathrm{i}})$ in the following way:*

$$\begin{aligned}
G &= \{\nu \in \mathcal{V} \mid \Theta', \emptyset \vdash \nu : \mathrm{cState}\},\\
g_0 &= \nu_0,\\
d_0 &= r_0,\\
\rho_{\mathrm{t}}(g) &= \begin{cases} g' & \text{if } \rho' \vdash e_{res} \downarrow \mathrm{Run}(g'),\\ k & \text{if } \rho' \vdash e_{res} \downarrow \mathrm{Done}(k), \end{cases}\\
&\text{when}\\
&ds \vdash \langle \mathrm{conT}(\nu_{\mathrm{ss}}, s), \emptyset\rangle \rightarrow^* \langle \mathrm{cRetT}(e_{\mathrm{ss}}, e_{res}), \rho'\rangle;\\
\rho_{\mathrm{i}}(g, m) &= \begin{cases} g' & \text{if } \rho' \vdash e_{res} \downarrow \mathrm{Run}(g'),\\ k & \text{if } \rho' \vdash e_{res} \downarrow \mathrm{Done}(k), \end{cases}\\
&\text{when}\\
&\qquad m = [l_1 \mapsto \nu_1, \ldots, l_n \mapsto \nu_n],\\
&ds \vdash \langle \mathrm{conI}(\nu_{\mathrm{ss}}, g, \nu_1, \ldots, \nu_n), \emptyset\rangle \rightarrow^* \langle \mathrm{cRetI}(e_{\mathrm{ss}}, e_{res}), \rho'\rangle;
\end{aligned}$$

where Θ' is Θ updated with mappings for tds_{act}, td_{run} and ν_{ss} is an arbitrary value. We write $\ulcorner(ds, \Theta, \Psi, \nu_0, r_0)\urcorner$ for $(G, g_0, d_0, \rho_{\mathrm{t}}, \rho_{\mathrm{i}})$.

The definition is again similar to the one for tactics. But instead of constructing an output map as output, either a new state or a pay-off must be passed to the return functions. We can now show that this gives a contract automaton:

Proposition 4.3.10. *If $\vDash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{wbContr}$ is a legal contract fragment, then the translated contract, $\ulcorner(ds, \Theta, \Psi, \nu_0, r_0)\urcorner$, (Def. 4.3.9) is a legal contract automaton.*

Proof sketch. Analogous to the proof of Prop. 4.3.5. □

4.3.3 Supervisor component

The supervisor component is the last part of the composition. It maintains the input trace, the accumulated pay-off and the state of both the tactic and the contracts. The

overall structure of the supervisor component is a single big loop, where each iteration of the loop corresponds to a step in the conformance relation from Def. 3.4.23. The supervisor component will have the following (very) simplified form:

$$
\begin{aligned}
&\text{start}(t_{\text{start}} : \texttt{real}, t_{\text{end}} : \texttt{real}, tr_{\text{in}} : \text{trace}_{\text{in}}) = \\
&\qquad \text{[Call loop with initial values]} \\
&\\
&\text{loop}(t_{\text{now}} : \texttt{real}, ts : \text{tState}, t_{\text{tac}} : \texttt{real}, \\
&\qquad rs_1 : \text{run}_1, t_{\text{c1}} : \texttt{real}, \ldots, rs_n : \text{run}_n, t_{\text{cn}} : \texttt{real}, \\
&\qquad t\text{end} : \texttt{real}, tr_{\text{in}} : \text{trace}_{\text{in}}, tr_{\text{out}} : \text{trace}_{\text{out}}) = \\
&\qquad\qquad \vdots \\
&\qquad \text{[Split traces]} \\
&\qquad\qquad \vdots \\
&\qquad \text{[Branch to tactic]} \\
&\qquad\qquad \vdots \\
&\qquad \text{[Branch to contract 1]} \\
&\qquad\qquad \vdots \\
&\qquad \text{[Branch to contract k]} \\
&\qquad\qquad \vdots \\
&\qquad \text{[Fail if payoff is negative]} \\
&\qquad\qquad \vdots \\
&\qquad \text{[Find time for next step]} \\
&\qquad\qquad \vdots \\
&\qquad \text{loop}(t'_{\text{now}}, ts', t'_{\text{tac}}, rs'_1, t'_{\text{c1}}, \ldots, rs'_n, t'_{\text{cn}}, t'_{\text{end}}, tr'_{\text{in}}, tr'_{\text{out}})
\end{aligned}
$$

In general, the variable in the loop function stems from the conformance relation. The running state of tactics and contracts is captured with both a state variable (ts, rs_i) and a variable holding the time for next timer transition ($t_{\text{tac}}, t_{\text{ci}}$). To make the concrete certification easier, we have split the trace in two parts; one part for the input to the supervisor (the test script) and another part for the generated output of the tactic. In this section we do not directly specify how the traces should be formalised, one could e.g. use a list for each link:

$$\texttt{type}\ \text{trace} = \text{Nil}\ \texttt{|}\ \text{Cons} : \texttt{real}, \text{act}, \text{trace}.$$

In the following, we describe each part of the supervisor. However, we do not formally specify each single part of a supervisor component, because basically it is just an implementation of the conformance relation in a coroutine language. There might also be several ways of implementing it, or automatically generating it. We consider a full implementation of a concrete supervisor in Sect. 4.5.

Split traces. In this part, the supervisor extracts all actions from the two traces that happen at the time specified by t_{now}. The actions must then be removed from

the trace so that they are not considered again in the next iteration. If we use the list representation described above with a single link, l, the split operation could be implemented in the following fashion:

$$
\begin{array}{l}
\texttt{case } tr \texttt{ of} \\
\quad \text{Nil} \rightarrow \texttt{let } l = \text{Emp } \texttt{in let } tr' = tr \texttt{ in } \ldots \\
\mid \text{Cons}(t, a, rest) \rightarrow \\
\qquad \texttt{if } t = t_{\text{now}} \\
\qquad \texttt{then let } l = a \texttt{ in let } tr' = rest \texttt{ in } \ldots \\
\qquad \texttt{else let } l = \text{Emp } \texttt{in let } tr' = tr \texttt{ in } \ldots
\end{array}
$$

In each path the potential action is kept in the variable l and the remaining trace in tr'.

Branch to tactic. In this part the supervisor implements the tstep operation from Def. 3.4.18. This means that the supervisor must compare the time for the timer transition with the current time, and also consider whether there is any input for the tactic. When the tactic returns, the output must be added to the output trace, so that it can be considered in a later iteration. A structure like the following can be used:

$$
\begin{array}{l}
\texttt{if } t_{\text{now}} < t_{\text{tac}} \\
\texttt{then if } l_1 = \text{Emp} \wedge \ldots \wedge l_n = \text{Emp} \\
\qquad \texttt{then let } ts' = ts \texttt{ in let } t'_{\text{tac}} = t_{\text{tac}} \texttt{ in let } tr'_{\text{out}} = tr_{\text{out}} \texttt{ in } \ldots \\
\qquad \texttt{else } \text{tacI}(ss, ts, l_1, \ldots, l_n) \\
\texttt{else } \text{tacT}(ss', ts)
\end{array}
$$

Here the supervisor state variables (ss, ss') capture all the live variables so that they can be used when the tactic returns to the supervisor. This includes all actions, traces and the state of the contracts. When the timer transition returns, the output is added to the trace, and, depending on whether there are any input, calls the input transition function.

$$
\begin{array}{l}
\text{tRetT}(ss, ts, l_1, \ldots, l_m) = \\
\quad \texttt{let } tr''_{\text{out}} = [\text{add } l_1, \ldots, l_m \text{ to } tr_{\text{out}'}] \texttt{ in} \\
\quad \texttt{let } ts' = ts \texttt{ in} \\
\quad \texttt{let } t'_{\text{tac}} = t_{\text{tac}} + [\text{timeout for the tactic}] \texttt{ in} \\
\quad [\text{call tacI if there is input}]
\end{array}
$$

$$
\begin{array}{l}
\text{tRetI}(ss, ts, l_1, \ldots, l_m) = \\
\quad \texttt{let } tr''_{\text{out}} = [\text{add } l_1, \ldots, l_m \text{ to } tr_{\text{out}'}] \texttt{ in} \\
\quad \texttt{let } ts' = ts \texttt{ in} \\
\quad \texttt{let } t'_{\text{tac}} = t_{\text{tac}} \texttt{ in} \ldots
\end{array}
$$

The method used for adding the actions to the trace depends on the representation used for the trace, and must take the latency of each link into account.

Branch to contract. In this part, the supervisor implements the cstep operation from Def. 3.4.10. This part is similar to the tactic, except the trace does not have to be updated, which makes it simpler.

Fail if pay-off is negative. This part sums up the pay-off of all contracts that are done. If the sum is less than zero, the supervisor fails with the `fail` command.

$$
\begin{array}{l}
\texttt{let } k_{\text{acc}} = 0 \texttt{ in} \\
\vdots \\
\texttt{case } rs \texttt{ of} \\
\quad \text{Done}(k) \rightarrow \texttt{let } k_{\text{acc}} = k_{\text{acc}} + k \texttt{ in} \\
\qquad \texttt{if } k_{\text{acc}} < 0 \texttt{ then fail else } \ldots \\
\mid \text{Run}(cs) \rightarrow \\
\qquad \texttt{if } k_{\text{acc}} < 0 \texttt{ then fail else } \ldots
\end{array}
$$

Find time for next step. In the last part, the supervisor finds the minimum time amongst the timer transitions times for the tactic, the contracts and the next actions in the traces. The concrete implementation depends on the representation of the traces; if the list representation is used an implementation similar to the following can be used:

$$
\begin{array}{l}
\texttt{case } tr \texttt{ of} \\
\quad \text{Nil} \rightarrow \text{loop}(t'_{\text{tac}}, ts', t'_{\text{tac}}, gs'_1, \ldots, gs'_n, t'_{\text{end}}, tr'_{\text{in}}, tr'_{\text{out}}) \\
\mid \text{Cons}(t, a, rest) \rightarrow \\
\qquad \texttt{if } t < t'_{\text{tac}} \\
\qquad \texttt{then loop}(t, ts', t'_{\text{tac}}, gs'_1, \ldots, gs'_n, t'_{\text{end}}, tr'_{\text{in}}, tr'_{\text{out}}) \\
\qquad \texttt{else loop}(t'_{\text{tac}}, ts', t'_{\text{tac}}, gs'_1, \ldots, gs'_n, t'_{\text{end}}, tr'_{\text{in}}, tr'_{\text{out}})
\end{array}
$$

where the difference between the different calls to loop is the value of the first parameter.

4.3.4 Implementation correctness

When a specific implementation of the supervisor has been created, one has to prove that it implements the conformance relation faithfully, so that it can be used as a basis for certifying the tactics. As we have not given a formal definition of the supervisor, we can only state the needed theorem; it is up to the implementer to prove it.

Obligation 4.3.11. *Given a well-behaved tactic component,* $\vDash ds, \Theta, \Psi, \nu_0, r_0 : \texttt{wbTac}$*, and well-behaved contract components:*

$$\vDash ds_1, \Theta_1, \Psi_1, \nu_0^1, r_0^1 : \texttt{wbContr}, \ldots, \vDash ds_n, \Theta_n, \Psi_n, \nu_0^n, r_0^n : \texttt{wbContr}.$$

A supervisor implementation, ds_{super} *is considered a correct implementation of the tactics and the contracts if it satisfies that*

$$\ulcorner (ds, \Theta, \Psi, \nu_0, r_0) \urcorner$$

conforms with (Def. 3.4.23) at time t_0

$$\ulcorner (ds_1, \Theta_1, \Psi_1, \nu_0^1, r_0^1) \urcorner, \ldots, \ulcorner (ds_n, \Theta_n, \Psi_n, \nu_0^n, r_0^n) \urcorner.$$

$$
\begin{array}{rll}
\text{Assert} \ni A ::= & \mid e & \text{(Expressions)} \\
& \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 & \text{(Logical connectives)} \\
& \mid \forall x : \tau.\ A \mid \exists x : \tau.\ A & \text{(Quantifiers)}
\end{array}
$$

Figure 4.7: The syntax of assertions.

if and only if $\forall t, \sigma$

$$ds_{\text{super}} \vdash \langle \text{start}(t_0, t, \ulcorner \sigma \urcorner), \emptyset \rangle \rightarrow^* \langle c, \rho \rangle \Rightarrow c \neq \texttt{fail}.$$

We use $\ulcorner \sigma \urcorner$ *as the representation of the given trace.*

4.4 Floyd-Hoare logic

For conformance we wish to prove that the supervisor program does not fail with an error. To certify concrete programs, we need a syntactic representation of such a proof of error-free execution. We therefore introduce a Floyd-Hoare logic to express such proofs. The logic will be given as a set of verification conditions extracted from an annotated program. If all these verification conditions hold, then the program will never fail. We could have used another approach to error-free execution, but have picked Floyd-Hoare logic, because it is well-known and easy to adopt for our language.

The first step to make a Floyd-Hoare logic is to define assertions, which can express properties about the environment of the program. Assertions are expressions of boolean type, but in addition we need to add quantifiers. To prevent errors in the assertions, e.g. an assertion containing the sum of a number and a data type, we use a typing system for the assertions. We will briefly elaborate on this in Sect. 4.4.1.

The syntax of assertions is given in Fig. 4.7. The only thing to note is that the quantifiers are equipped with a type to more easily relate them to the type environment.

The typing judgement for assertions, $\cdot, \cdot \vdash \cdot : \texttt{assert} \subseteq \text{Tdef} \times \text{Tenv} \times \text{Assert}$, is straightforward, using the type judgement for expressions. The rules are shown in Fig. 4.8.

For well-typed assertions and environments, we can define when an assertion holds in an environment:

Definition 4.4.1. *Given a well-typed assertion,* $\Theta, \Gamma \vdash A : \texttt{assert}$*, and a well-typed environment* $\Theta \vdash \rho : \Gamma$*. The satisfies relation* $\cdot \vDash \cdot \subseteq \text{Env} \times \text{Assert}$*, is defined by structural induction on* A *in the following way:*

$$
\begin{array}{rl}
\rho \vDash e & \textit{iff } \rho \vdash e \downarrow \texttt{true}, \\
\rho \vDash \neg A & \textit{iff } \rho \nvDash A, \\
\rho \vDash A_1 \wedge A_2 & \textit{iff } \rho \vDash A_1 \wedge \rho \vDash A_2, \\
\rho \vDash A_1 \vee A_2 & \textit{iff } \rho \vDash A_1 \vee \rho \vDash A_2, \\
\rho \vDash \forall x : \tau.\ A & \textit{iff } \forall \nu \in \mathcal{V}.\ \Theta, \Gamma \vdash \nu : \tau \Rightarrow \rho[x \mapsto \nu] \vDash A, \\
\rho \vDash \exists x : \tau.\ A & \textit{iff } \exists \nu \in \mathcal{V}.\ \Theta, \Gamma \vdash \nu : \tau \wedge \rho[x \mapsto \nu] \vDash A.
\end{array}
$$

$\boxed{\Theta, \Gamma \vdash A : \texttt{assert}}$

$$\frac{\Theta, \Gamma \vdash e : \texttt{bool}}{\Theta, \Gamma \vdash e : \texttt{assert}}$$

$$\frac{\Theta, \Gamma \vdash A : \texttt{assert}}{\Theta, \Gamma \vdash \neg A : \texttt{assert}}$$

$$\frac{\Theta, \Gamma \vdash A_1 : \texttt{assert} \qquad \Theta, \Gamma \vdash A_2 : \texttt{assert}}{\Theta, \Gamma \vdash A_1 \wedge A_2 : \texttt{assert}}$$

$$\frac{\Theta, \Gamma \vdash A_1 : \texttt{assert} \qquad \Theta, \Gamma \vdash A_2 : \texttt{assert}}{\Theta, \Gamma \vdash A_1 \vee A_2 : \texttt{assert}}$$

$$\frac{\Theta, \Gamma[x \mapsto \tau] \vdash A : \texttt{assert}}{\Theta, \Gamma \vdash \forall x : \tau.\ A : \texttt{assert}} \qquad \frac{\Theta, \Gamma[x \mapsto \tau] \vdash A : \texttt{assert}}{\Theta, \Gamma \vdash \exists x : \tau.\ A : \texttt{assert}}$$

Figure 4.8: The typing judgement for assertions.

We only need closed assertions for the verification conditions later, so given a closed assertion $\Theta, \emptyset \vdash A : \texttt{assert}$, we can define validity of an assertion $\vDash A$ as $\emptyset \vDash A$.

To define verification conditions, we annotate each function with an assertion; every call to that function must satisfy that assertion.

Definition 4.4.2. *An annotation environment is a mapping:*

$$\varphi \in \text{Aenv} := \text{Fvar} \rightharpoonup_{\text{fin}} \text{Assert}.$$

An annotation environment is well-typed for a list of declarations, ds, and a type definition environment, Θ, written $ds, \Theta \vdash \varphi : \texttt{aenv}$ iff:

$$\begin{aligned} \forall f \in \text{dom}(\varphi).\ (f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c) \in ds \wedge \\ \Theta, [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash \varphi(f) : \texttt{assert} \end{aligned}$$

Given a well-typed annotation environment, the verification conditions for a command can be defined. To simplify the conditions we assume that all bound variables (function parameters, let and case bound variables) in a command are distinct. This can be accomplished by renaming. With this assumption we specify the verification conditions:

Definition 4.4.3. *Given a well-typed annotation environment $ds, \Theta \vdash \varphi : \texttt{aenv}$. When $\Theta, \Gamma \vdash A : \texttt{assert}$ and $ds, \Theta, \Gamma \vdash c : \texttt{com}$ we can define the verification conditions as a function* $\text{vc} : \text{Tenv} \times \text{Assert} \times \text{Com} \to \mathcal{P}_{\text{fin}}(\text{Assert})$. *The function satisfies that each assertion, A', in the verification conditions is closed ($\Theta, \emptyset \vdash A' : \texttt{assert}$), and it is*

defined in the following way:

$$\begin{aligned}
&\mathrm{vc}(\Gamma, A, f(e_1, \ldots, e_n)) = \{\forall x'_1 : \tau'_1, \ldots, x'_m : \tau'_m.\ A \Rightarrow \varphi(f)[e_1/x_1, \ldots, e_n/x_n]\} \\
&\quad \textit{where } \Gamma = [x'_1 \mapsto \tau'_1, \ldots, x'_m \mapsto \tau'_m] \\
&\quad \textit{and } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c \in ds; \\
&\mathrm{vc}(\Gamma, A, \texttt{case } e \texttt{ of } p_1 \to c_1 \texttt{ / } \ldots \texttt{ / } p_n \to c_n) = \mathrm{vc}(\Gamma_1, A_1, c_1) \cup \ldots \cup \mathrm{vc}(\Gamma_n, A_n, c_n) \\
&\quad \textit{where } p_i = k_i(x_1, \ldots, x_m), \\
&\qquad \Theta(tn)(k_i) = (\tau_1, \ldots, \tau_m), \\
&\qquad \Gamma_i = \Gamma[x_1 \mapsto \tau_1, \ldots, x_m \mapsto \tau_m], \\
&\qquad A_i \equiv k_i(x_1, \ldots, x_m) = e \wedge A; \\
&\mathrm{vc}(\Gamma, A, \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) = \mathrm{vc}(\Gamma, A \wedge e, c_1) \cup \mathrm{vc}(\Gamma, A \wedge \neg e, c_2); \\
&\mathrm{vc}(\Gamma, A, \texttt{let } x = e \texttt{ in } c) = \mathrm{vc}(\Gamma[x \mapsto \tau], x = e \wedge A, c) \\
&\quad \textit{when } \Theta, \Gamma \vdash e : \tau \\
&\mathrm{vc}(\Gamma, A, \texttt{fail}) = \{\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ A \Rightarrow \texttt{false}\}; \\
&\quad \textit{where } \Gamma = [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \\
&\mathrm{vc}(\Gamma, A, \texttt{done}) = \emptyset;
\end{aligned}$$

substitutions happen simultaneous. Furthermore, we assume that all nested variables are distinct, as mentioned above.

As we see, most of the rules are standard. Note that because we assume the variables to be distinct, we can use simple rules for let and case. If the bound variables could occur in the store, we would need to use an existential quantifier to capture the old value, e.g. the right hand side of the let clause would have the following form:

$$\mathrm{vc}(\Gamma[x \mapsto \tau], \exists x' : \Gamma(x).\ x = e[x'/x] \wedge A[x'/x], c).$$

The verification conditions are sound, meaning that if all verification conditions are valid, then the program does not step to an error. Before proving soundness, we need a couple of standard lemmas. The first lemma relates substitution to environment update:

Lemma 4.4.4. *If*

$$\rho \vdash e_1 \downarrow \nu_1, \ldots, \rho \vdash e_n \downarrow \nu_n$$

then

$$(\rho[x_1 \mapsto \nu_1, \ldots, x_n \mapsto \nu_n] \vDash A) \Leftrightarrow (\rho \vDash A[e_1/x_1, \ldots, e_n/x_n])$$

Proof. Straightforward induction, using a similar result for expression evaluation. □

The second lemma shows how to remove the universally quantified variables:

Lemma 4.4.5. *If* $\Gamma = [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]$

$$\vDash \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ A$$

then for any ρ*, satisfying* $\Theta \vdash \rho : \Gamma$*, we have:*

$$\rho \vDash A.$$

Proof. Follows directly from the definitions. □

The soundness of the verification conditions is proven by the following theorem:

Theorem 4.4.6. *Given a well-typed list of declarations, $\Theta \vdash ds : \Psi$, and given a well-typed annotation environment $ds, \Theta \vdash \varphi : \texttt{aenv}$. If all annotations are valid:*

$$\begin{aligned}&\forall f \in \mathrm{dom}(\varphi).\ (f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c) \in ds \ \wedge \\ &\quad (\forall A \in \mathrm{vc}([x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n], \varphi(f), c).\ \vDash A)\end{aligned}$$

then for each $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = c \in ds$ and every well-typed environment $\Theta \vdash \rho : [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]$ we have that:

$$\big(\rho \vDash \varphi(f) \wedge ds \vdash \langle c, \rho\rangle \to^* \langle c', \rho'\rangle\big) \Rightarrow c' \neq \texttt{fail}$$

Proof. We show the following which implies the result: For $\Theta \vdash \rho : \Gamma$, if

$$\rho \vDash A \wedge ds \vdash \langle c, \rho\rangle \to^* \langle c', \rho'\rangle \wedge \forall A' \in \mathrm{vc}(\Gamma, A, c).\ \vDash A'$$

then $c' \neq \texttt{fail}$. We show the statement using induction of the stepping judgement.

- Case $\dfrac{}{ds \vdash \langle c, \rho\rangle \to^* \langle c, \rho\rangle}$:

  Assume that $c = \texttt{fail}$ then $\vDash \forall x_1, \ldots, x_n.\ A \Rightarrow \texttt{false}$ because of the verification conditions, but then $\rho \vDash A \Rightarrow \texttt{false}$, using Lemma 4.4.5 and together with $\rho \vDash A$ we get $\rho \vDash \texttt{false}$ which is a contradiction.

- Case $\dfrac{ds \vdash \langle c, \rho\rangle \to \langle c'', \rho''\rangle \qquad ds \vdash \langle c'', \rho''\rangle \to^* \langle c', \rho'\rangle}{ds \vdash \langle c, \rho\rangle \to^* \langle c', \rho'\rangle}$:

  Here we do a case split on the single step derivation; the most interesting cases are the ones for tail-call and case, we show these below:

  - Case $\dfrac{\forall i.\ \rho \vdash e_i \downarrow \nu_i \qquad f(x_1, \ldots, x_n) = c \in ds}{ds \vdash \langle f(e_1, \ldots, e_n), \rho\rangle \to \langle c, [x_1 \mapsto \nu_1, \ldots, x_n \mapsto \nu_n]\rangle}$:

    Here, by the validity of the verification conditions, we get:

    $$\vDash \forall x'_1 : \tau'_1, \ldots, x'_m : \tau'_m.\ A \Rightarrow \varphi(f)[e_1/x_1, \ldots, e_n/x_n].$$

    Now from Lemma 4.4.5 we get:

    $$\rho \vDash A \Rightarrow \varphi(f)[e_1/x_1, \ldots, e_n/x_n],$$

    which together with $\rho \vDash A$ gives that:

    $$\rho \vDash \varphi(f)[e_1/x_1, \ldots, e_n/x_n].$$

    Now from Lemma 4.4.4 we get:

    $$\rho[x_1 \mapsto \nu_1, \ldots, x_n \mapsto \nu_n] \vDash \varphi(f).$$

    And because $\varphi(f)$ only contains variables from $\{x_1, \ldots, x_n\}$, because of the typing, we have that:

    $$[x_1 \mapsto \nu_1, \ldots, x_n \mapsto \nu_n] \vDash \varphi(f).$$

    And because all functions have valid annotations, we can use IH to give the needed result.

– Case $\dfrac{\rho \vdash e \downarrow k(\nu_1, \ldots, \nu_m) \qquad \begin{array}{c} p_i = k(x_1, \ldots, x_m) \\ \rho' = \rho[x_1 \mapsto \nu_1, \ldots, x_m \mapsto \nu_m] \end{array}}{ds \vdash \langle \texttt{case}\ e\ \texttt{of}\ p_1 \to c_1 \mid \cdots \mid p_n \to c_n, \rho\rangle \to \langle c_i, \rho'\rangle}$:

We have that $\rho' \vdash k_i(x_1, \ldots, x_m) \downarrow k_i(\nu_1, \ldots, \nu_n)$ and because ρ does not contain the variables $\{x_1, \ldots, x_m\}$ by assumption, we also have that $\rho' \vdash e \downarrow k_i(\nu_1, \ldots, \nu_n)$ which means that $\rho' \vdash k_i(x_1, \ldots, x_m) = e \downarrow \texttt{true}$ and therefore:

$$\rho' \vDash k_i(x_1, \ldots, x_m) = e.$$

Similarly, because the variables in A does not contain any variables from the set $\{x_1, \ldots, x_m\}$, we get from the assumption, $\rho \vDash A$ that $\rho' \vDash A$ and therefore:

$$\rho' \vDash k_i(x_1, \ldots, x_m) = e \wedge A.$$

We now have that the IH applies, and we get the needed result.

□

With this theorem we have that, if all verification conditions hold, then the program will not fail, which in turn means (if the implementer has proven Obl. 4.3.11) that the translated tactic and contract automata conform, and then (by Thm. 3.4.26) that the trace-based denotation of the tactic and the contracts conform. We can therefore construct a proof for a conforming substrategy, using the Floyd-Hoare logic defined in this section.

4.4.1 A comment on partial assertions

As mentioned earlier, we impose a type system on programs to avoid partial operators. The problem with partial operators is that assertions containing such operators do not have a directly clear semantic. Consider the example:

$$A \equiv \overline{3}\ \texttt{+}\ \texttt{true}\ \texttt{=}\ \overline{4}.$$

The evaluation of the left side is stuck, and if we use the semantics given above, we have that $\emptyset \nvDash A$, but we also have that the left side of

$$A' \equiv \overline{3}\ \texttt{+}\ \texttt{true}\ \texttt{/=}\ \overline{4}$$

is stuck. So we must also have that $\emptyset \nvDash A'$. However, in classical logic we expect two expressions to be either equal or not equal, which is not the case when including partial assertions. There are several traditional ways of handling this problem:

- Introduce typing as we have done.
- Make all operators total, e.g. by adding explicit errors ($\emptyset \vdash \overline{3}\ \texttt{+}\ \texttt{true} \downarrow \texttt{err}$).
- Require that all assertions we consider are *safe*. Safe means that they evaluate to true or false in any state, e.g the assertion $\frac{5}{x} = y$ is not safe, whereas the assertion $x \neq 0 \wedge \frac{5}{x} = y$ is, assuming short-circuit semantics of $\wedge$.

None of these approaches are completely satisfactory. The first solution introduces a type system, which adds additional complexity, and is not enough for several usages of partial functions (e.g. division by zero, or array indexing). The second solution sidesteps the problem, but in practice oddities turn up. One has to consider e.g. whether there should be a single error expression, `err`, so that:

$$\overline{3}\ \texttt{+}\ \texttt{Tag}(\overline{5})\ \texttt{=}\ \overline{4}\ \texttt{+}\ \texttt{true},$$

is true, or whether several different errors are needed. In the theorem prover Z3 [24], the value of a partial expression can be assigned an arbitrary value, so that both $\frac{0}{0} = 2$ and $\frac{0}{0} = 10$ are true in different queries. It is, however, still considered a function, so in a single query the assertion $\frac{0}{0} = 2 \wedge \frac{0}{0} = 10$ would be false. The third solution is also fairly complex, and requires extra checks before assertions can be assigned a meaningful truth value.

An alternative to these approaches would be to move to a constructive logic with existence predicate (e.g. the logic IQCE by Scott [90]). Such a logic includes a special predicate constant, $\mathbf{E}$, (sometimes written $t\downarrow$), where $\mathbf{E}(t)$ means that the term t exists. With this predicate, the rules for quantifiers and equality are changed to include this predicate, e.g. the rules for the universal quantifier and reflexivity of equality are:

$$\frac{\begin{array}{c}[\mathbf{E}(x)]\\ \vdots\\ A\end{array}}{\forall x.\ A[x/y]} \qquad \frac{\forall x.\ A \qquad \mathbf{E}(t)}{A[t/x]} \qquad \frac{\mathbf{E}(t)}{t = t}$$

Equality between two terms, $s = t$, means that both exist and they are equal. We believe this is a more satisfactory account for partial functions, because it relies on a well-known logic and not some implementation specific ad hoc solution.

Even though constructive logic seems well suited for partial functions, most off-the-shelf theorem provers or decision producers are classical, which made us stick with classical logic in this work.

4.5 Case study

In this section we describe a proof-of-concept implementation of the certification framework. The implementation consists of a type checker and verification condition generator, together with a supervisor and the +2 tactic, and the contracts from Ex. 3.3.18. To prove conformance of the +2 tactic, we have annotated the entire program, and used an external theorem prover to prove the generated predicates. Note that there is nothing in this approach, which requires the use of an external theorem prover; we have only chosen to do so, because the predicates mostly concern inequalities between reals (for timing constraints), which are easy to verify with a reasonable theorem prover. Another reason, which we also see later, is that the predicates became fairly large and therefore tedious to reason about by hand. We, however, consider this an artefact of this proof-of-concept implementation, and we leave practical realisations of the framework for future work. In the rest of this section we describe the various parts of the implementation.

4.5.1 Type checker and VC generator

All helper programs (parser, type checker, vc generator and theorem prover interface), are written in Haskell. The implementation of the different components is straightfor-

ward based on the language definition presented in Sect. 4.2. The only difference to the presented language is that the implemented language includes an if-then-else construction on the expression level, with the obvious semantics. The parser was created with the Parsec combinator library. As interface for external theorem provers we have chosen the SMT-LIB Standard: Version 2.0 [8], which is recognised by several theorem provers. Although it is designed for satisfiability checkers, we can use it to check validity also, because a formula is valid if and only if its negation is not satisfiable. The source code is available on the Author's web page [1].

4.5.2 Example program

The source code for the entire example program is provided in Appendix A.1. The tactic is implemented directly as written in Ex. 4.3.1. Similarly, the contracts are implemented directly as the one given in Ex 4.3.6. The full supervisor, however, is fairly large, and while not conceptually complicated, contains a fair bit of machinery for implementing the conformance relation of Def. 3.4.23. The implementation of the supervisor follows the general pattern of Sect. 4.3.3. In the following we describe the various parts in detail.

The loop function is mostly implemented as stated, the traces are however divided into traces over a single link, this seems to make the splitting of traces simpler. Furthermore, we distinguish between the input and output traces, because the output traces can contain at most one element. This gives the following definition of the trace types:

```
-- Input trace: lo1,li2
type train = NoIn | Actin : real, int, train

-- Output trace: li1, lo2
type traout = NoOut | Actout : real,int
```

Which gives the following header for the loop function:

```
loop(tnow : real, tend : real,
     ttac : real, ts : tState,
     tcon1 : real, rs1 : run1,
     tcon2 : real, rs2 : run2,
     alo1 : train, ali2 : train,
     ali1 : traout, alo2 : traout)
```

When the loop function is called, it checks whether the current time (`tnow`) is less than the end time (`tend`), and if so the program continues, otherwise it terminates with `done`, because there is nothing more to check. After this comparison, the traces are split into the action at the particular time. Because there is no case on expression level, this is done by several function invocations:

```
splitLo1(tnow : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case alo1 of
    NoIn -> splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2,Emp)
  | Actin (tlo1,lo1,alo1_) ->
    if tlo1 <= tnow then splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
```

[1] http://www.diku.dk/hjemmesider/ansatte/starcke/

```
                                    alo1_,ali2,ali1,alo2,V(lo1))
                        else splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                    alo1,ali2,ali1,alo2,Emp)
  end
```

and so on for the remaining links.

After the splitting of traces, the tactic is invoked in two rounds: first the timer time is checked, and then the input is checked. In each case the supervisor's state is tagged and used as an input parameter. The timer transition function is invoked in the following way:

```
-- Supervisor state
type sTac1 = STac1 : real,real,real,real,run1,real,run2,
                     train,train,traout,traout,
                     act,act,act,act

-- Invoke tactic
invTac(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  if ttac <= tnow
  then tacT(STac1(tnow,tend,ttac + 1.0,tcon1,rs1,tcon2,rs2, -- Timeout
                  alo1,ali2,ali1,alo2,lo1now,li2now,li1now,lo2now),ts)
  else tacRetT(STac1(tnow,tend,ttac,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2,lo1now,li2now,li1now,lo2now),
               ts,Emp,Emp)
```

The number `1.0` is the time between timer transitions of the tactic. The invocation of the input transition is similar. After the two tactic transitions, the output traces are updated with the actions returned from the transitions:

```
updLi1(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act,
       li1 : act,lo2 : act) =
  if ali1 = NoOut
  then case li1 of
         Emp -> updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                       NoOut,
                       alo2,lo1now,li2now,li1now,lo2now,lo2)
       | V(n) -> updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                        Actout(tnow + 1.0,n), -- latency
                        alo2,lo1now,li2now,li1now,lo2now,lo2)
       end
  else updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
              ali1,
              alo2,lo1now,li2now,li1now,lo2now,lo2)
```

The action is added to the output trace, and the number `1.0` is the latency of the link. After the output actions are added to the output trace, the first contract (+1) is invoked, which is done in a manner similar to the tactic; however, one has to account for the

case where the contract is already done. The state of the supervisor is again saved and passed as the first parameter:

```
-- State of the supervisor
type sCon1 = SCon1 : real,real,real,tState,real, real, run2,
                     train,train,traout,traout,
                     act,act,act,act

-- Invoke contract 1
invCon1(tnow : real, tend : real,
        ttac : real, ts : tState,
        tcon1 : real, rs1 : run1,
        tcon2 : real, rs2 : run2,
        alo1 : train, ali2 : train,
        ali1 : traout, alo2 : traout,
        lo1now : act,li2now : act,li1now : act,lo2now : act) =
  case rs1 of
    Done1(k) -> invCon2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                        alo1,ali2,ali1,alo2,
                        lo1now,li2now,li1now,lo2now)
  | Run1(cs) ->
    if tcon1 <= tnow
    then con1T(SCon1(tnow,tend,ttac,ts,tnow + 1.0, -- Timeout
                     tcon2,rs2,alo1,ali2,ali1,alo2,
                     lo1now,li2now,li1now,lo2now),cs)
    else con1RetT(SCon1(tnow,tend,ttac,ts,tcon1,
                        tcon2,rs2,alo1,ali2,ali1,alo2,
                        lo1now,li2now,li1now,lo2now),rs1)
  end
```

When the first contract is done, the second contract is invoked in the same manner, and finally, after the second contract is finished, control moves to the pay-off calculation:

```
con2RetI(ss : sCon2, rs2 : run2) =
  case ss of
    SCon2(tnow,tend,ttac,ts,tcon1,rs1,
          tcon2,alo1,ali2,ali1,alo2,
          lo1now,li2now,li1now,lo2now) ->
      payoff(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
             alo1,ali2,ali1,alo2,
             lo1now,li2now,li1now,lo2now)
  end
```

The pay-off calculation checks the pay-off of each contract, and if the sum is negative, the program fails with the `fail` command:

```
payoff(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  case rs1 of
    Done1(k1) -> case rs2 of
                   Done2(k2) -> if k1 + k2 < 0.0 then fail
                                else done
                 | Run2(cs2) -> if k1 < 0.0 then fail
                                else finish(tnow,tend,ttac,ts,
                                            tcon1,rs1,tcon2,rs2,
                                            alo1,ali2,ali1,alo2,
```

```
                                              lo1now,li2now,li1now,lo2now)
                end
  | Run1(cs1) -> case rs2 of
                   Done2(k2) -> if k2 < 0.0 then fail
                                else finish(tnow,tend,ttac,ts,
                                            tcon1,rs1,tcon2,rs2,
                                            alo1,ali2,ali1,alo2,
                                            lo1now,li2now,li1now,lo2now)
                | Run2(cs2) -> finish(tnow,tend,ttac,ts,
                                      tcon1,rs1,tcon2,rs2,
                                      alo1,ali2,ali1,alo2,
                                      lo1now,li2now,li1now,lo2now)
                end
  end
```

After the payoff calculation, the next timepoint has to be found as the minimum amongst the timeouts of the tactic and the contracts and the actions on the traces:

```
finish(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  let tmin = if ttac < tcon1
             then if ttac < tcon2 then ttac else tcon2
             else if tcon1 < tcon2 then tcon1 else tcon2
  in firstLo1(tnow,tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
              alo1,ali2,ali1,alo2)
  end
```

Each link is considered in turn, and the last comparison finishes with a call back to the loop function:

```
firstLo2(tnow : real, tmin : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case alo2 of
    NoOut -> if tnow < tmin
             then loop(tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                       alo1,ali2,ali1,alo2)
             else done
  | Actout (tlo2,lo2) ->
    if tnow < tmin
    then loop(if tlo2 < tmin then tlo2 else tmin,
              tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,ali1,alo2)
    else done
  end
```

Because the data structure for the input does not enforce that the time points are strictly increasing, this is checked before the program loops.

The full program might seem fairly big, but it is mostly due to the quite restrictive language. And while we could have extended the language with additional constructions, we have preferred as simple a language as possible, both to make the presentation more clear, and to make the Floyd-Hoare logic simpler. A possible extension is the addition of an expression level case; but then one would either need a case expression in the logic

as well, or to translate the expression in some way. Because we wanted to use an off-the-shelf theorem prover in the end, we did not want to force it to include a case expression, and for the sake of simplicity, we wanted to use the expressions directly in the assertions. But extending and modifying the language, based on practical experience, is certainly an important part of future work.

Another point is that most of the program is the supervisor, which is not dependent on the actual tactics and contracts, therefore, when trying to certify other tactics and contracts, most of the program can be used unchanged.

We do not prove Obligation 4.3.11 here, because it is a fairly tedious proof and only expresses that this concrete implementation is correct. It would be interesting to investigate how to generate the supervisor automatically from a given scenario (consisting of the number of contracts and the links), and in that case prove that all generated supervisors are correct. We leave this for future work.

4.5.3 Program annotation and theorem proving

To prove that the function conforms with the contracts, one has to annotate all functions with an assertion and then use the verification condition generator, to extract predicates, which then has to be proven valid. As mentioned earlier, the predicates tend to be simple but tedious to prove, so we have used an automatic theorem prover to check that the predicates are valid. The requirements for a theorem prover are that it is able to handle integer and real arithmetic, and that we can translate our data types in a fairly straightforward manner. Concretely, we have used the Z3 prover by Microsoft Research [86], and we have not experienced problems with the proofs. The only non-standard configuration parameter we have used is `ELIM_QUANTIFIER=true`, which is used to eliminate quantifiers in cases with arithmetic variables. This is useful, because our formulas include a lot of quantifiers to handle the data types. The prover includes support for algebraic data types, so our data types can be translated almost directly. For example the `train` type can be modelled with:

```
(declare-datatypes
  ((train NoIn
          (Actin (Actin_1 Real)
                 (Actin_2 Int)
                 (Actin_3 actin)))))
```

With the proof being done automatically, the only thing that has to be implemented is the annotations for each function. Unfortunately, there are quite a few functions to consider, and each takes a lot of arguments. The contracts, while functionally very simple, still have a fairly complex timing interaction, so there are several different stages in the program. Additionally, the +1 service can fail at any time, and while the pay-off received in this case is enough to cover any future losses, this is not reflected in the program and has to be captured by the annotations as an invariant. So in general, while the effect of the program is simple, the reason for its correctness is not.

All these different states are modelled as a big disjunction in the assertions, and because there are a lot of functions, the resulting annotations are very large ($\sim$ 4500 lines in total), a lot of those lines come from repetition in the annotations, because most functions are rather simple, so the invariant changes only a little.

To give a feel for the annotations, we show the ones for the loop function, because they illustrate the different states the program goes through. The remaining annotations can be seen as just propagating these states through the program.

```
----- LOOP -----
loop :
  -- Either contract is finished with positive payoff (State1)
  (rs1 = Done1(1.0) /\
    (Ex cs2 : cState2. rs2 = Run2(cs2) \/
     Ex k : real. 0.0 <= k + 1.0 /\ rs2 = Done2(k)))
  \/
  (rs2 = Done2(1.0) /\
    (Ex cs1 : cState1. rs1 = Run1(cs1) \/
     Ex k : real. 0.0 <= k + 1.0 /\ rs1 = Done1(k)))
  \/
  -- First input not received (State2)
  (ts = S1 /\
   tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1start) /\
   tnow <= tcon2 /\ rs2 = Run2(C2start) /\
   ali1 = NoOut /\ alo2 = NoOut)
  \/
  -- Input on li2 received, 1. output on li1 not received (State3)
  (Ex n : int. Ex tn : real. Ex tc2 : real.
   tn + 20.0 <= tcon2 + tc2 /\
   tnow <= tn + 1.0 /\
   ts = S2 /\
   tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1start) /\
   tnow <= tcon2 /\ rs2 = Run2(C2run(n,tc2)) /\
   ali1 = Actout(tn + 1.0,n) /\ alo2 = NoOut)
  \/
  -- 1. output on li1 sent (State4)
  (Ex n : int. Ex tn : real. Ex tc1 : real. Ex tc2 : real.
   tcon1 + tc1 <= tn + 5.0 /\ 0.0 <= tc1 + 1.0 /\
   tn + 20.0 <= tcon2 + tc2 /\
   ts = S2 /\
   tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1run(n,tc1)) /\
   tnow <= tcon2 /\ rs2 = Run2(C2run(n,tc2)) /\
   ali1 = NoOut /\ alo2 = NoOut)
  \/
  -- 1. input on lo1 received, 2. output on li1 not received (State5)
  (Ex n : int. Ex tn : real. Ex tn1 : real. Ex tc2 : real.
   tn1 <= tn + 6.0 /\
   tn + 20.0 <= tcon2 + tc2 /\
   tnow <= tn1 + 1.0 /\
   ts = S3 /\
   tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1start) /\
   tnow <= tcon2 /\ rs2 = Run2(C2run(n,tc2)) /\
   ali1 = Actout(tn1 + 1.0,n + 1) /\ alo2 = NoOut)
  \/
  -- 2. output on li1 sent (State6)
  (Ex n : int. Ex tn : real. Ex tc1 : real. Ex tc2 : real.
   tcon1 + tc1 <= tn + 11.0 /\ 0.0 <= tc1 + 1.0 /\
   tn + 20.0 <= tcon2 + tc2 /\
   ts = S3 /\
   tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1run(n+1,tc1)) /\
   tnow <= tcon2 /\ rs2 = Run2(C2run(n,tc2)) /\
   ali1 = NoOut /\ alo2 = NoOut)
  \/
  -- 2. input on lo1 received, 1. output on lo2 not received (State7)
  (Ex n : int. Ex tn : real. Ex tn2 : real. Ex tc2 : real.
```

```
tn2 <= tn + 12.0 /\
tn + 20.0 <= tcon2 + tc2 /\
tnow <= tn2 + 1.0 /\
ts = S1 /\
tnow <= tcon1 /\ tcon1 <= tnow + 1.0 /\ rs1 = Run1(C1start) /\
tnow <= tcon2 /\ rs2 = Run2(C2run(n,tc2)) /\
ali1 = NoOut /\ alo2 = Actout(tn2 + 1.0,n+2))
```

We briefly explain each state:

State1. In this state at least one of the contracts is finished with a pay-off of 1, and because we know that the maximum negative pay-off is –1, then we do not need to know more.

State2. In this state the tactic and both contracts are at their starting point, and nothing has been received so far.

State3. In this state the first input on l_{in2} has been received by the tactic and the +2 contract. The variable `tn` keeps the time for that input, and the tactic has produced output on l_{in1}, which is received at `tn + 1.0`. The first inequality ensures that there is time enough for the tactic to receive the values from the +1 service.

State4. In this state the output on l_{in1} has been received and the +1 contract is now also running. The extra inequality ensures that the output from the +1 service is obtained within reasonable time. Note that the value of the input `n` is the same for both contracts.

State5. This state is similar to state3 in that the tactic has produced output on l_{in1}, which has not been received yet.

State6. This state is similar to state4, now the tactic is waiting for input from the +1 service again.

State7. In this state the tactic has received the value and sent it on l_{out2}, but it has not been received yet.

As we can see, the timing inequalities are not entirely simple and has to be accounted for in the annotations.

These annotations have been propagated all the way through the program by hand. The generated predicates have all been proven valid by Z3 version 2.19 in less than 10 seconds on a normal laptop. Even though the complete annotations are very big, this example shows that the framework is possible to implement. As a small experiment we modified the implementation to allow functions without annotation. When such functions are being called the verification generation just continues with the body of the function, as if the call was just a series of let expressions. With this implementation we could remove all annotations for functions that are only called once, thus reducing the size of the annotations to around 3000 lines with a slightly faster running time as well (9 seconds). We also tried removing almost all annotations, leaving only the ones for loop and the invocation of tactic and contracts (4 labels in total). This reduced the size of the annotations to less that 500 lines, but extended the checking time to 9 minutes. The larger running time is due to the fact that when a function is called several times, the same piece of code is checked several times. But nevertheless this shows that there is lots of room for improvements, but how to make it practically applicable is left for future work.

4.6 Related and future work

In this section we consider related and future work.

4.6.1 Related work

We already mentioned related work on proof-carrying code. A way of extending traditional PCC to concurrency is to take one of the existing proof systems for concurrency (e.g. the system by Hooman [48]) and then mechanise derivations in that system. These systems are, however, not as well understood as standard Floyd-Hoare logic, and it is not entirely clear what the new safety theorem should say. In comparison, our model uses a standard logic, which is well understood, and should therefore be able to benefit from the development of sequential PCC.

Our certification paradigm, verification-time monitoring, is related to runtime monitoring, and especially a paradigm called runtime verification. In runtime verification, standard verification techniques, like model checking, are applied to execution runs to check whether they conform with a given correctness property. Leucker and Schallhart [51] give an overview of the runtime verification paradigm. Compared to runtime verification, our work have been focused on the static certification before runtime. But we do not see any conceptual problems with using our framework for dynamic verification.

The idea of expressing contracts and processes in a single language, for which error-free execution corresponds to conformance, bears similarities to other approaches. Vardi and Wolper [98] show how to express both programs and specifications as automata and thereby reduce the problem of whether the program satisfies a specification to an automata-theoretic property of their composition as automata. Findler and Felleisen [28] introduce contracts for higher-order functions. These contracts are compiled into the code of the functions, and these modified functions can then fail and assign blame, if a contract is broken. In our terminology, the test harness and the tactic are compiled to a single program which is then monitored, they do, however, not specify any methods for static fulfilment of the contracts. Xu, Jones and Claessen [103] show how to use a similar approach for Haskell, and how to incorporate static checking. The general idea being that the composition of a contract and a program returns an error if the contract is broken. Standard techniques can then be applied to show that the composition never returns an error. While the idea is similar to our approach, there are several differences. In our approach, contracts are a separate part of the test harness, and it would therefore be possible to e.g. use different languages for the tactic and the contracts, in the Haskell framework they are tied more closely together (which also makes it more integrated, and probably more straightforward to use). Our approach also allows contracts, which can capture both communication behaviour and real-time guarantees.

4.6.2 Future work

We describe paths for future work:

Concrete scenarios. As part of future work, more concrete scenarios should be investigated to estimate the practicality of the paradigm. More real-time programs should be certified, and more high-level scenarios could be attempted, in order to see which features are lacking. Another aspect is also the usability for testing or runtime monitoring in practice, which has not been investigated thoroughly.

Verification-time monitoring. Because the verification-time monitoring paradigm is very general and not fixed to our concrete models, a direction for future work is too look at other models, for which verification-time monitoring can be used as a syntactic model for correctness. A simple start is to look at changes to the conformance relation (e.g. for some of the earlier mentioned extensions). But in general, the paradigm should be applicable to any method in which a suitable supervisor can be formalised.

Certification implementation. There are several extensions to the concrete framework presented here that could be explored. Firstly, the language used could be modified; we originally used a machine code language, but preferred the coroutine language introduced in this chapter, because of better control with the free variables – at the cost of a little more complex Floyd-Hoare logic.

In addition to the language itself, it is also interesting to experiment with different implementations of the supervisor, as long as they still behave faithfully to the conformance relation. Because the entire program is annotated, another supervisor implementation might lead to easier certifiable programs. In the same direction are also different methods for proving that the supervisor avoids the error state. E.g. a suitable type system.

Another direction for future work, which is not only interesting for this work, is the development of a constructive Floyd-Hoare logic, with a more satisfying account for partial functions.

Proof engineering. To make the concrete framework developed here usable in practice, it would be interesting to consider various methods to improve the proof obligations. One idea is to use higher-level languages as a basis, and then generate the concrete tactic, contract and supervisor components automatically. In some cases it might then be possible to generate the certification proof automatically, by exploiting special properties of the high-level code. This is similar to the design of a certifying compiler in proof-carrying code.

Another idea is to exploit the fact that the supervisor program is the same given an unchanged link topology. This might make it easier to generate part of a proof or annotations automatically. Similarly, we might even let a theorem prover try to annotate parts of the program, or maybe do an interactive process when proving, similar to interactive theorem proving.

In the case study we used a theorem prover to verify the verification conditions; to do proper certification, one also needs a representation of the proof. Here, one can hope to use a certifying theorem prover, use parts of the certifying compiler as described above, or of course do the proofs by hand.

Chapter 5

Towards resource-aware interaction

Typical business contracts include a notion of a physical transfer of goods or *resources*. As briefly mentioned in Sect. 3.5.2, we can we can encode linear (non-duplicatable, single use) resources with the help of a resource manager principal. The interface of such a resource manager can be specified through contracts, which makes sure that linearity constraints are satisfied. While this approach works in principle, it is cumbersome to use in practice, especially with many different kinds of resources. In this chapter we discuss an extension to the model to allow for first-class resources. Note, however, that the work presented here is preliminary. We sketch, how the resources extend the previous model, but we postpone the precise mathematical definitions, of e.g. conformance, to future work.

Resource transfers are handled by the agents. A crucial property of resources is that they have a general meaning (but not necessarily a general utility), so that a resource obtained from one contract can be used to fulfil an obligation in another contract.

Our model of resources will allow dynamic introduction of resources; that is, we allow the creation of resources local to a principal, and these resources can then be transferred to other principals, acting as a handle for the original principal. A benefit of this dynamic structure is that we can use resources to model dynamically changing communication setups; in particular, we no longer need the static notion of channels or links.

5.1 Resource setup

Before presenting the formal model, we revisit the business protocol example (Ex. 2.1.2). This example is a bit incomplete, because the goal for Buyer seems to be to obtain 'delivery details', which do not seem to have a clear value. Of course, the goal is for Buyer to actually obtain the ordered item, and not only the delivery details. We therefore show a scenario, in which there is an actual delivery of goods:

Example 5.1.1 (Delivery protocol)**.** We consider the interactions from the viewpoint of a book-selling webshop. A customer contacts the webshop, asking for a specific book, and the price ($2) is paid during the order. The webshop must then deliver the book to the address specified by the customer before the deadline of 21 days.

The webshop does not necessarily keep all books in store, and some must therefore be ordered from a publisher unknown to the customer. For simplicity, the publisher and

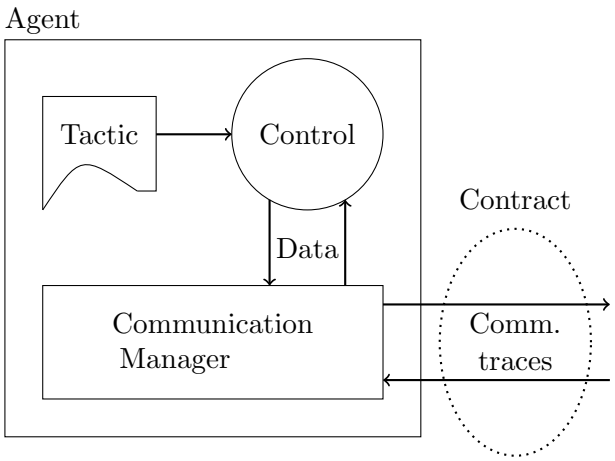


Figure 5.1: Communicating agent.

the webshop have the same type of sale contract, but with different parameters. We assume a price of $1 and a delivery deadline of 5 days.

In the resource-aware model, every interaction is codified with resource exchanges. A principal who currently holds a resource can either interact with it directly (which in general results in the resource being altered), or transfer it untouched to another principal. It is the agents of the principals that perform the actual resource exchanges, just like they performed the actions in the previous model. To be able to capture data transmissions (actions on links), we model agent addresses as resources as well. As all resources are single-use, an address can only be used once. One can view such an address as a service certificate that a principal can transfer to another principal. That principal can then either use it or transfer it again. To model persistent data communication links, the receiver of a transfer (whose resource was consumed) can create a new resource immediately, and send it back to the original sender. This new resource can be seen as an acknowledgement of the first message received, and the new resource ensures that communication can proceed.

Before presenting the different extensions in order to to get a resource-aware model, we wish to discuss the relation to the resource-less model. This will hopefully give a clearer understanding of the concepts. To aid the discussion, we show two conceptual pictures of an agent. Fig. 5.1 illustrates a purely communicating agent. One can think of an agent as consisting of two parts: a control unit and a communication manager. The control unit runs the program given by the tactic, and orders the communication manager to perform the actual communication. Likewise the control unit should be informed of incoming messages. Each contract watches the communication, and returns a verdict based on its rules. In the conformance relation we encode the effect of the communication manager (e.g. the latency), and therefore we can view the contracts in terms of the inputs and outputs from the control unit. In the resource-less model there are no essential benefits in viewing control and the communication manager as separate objects, because ultimately they both are exchanging bits. In the resource-aware model, illustrated in Fig. 5.2, the communication manager is replaced with a resource manager (the shaded component). The other parts stay the same. The role

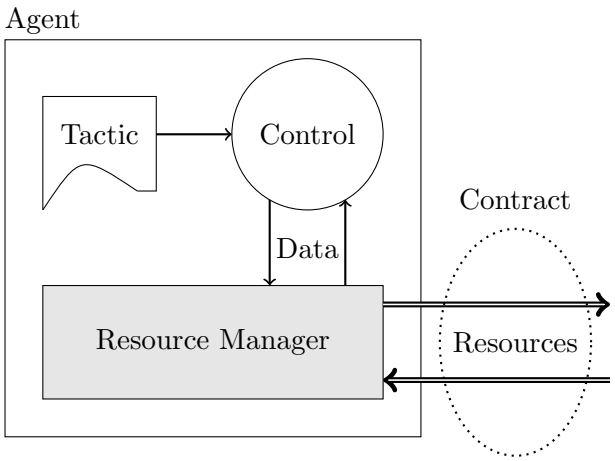


Figure 5.2: Resource-exchanging agent.

of the resource manager is similar to the one of the communication manager, but it operates on physical resources instead of only messages. Physical resources cannot be used freely, and therefore certain requests from control are nonsensical, such as a request to transfer a resource that the manager does not possess.

Resources are defined by agents, but can be so in two different ways: either each resource is a single unique agent, or each resource can represent a claim of some service. We illustrate the difference with a book example:

- A specific physical book is modelled as a specific agent. One holding that book's resource can interact with it, e.g., by asking for a particular page, which is then returned as data. Because resources are single-use, the book agent returns a new version of the book together with the requested page. This can be used to model that the book can become worn after several uses, e.g. suddenly some pages might not be available any more. In general this means that interacting with a resource might have a side effect on the resource itself.

- A publisher offers a subscription model for reading the book on-line. Here the agent corresponds to the website, which allows queries for specific pages, which are then returned to the user. These queries are data requests, which are different from actual resource interactions. In this case it is also possible to specify constraints, e.g. a maximum number of pages that can be read. This option is very similar to the bank example considered in Sect. 3.5.2.

In the book example, we actually have a third option; to model the book as a downloadable document, representing it as purely data and not as a resource.

It is important to consider how resources affect the contractual obligations. Consider the book selling example from before: when the publisher transfers the book resource to the webshop, it is specified in their contract, how the resource can be used. If the webshop afterwards transfers the book to a customer that it has another contract with, the publisher's obligations are not changed. This means that the publisher can only be held accountable by the webshop. So if the book is faulty in some way, e.g., if the pages are torn or wrong, then the customer can only complain to the webshop. The webshop must then in turn complain to the publisher. This interpretation gives a robust model

in terms of delegation, because a principal only needs to look at his own contracts to see, which obligations needs to be fulfilled.

We do not specify the internal structure of resources, only that each resource is defined by a single agent, as described above:

Definition 5.1.2. *Resources, r, are taken from an abstract set* **Resource**. *Each resource is associated with a single agent, captured by a function:*

$$\text{agent} : \mathbf{Resource} \to \mathbf{Agent}.$$

Each resource transfer is called an *event*. As mentioned above, each resource can be viewed as a single-use address or pointer to the agent defining the resource. Each event contains the destination (represented with the receiving agent's resource), zero or more resources transferred and a single data value. The data value is used to model communication of pure information; this value can be freely copied or discarded, unlike the resources. Often the value will be used to describe the resources to the recipient. For simplicity, we assume that values are taken from some universal set of values, **Value**, instead of assigning restricted value sets. But it could also be possible to restrict the value set based on e.g. the destination resource.

Definition 5.1.3. *An event is a resource transfer:*

$$\mathbf{Event} := \mathbf{Resource} \times \mathbf{Value} \times \mathbf{Resource}^*,$$

When $e = (r, v, [r_1, \ldots, r_n])$ then the resources $r_1, \ldots, r_n$ are transferred to the agent, $\text{agent}(r)$. *The value carried is specified by v.*

The last part of the resource-aware communication model is timing. Like with plain communication each resource transfer, takes a certain amount of time. For simplicity, we assume that this time is only dependant on the sender and receiver.

Definition 5.1.4. *The function* $\text{lat} : \mathbf{Agent} \times \mathbf{Agent} \to \mathbf{TimeD}_+$ *assigns a positive latency to each pair of sending and receiving agents.*

It is possible to have a more complex latency definition, e.g. by letting it evolve over time or depending on the entire destination resource and not only the agent. But this simple definition should suffice. Compared to the link-based model, this corresponds to a single link (in each direction) between each pair of agents. Some latencies might even be effectively ∞, if those pairs of agents cannot communicate directly.

5.2 Contracts and tactics

We sketch the extension of contracts and tactics.

5.2.1 Contracts

Similar to how resource-less contracts monitor the communication, the resource-aware contracts monitor the resource transfers. We do, however, have to take the dynamic behaviour of resources into account. When we considered traces in the resource-less model, the set of links was fixed, and the contracts could monitor these links without having to know all links in the universe. Resources are, however, created and destroyed (through usage) dynamically, and having another agent's resource corresponds to having

a relevant link to that agent. Therefore, it is not a priori clear, which events a given contract depends upon.

The approach we take is to ensure that all contracts are *finitely supported*, that is, for every state in the contract, there are only a finite set of destination resources which can affect the contract. Consider a contract between two principals that starts with support of a single resource, r_1. After a while that resource is used as destination in a transfer $(r_1, v, [r_2, r_3])$ and thus consumed; it is therefore removed from support, but r_2 and r_3 are added instead, and so on. This idea ensures that at every point in time, only a finite amount of resources are included in the support. In the link terminology it corresponds to having a finite but different, set of links at every time point. To formalise finite support for contracts, we introduce a notion of support for traces of events.

Definition 5.2.1. *An event trace is a list of time-stamped events:*

$$\omega \in \mathbf{TraceE} := (\mathbf{Time} \times \mathbf{Event})^*.$$

The list is sorted according to the time-stamps in strict ascending order[1]*. The support predicate* $(\cdot \vdash \cdot) \subseteq \mathcal{P}_{\text{fin}}(\mathbf{Resource}) \times \mathbf{TraceE}$*, is defined inductively in the following way:*

$$\begin{aligned} &R \vdash [\,] \text{ always}\,, \\ &R \vdash [(t, (r, v, [r_1, \ldots, r_n]))] +\!\!+ \ \omega \text{ iff} \\ &\quad r \in R \wedge (R \setminus \{r\}) \cup \{r_1, \ldots, r_n\} \vdash \omega. \end{aligned}$$

To capture all traces supported by a given set of initial resources, we define the set:

$$\mathbf{TraceE}_R := \{\omega \in \mathbf{TraceE} \mid R \vdash \omega\}.$$

The support predicate ensures that at every point in time only transfers to the supporting resources can occur in the trace. Similar to the resource-free traces we have notation for time restricted traces:

Definition 5.2.2. *Traces restricted to an end time are defined in a straightforward way:*

$$\mathbf{TraceE}^t_R := \{\omega \in \mathbf{Trace}_R \mid \forall (t', e) \in \omega.\ t' \leq t\}.$$

Similarly, for the restriction of a single trace to a time, $\omega_{\leq t}$*.*

A contract is defined with a finite set of starting resources, and then it gives a verdict based on a trace supported by those starting resources, very similar to the definition of resource-less contracts:

Definition 5.2.3. *A (resource-based) contract regarding a finite set of starting resources,* $R \subseteq_{\text{fin}} \mathbf{Resource}$ *is a function:*

$$c \in \mathbf{Contract}_R := \prod_{t \in \mathbf{Time}} (\mathbf{TraceE}^t_R \to \mathbf{Verdict}),$$

satisfying a monotonicity condition:

$$\forall \omega \in \mathbf{TraceE}^{t'}_R.\ t \leq t' \Rightarrow \big(c(t)(\omega_{\leq t}) = \bot \vee c(t)(\omega_{\leq t}) = c(t')(\omega)\big)$$

[1] For simplicity we assume that no two events happen at the same time; to overcome this restriction we could consider sets of events at every time point.

We consider the book selling example:

Example 5.2.4 (Continuing Ex. 5.1.1)**.** We do not formalise the contract functions required for the example, but describe the needed contracts, and how the supporting resources evolve. From the viewpoint of the webshop, there are two contracts:

- c_{wc}: between the webshop and the customer.
- c_{wp}: between the webshop and the publisher.

For both contracts, the start resources consist of a single resource. In the first contract this resource is an address that the customer can use to contact the webshop, and in the second contract it is an address that the webshop can use to contact the publisher. To illustrate the supporting resouces, we describe how both contracts evolve in a successful case (in the following R_{wc} is the supporting resource set for c_{wc} and similarly for R_{wc}):

1. Both contracts are in their initial state.

$$R_{\text{wc}} = \{r_{\text{w}}\}, \quad R_{\text{wp}} = \{r_{\text{p}}\}.$$

2. The customer contacts the webshop with the name of the book, two $ resources, a delivery address, and an address the webshop can use to send a new resource for another order. Concretely, the following event (resource transfer) takes place:

$$(r_{\text{w}}, \text{“The Wind in the Willows”}, [r_{\$1}, r_{\$2}, r_{\text{addr}}, r_{\text{c\_ret}}]),$$

   resulting in the following updated event sets:

$$R_{\text{wc}} = \{r_{\$1}, r_{\$2}, r_{\text{addr}}, r_{\text{c\_ret}}\}, \quad R_{\text{wp}} = \{r_{\text{p}}\}.$$

3. The webshop contacts the publisher with one $ resource, the customer's delivery address and a new return address for the publisher. In addition, the webshop sends a new contact resource back to the customer. This is captured by the following two events:

$$(r_{\text{p}}, \text{“The Wind in the Willows”}, [r_{\$1}, r_{\text{addr}}, r_{\text{w\_ret}}]),$$
$$(r_{\text{c\_ret}}, \text{“Renewed webshop address”}, [r'_{\text{w}}]).$$

   After these events the resource sets are:

$$R_{\text{wc}} = \{r_{\$1}, r_{\$2}, r_{\text{addr}}, r'_{\text{w}}\}, \quad R_{\text{wp}} = \{r_{\$1}, r_{\text{addr}}, r_{\text{w\_ret}}\}.$$

4. The publisher delivers the book to the customer's address, and transfers a new contact address to the webshop:

$$(r_{\text{addr}}, \text{“Delivery of The Wind in the Willows”}, [r_{\text{book}}]),$$
$$(r_{\text{w\_ret}}, \text{“Renewed publisher address”}, [r'_{\text{p}}]).$$

   The event sets are updated to be:

$$R_{\text{wc}} = \{r_{\$1}, r_{\$2}, r_{\text{book}}, r'_{\text{w}}\}, \quad R_{\text{wp}} = \{r_{\$1}, r_{\text{book}}, r'_{\text{p}}\}.$$

After the delivery of the book, the example could continue with the customer interacting with the book resource (r_{book}). Note that because the book resource is present in the resource set of both contracts, any interaction with the book will be relevant for both contracts. This means that if the book is defective, this could be registered by both contracts. Whether the webshop is covered by the contract with the publisher in terms of failure depends, however, on the concrete contracts.

The example illustrates, how resources obtained in one game ($ obtained from the customer) are used to obtain other resources (books from the publisher), and those can again be used to obtain more resources.

Another aspect of this example is that the webshop is $1 'richer' after the interaction, but because the contracts are zero-sum, this profit has to come from somewhere. If the book is assumed to have a fixed price, either the publisher or the customer loses money in the interaction, and then one might wonder, whether one of them was acting irrationally. We take another viewpoint, namely that there are no fixed prices for resources, meaning that each principal might have a different utility for each resource. An interaction then represents a growth in the accumulated utility across all principals. This utility is not encoded formally, but instead captured only by the contracts a given principal has, e.g. the customer might highly value being able to read in the book. With this viewpoint, each principal can still have a rational reason for committing to the interactions.

Related to the utility discussion is the question of where the resources come from. Interactions, as described above, only pushes resources around. The creation of specific resources could still be modelled with resource exchanges. A book might be created by combining paper with ink in a printing press. But ultimately these components could be viewed as coming from Nature. In this way, Nature becomes the only participant that is disinterested in utility, and is both able to generate value and absorb losses.

5.2.2 Tactics

The tactics in the resource-aware model are similar to the resource-less tactics. But whereas the resource-free tactics could effectively specify the outputs of the communication manager directly (only constrained by the link latencies), the output of the resource-aware tactic must give orders to the resource manager, which then performs the resource transfers. Staying in the terminology, the output of a tactic is still actions, but the resource manager converts these into events. Similarly, incoming events are converted into observations. These concepts are illustrated on Fig. 5.3.

There are several ways of defining actions and observations. They must contain a suitable way of referring to the physical resources, and have some way of specifying the creation of new resources. We show one way of formalising them, but other approaches are possible. Our formalism is based on the tactic having a set of *handles* for the actual resources. When incoming transfers are observed, the transferred resources are assigned new handles by the resource manager, and requested actions can then be specified in terms of these handles. The creation of new resources is modelled by allowing the tactic to refer to handles that are not already in use, which forces the creation of new resources with those handles. These handles are remembered, so that when their corresponding resource is used as an address resulting in an observation, the tactic is informed of which handle was used at that point. The recipient cannot identify the exact sending agent,

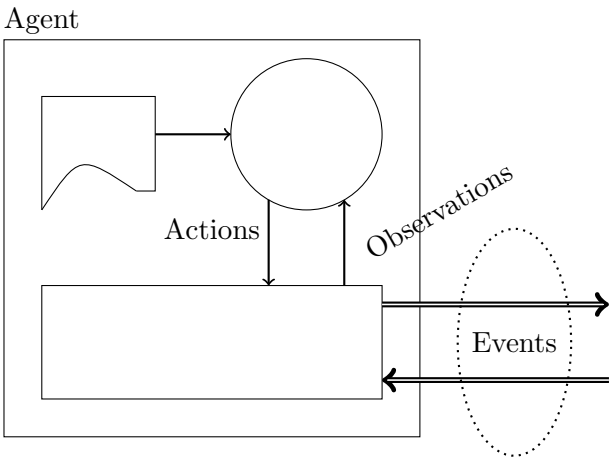


Figure 5.3: Actions, observations and events.

but knows which ones of the outstanding communication offers was used (which handle), and can therefore identify the responsible principal.

This ensures that when several agents transfer resources, they can be distinguished from each other.

Definition 5.2.5. *We assume an unspecified set of handles,* **Handle***, exists. An observation contains a value, a list of new handles (for referring to the new resources received) and the handle for the resource that was used as an address (this handle is now free to be reused by the tactic):*

$$\mathbf{Observation} := \mathbf{Value} \times \mathbf{Handle}^* \times \mathbf{Handle}.$$

Similarly, an action contains a value to be transmitted, a list of handles (for the resources sent) and a special handle that specifies the receiver: [2]

$$\mathbf{Action} := \mathbf{Value} \times \mathbf{Handle}^* \times \mathbf{Handle}.$$

Similar to traces over events, traces over actions and observations are just lists of actions/observations with time points:

Definition 5.2.6. *A trace over actions (resp. observations) is defined as a list of timestamped actions (resp. observations) sorted by the timestamp in strict ascending order:*

$$\mathbf{TraceA} := (\mathbf{Time} \times \mathbf{Action})^*,$$
$$\mathbf{TraceO} := (\mathbf{Time} \times \mathbf{Observation})^*.$$

We use α to range over action traces and θ to range over observation traces. We use $\mathbf{TraceA}^t, \mathbf{TraceO}^t$ to refer to the end-time-restricted traces as usual.

A tactic can now be defined very similarly to its resource-less version:

[2] The handles in the list can be new handles, which are then created. In practice, a more robust method, with explicit resource creation, would probably be preferred.

Definition 5.2.7. *A resource-aware tactic is a function:*

$$\tau \in \mathbf{Tactic} := \prod_{t \in \mathbf{Time}} (\mathbf{TraceO}^t \to \mathbf{TraceA}^t),$$

satisfying a monotonicity condition:

$$\forall \theta, \theta' \in \mathbf{TraceO}^t.\ \left(t' \leq t \wedge \theta_{\leq t'} = \theta'_{\leq t'}\right) \Rightarrow \tau(t)(\theta)_{\leq t'} = \tau(t)(\theta')_{\leq t'}.$$

Compared to the resource-less tactic, the latency requirement of the resource-aware tactic is built into the resource system, and is therefore not part of the definition. This preliminary account ends with this definition of contracts and tactics; in the next section we discuss how to extend the work.

5.3 Related and future work

In this section we mention related and future work.

5.3.1 Related work

Amongst the different process calculi, especially the π-calculus [65] is related to the resource-aware extension proposed here. The π-calculus allows the sending of channel endpoints across channels, much like we allow resources to work as addresses and be passed around. The notion of channel is, however, different from our usage of resources for communication, in that a resource transmission in our model is not contingent upon whether the recipient is ready or willing to receive the message.

The Actor model [3, 39] is related to our resource-aware model, both in that it allows addresses to be transferred between actors, but also because it employs message passing between actors as its main communication action, similar to a continuation-passing style. The main difference here is that new actors are routinely created throughout a computation, whereas we consider the set of agents fixed, or at least only slowly evolving.

Using games to model resource interactions is also studied in traditional game-theoretic work. In particular, coalition resource games [25] are similar, in the way that agents spend resources to gain other resources, eventually obtaining some goal. Related is also work on finding equilibria in resource games, where harvesters can use up a limited resource [74]. Our work focuses on modelling interaction scenarios, where the focus is on the interactions and not the result of the interactions, but we view this game-theoretic work as a valuable tool for analysing tactics and contracts.

5.3.2 Future work

The material presented here is work-in-progress, so naturally there are several directions for future work:

Finalise model. The model should be finalised, including account for conformance of tactics with respect to contracts. This includes indirectly formalising the 'contract' of the resource manager. Specifically, how to go from the action trace to the event trace. Similarly, how to employ several agents, should be considered. With several agents it is then natural to consider a compositionality theorem similar to Thm. 3.3.20. As long as conformance, is specified in such a way that all events that are not directly generated from the tactic's orders are taken into account, the compositionality theorem could probably be proven in a similar way.

Certification framework. The next natural step is to consider how to extend the certification framework. The first step is a resource-aware automaton model; we do not expect too many problems with such a model. To extend the verification-time monitoring paradigm to the resource setting, we can exploit that the behaviour of the resource manager is fixed and independent of the tactic. Therefore it suffices to consider only the actions and observations, which is just like the communication model (where formalised contracts also monitor actions and observations, not actual communication traces).

Resource-based pay-offs. An interesting idea is to consider, whether it is possible to eliminate final contract pay-offs in favour of normal resource transfers. One promising idea here is to allow each ongoing contract to hold a collection of *escrowed* resources by itself (much like the *pot* in a poker game), that participants have committed, but not yet transferred to the final recipients. When the contract stops (either normally or due to a violation), the escrowed resources are distributed amongst the participants as specified by the contract. However, with no way of truly breaching such a contract, the notion of conformance and composition of strategies would need to be suitably refined.

Part II

Aspects of focusing

Focusing and certification

An important question in a certification-based methodology is how to represent the formal proof of program safety. In the context of proof-carrying code, the most commonly used frameworks include the Edinburgh Logical Framework (LF) [35], and the Calculus of Inductive Constructions (CiC) [77]. Having already successfully created a model proof-carrying code infrastructure using a metalogical approach based on LF and the Twelf implementation [81] as part of a previous project [36], we wished to investigate a recently proposed framework by Nigam and Miller [73]. In their work, they show how to use a *focused* proof system for classical linear logic to 'host' a wide range of different proof systems for both classical and intuitionistic logic. Focused proof systems allow a great deal of control over the possible shapes of proofs, without affecting provability. This control can be exploited to ensure that open derivations in the hosted system are in bijective correspondence with the open derivations in the host system.

The results of Nigam and Miller are tightly coupled with classical linear logic: exponentials are used heavily in their encodings, and several proofs use classical equivalences. As discussed in Sect. 4.4.1, we expect that a constructive or intuitionistic logic could give a more logically well-founded Floyd-Hoare logic. We therefore investigated, whether the results of Nigam and Miller could be transferred to a focused proof system for intuitionistic logic instead. The result of the investigation turned out positive, as we show in Chapter 6; we can therefore conclude that focusing is the essential property that makes this approach work.

While investigating focused proof systems, we noticed a redundancy in the proofs. All the focused systems we saw allowed contraction exactly for the formula chosen for focus. This introduces redundancy in proofs, as the same formula can be chosen again and again without adding new information. Removing this use of contraction is not trivial, however, because the completeness of the proof systems depends on being able to refocus in order to pick a different path through a formula. As another piece of proof-theoretical work, in Chapter 7 we show how to remove the use of contraction from the propositional fragment of the focused proof system for classical logic, LKF, while still retaining a complete system. The key insight being that only disjunction can break completeness, and therefore special care has to be taken in the disjunctive case.

The material in this part is structured in two independent chapters. The first chapter contains a reformatted and lightly edited version of the tech-report [37], which shows how to use a focused proof system for intuitionistic logic as a host for other proof systems. The second chapter contains a manuscript for a paper detailing the proof system, $\mathrm{LKF}_{\mathrm{CF}}$, which is a focused proof system without contraction.

Chapter 6

Using LJF as a framework for proof systems

Abstract

In this work, we show how to use the focused proof system LJF for intuitionistic logic as a framework for encoding several different proof systems for both intuitionistic and classical logic. The proof systems are encoded at a strong level of adequacy, namely the level of (open) derivations. Furthermore, we show how to prove *relative completeness* between the different systems, i.e., that the systems prove the same formulas. The proofs of relative completeness exploit the encodings to give, in most cases, fairly simple proofs. This work is heavily based on the recent work by Nigam and Miller, which uses the focused linear logic system LLF to encode the same proof systems as we do. Our work shows that the features of linear logic are not needed for the full adequacy result, and furthermore, we show that even though the encodings in LLF are more generic and streamlined, the encodings in LJF sometimes give simpler, more natural encodings and easier proofs.

6.1 Introduction

In recent work by Nigam and Miller [73], they propose to use a *focused* proof system for classical linear logic as a logical framework for representing different object-level proof systems.

In a focused proof system the connectives are divided into two groups. The *negative* or *asynchronous* connectives are usually associated with invertible rules, whereas the *positive* or *synchronous* connectives in general do not need to have invertible rules. Formulas are assigned to the groups according to their top most connective. Atoms are arbitrarily assigned to either the negative or positive group. We say that a formula or connective assigned to the negative (resp. positive) group has negative (resp. positive) *polarity*.

Derivations in a focused proof system are divided into two *phases* corresponding to the groups above. The negative or asynchronous phase applies all invertible rules to the sequent. The positive or synchronous phase *focuses* on a particular formula and then keeps applying the remaining rules, until the formula becomes negative or an atom. As all rules applied in the negative phase are invertible, we can exploit the 'don't-care' nondeterminism of the invertible rules and see the entire negative phase as one single *macro-rule*, which applies all negative rules in one go. The macro-rule concept is critical for the strong encodings shown later. The positive phase may contain 'don't-know'

nondeterminism (e.g. which branch to prove in a disjunction), but because it is fixed by the focused formula, we can also see the synchronous phase as consisting of one macro-rule, although there may be several applicable macro-rules.

Focusing was first developed for classical linear logic by Andreoli [5], and later adopted into various proof systems [26, 80]. An important result is that, regardless of how polarity is assigned to atoms, the system is complete with respect to classical linear logic.

In previously mentioned work by Nigam and Miller they show how to encode a long range of different intuitionistic and classical proof systems using Andreoli's focused system LLF. The encoded systems are strongly related to the object systems, as there is a provable bijection between the *open derivations* in LLF and open derivations in the object system. To establish the bijection, the assignment of polarities to atoms plays a big role, as the polarity can be used to enforce either forward or backward reasoning.

In this work we show that encoding object-level systems using a focused proof system is not dependent on the linear aspects of LLF, and we show how to encode the same systems, and get full completeness of open derivations, using the focused system LJF [52] for intuitionistic logic. Furthermore, we compare the LJF encodings to the LLF encodings and comment on the differences.

In addition to encoding different systems, Nigam and Miller use the meta-logic encodings to prove properties of the object-logic systems. In their paper they focus on *relative completeness*, which is completeness with respect to provability. They show how to relate the different proof systems back to a generic set of rules with the inclusion or exclusion of specific structural rules.

In this work we also look at relative completeness, but instead of relating back to a generic set of rules, we relate the different systems to each other. The reason is that the encodings of intuitionistic and classical systems are rather different when encoded with an intuitionistic meta-logic. But in the end we get the same results, and in several places the proofs seems easier to do.

A disclaimer: the work in this report is based on the article by Nigam and Miller [73], leading to a lot of similarities; for instance, all the systems we have encoded, are found in their article. We have made some changes to the systems on the syntactic level and a few other changes; these changes will be described in the later sections. Furthermore, most of the references here are from the article as well; they are included here as convenience for the reader.

6.2 Focused intuitionistic logic

As a framework for hosting different object-level systems we use the focused system LJF [52]. The choice of LJF is somewhat arbitrary, and we think that a similar focused system e.g. the system by Pfenning [80] could be used as well. An important criterion of the system is the possibility of assigning arbitrary polarities to atoms and still retaining completeness with regards to unfocused intuitionistic logic.

The rest of this section presents LJF. For a more complete description, the reader is referred to the original paper on LJF.

LJF in a many-sorted logic, we use S to range over the sorts, the sorts consists of primitive and functional sorts:

$$S ::= s \mid S_1 \to S_2.$$

where s ranges over primitive sorts. The formulas in LJF are given by the following grammar:

$$\begin{aligned} A ::= \quad & M \mid \mathbf{false} \mid \mathbf{true} \mid \\ & A_1 \supset A_2 \mid A_1 \wedge^- A_2 \mid A_1 \wedge^+ A_2 \mid A_1 \vee A_2 \mid \\ & \forall x{:}S.\, A \mid \exists x{:}S.\, A \end{aligned}$$

Atoms, M, are predicates applied to terms, both defined below:

$$\begin{aligned} M &::= p\ t_1\ \ldots\ t_n \\ t &::= c \mid x \mid \lambda x{:}S.\, t \mid t_1\ t_2 \mid a \end{aligned}$$

where p is a sorted predicate; the sorts for a given predicate can be written as a tuple $(S_1, \ldots, S_n)$ or by using o to refer to meta-level predicates: $p : S_1 \to \ldots S_n \to o$. c is a sorted constant $(c : S)$, and a is a parameter. We note that there is no negation ($\neg A$ can be defined as $A \supset \mathbf{false}$), and that there are two conjunctions. The two conjunctions are equivalent in terms of intuitionistic provability, but have different focused proofs, as one is positive and the other is negative. We refer to the original paper for more information. A *signature* for LJF consists of the primitive sorts, s; the predicates, p, and their sorting; and the constants, c, and their sorts.

Predicate symbols are arbitrarily assigned either a positive or a negative polarity, and the polarity of an atom is defined as the polarity of its predicate. Positive formulas are given by the following grammar:

$$P ::= M_{\mathrm{P}} \mid \mathbf{false} \mid \mathbf{true} \mid A_1 \wedge^+ A_2 \mid A_1 \vee A_2 \mid \exists x A$$

where M_{P} is a positive atom (an atom with positive predicate). Negative formulas are given by the following grammar:

$$N ::= M_{\mathrm{N}} \mid A_1 \supset A_2 \mid A_1 \wedge^- A_2 \mid \forall x A$$

where M_{N} is a negative atom (an atom with negative predicate).

The sequents in LJF have one of the following forms:

1. $[\Gamma], \Theta \longrightarrow \mathcal{R}$ is an unfocused sequent. Γ and Θ are multisets and Γ only contains negative formulas and atoms. $\mathcal{R}$ is either a formula R or a bracketed formula $[R]$. The formulas inside brackets are saved for the synchronous phase. Both Γ, Θ, and $\mathcal{R}$, can contain parameters, but not free variables.

2. $[\Gamma] \xrightarrow{A} [R]$ is a left-focused sequent with focus on A.

3. $[\Gamma] \ -_A\!\!\rightarrow$ is a right-focused sequent with focus on A.[1]

The rules for LJF are given in Figure 6.1. The initial rules end the proof, if the atom has the correct polarity. The decision rules pick a formula from either the left side or the right side and continue with focus on that formula. If a formula is positive in the left-focused phase or negative in the right-focused phase, focus is lost by one of the reaction rules. Furthermore, in the asynchronous phase, synchronous formulas are inserted into the bracketed context by one of the remaining reaction rules. The

[1] The focused formula should be written under the arrow, but for typographic reasons it is written inside the arrow.

Initial Rules

$$\frac{}{[\Gamma] \xrightarrow{M_{\mathrm{N}}} [M_{\mathrm{N}}]}\ \mathsf{I_L} \qquad \frac{}{[M_{\mathrm{P}}, \Gamma] -_{M_{\mathrm{P}}}\!\!\rightarrow}\ \mathsf{I_R}$$

Decision Rules

$$\frac{[N, \Gamma] \xrightarrow{N} [R]}{[N, \Gamma] \longrightarrow [R]}\ \mathsf{L_F} \qquad \frac{[\Gamma] -_P\!\!\rightarrow}{[\Gamma] \longrightarrow [P]}\ \mathsf{R_F}$$

Reaction Rules

$$\frac{[\Gamma], P \longrightarrow [R]}{[\Gamma] \xrightarrow{P} [R]}\ \mathsf{R_L} \qquad \frac{[\Gamma] \longrightarrow N}{[\Gamma] -_N\!\!\rightarrow}\ \mathsf{R_R}$$

$$\frac{[C, \Gamma], \Theta \longrightarrow \mathcal{R}}{[\Gamma], \Theta, C \longrightarrow \mathcal{R}}\ []_L \qquad \frac{[\Gamma], \Theta \longrightarrow [D]}{[\Gamma], \Theta \longrightarrow D}\ []_R$$

Introduction Rules

$$\frac{}{[\Gamma], \Theta, \mathbf{false} \longrightarrow \mathcal{R}}\ \mathbf{false_L}$$

$$\frac{[\Gamma], \Theta \longrightarrow \mathcal{R}}{[\Gamma], \Theta, \mathbf{true} \longrightarrow \mathcal{R}}\ \mathbf{true_L} \qquad \frac{}{[\Gamma] -_{\mathbf{true}}\!\!\rightarrow}\ \mathbf{true_R}$$

$$\frac{[\Gamma] \xrightarrow{A_i} [R]}{[\Gamma] \xrightarrow{A_1 \wedge^- A_2} [R]}\ \wedge^-_{\mathsf{L}} \qquad \frac{[\Gamma], \Theta \longrightarrow A_1 \qquad [\Gamma], \Theta \longrightarrow A_2}{[\Gamma], \Theta \longrightarrow A_1 \wedge^- A_2}\ \wedge^-_{\mathsf{R}}$$

$$\frac{[\Gamma], \Theta, A_1, A_2 \longrightarrow \mathcal{R}}{[\Gamma], \Theta, A_1 \wedge^+ A_2 \longrightarrow \mathcal{R}}\ \wedge^+_{\mathsf{L}} \qquad \frac{[\Gamma] -_{A_1}\!\!\rightarrow \qquad [\Gamma] -_{A_2}\!\!\rightarrow}{[\Gamma] -_{A_1 \wedge^+ A_2}\!\!\rightarrow}\ \wedge^+_{\mathsf{R}}$$

$$\frac{[\Gamma], \Theta, A_1 \longrightarrow \mathcal{R} \qquad [\Gamma], \Theta, A_2 \longrightarrow \mathcal{R}}{[\Gamma], \Theta, A_1 \vee A_2 \longrightarrow \mathcal{R}}\ \vee_{\mathsf{L}} \qquad \frac{[\Gamma] -_{A_i}\!\!\rightarrow}{[\Gamma] -_{A_1 \vee A_2}\!\!\rightarrow}\ \vee_{\mathsf{R}}$$

$$\frac{[\Gamma] -_{A_1}\!\!\rightarrow \qquad [\Gamma] \xrightarrow{A_2} [R]}{[\Gamma] \xrightarrow{A_1 \supset A_2} [R]}\ \supset_{\mathsf{L}} \qquad \frac{[\Gamma], \Theta, A_1 \longrightarrow A_2}{[\Gamma], \Theta \longrightarrow A_1 \supset A_2}\ \supset_{\mathsf{R}}$$

$$\frac{[\Gamma], \Theta, A[a/x] \longrightarrow \mathcal{R}}{[\Gamma], \Theta, \exists x A \longrightarrow \mathcal{R}}\ \exists_{\mathsf{L}}{}^a \qquad \frac{[\Gamma] -_{A[t/x]}\!\!\rightarrow}{[\Gamma] -_{\exists x A}\!\!\rightarrow}\ \exists_{\mathsf{R}}$$

$$\frac{[\Gamma] \xrightarrow{A[t/x]} [R]}{[\Gamma] \xrightarrow{\forall x A} [R]}\ \forall_{\mathsf{L}} \qquad \frac{[\Gamma], \Theta \longrightarrow A[a/x]}{[\Gamma], \Theta \longrightarrow \forall x A}\ \forall_{\mathsf{R}}{}^a$$

Figure 6.1: The proof system LJF [52]. M_{P} is a positive atom, M_{N} is a negative atom. P is positive, N is negative. C is negative or an atom, D is positive or an atom. a is not free in Γ, Θ or $\mathcal{R}$. $i \in \{1, 2\}$.

introduction rules decompose a specific formula. For the synchronous phase the focused formula is decomposed, and for the asynchronous phase one of the asynchronous formulas is decomposed.

As mentioned in the introduction, we can see the two phases as applying macro-rules, which takes an entire phase in one go. Consider for example the possible derivations of $[\Gamma] -_{A_1 \vee (A_2 \wedge^+ A_3)}\!\!\rightarrow$ where A_1, A_2, A_3 are negative:

$$\dfrac{\dfrac{[\Gamma] \longrightarrow A_1}{[\Gamma] -_{A_1}\!\!\rightarrow}}{[\Gamma] -_{A_1 \vee (A_2 \wedge^+ A_3)}\!\!\rightarrow} \qquad \dfrac{\dfrac{[\Gamma] \longrightarrow A_2}{[\Gamma] -_{A_2}\!\!\rightarrow} \quad \dfrac{[\Gamma] \longrightarrow A_3}{[\Gamma] -_{A_3}\!\!\rightarrow}}{\dfrac{[\Gamma] -_{A_2 \wedge^+ A_3}\!\!\rightarrow}{[\Gamma] -_{A_1 \vee (A_2 \wedge^+ A_3)}\!\!\rightarrow}}$$

These can be seen as applications of one of two different macro-rules:

$$\dfrac{[\Gamma] \longrightarrow A_1}{[\Gamma] -_{A_1 \vee (A_2 \wedge^+ A_3)}\!\!\rightarrow} \qquad \dfrac{[\Gamma] \longrightarrow A_2 \qquad [\Gamma] \longrightarrow A_3}{[\Gamma] -_{A_1 \vee (A_2 \wedge^+ A_3)}\!\!\rightarrow} \quad (*)$$

Note that we use double lines when we merge several rule applications into one.

As an example for the asynchronous phase, consider the possible derivations of $[\Gamma], (A_1 \vee A_2) \wedge^+ (A_3 \wedge^+ A_4) \longrightarrow [R]$ where the A_i's are negative:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\Gamma, A_1, A_3, A_4] \longrightarrow [R]}{\vdots}}{[\Gamma], A_1, A_3, A_4 \longrightarrow [R]}}{[\Gamma], A_1, A_3 \wedge^+ A_4 \longrightarrow [R]} \quad \dfrac{\dfrac{\dfrac{[\Gamma, A_2, A_3, A_4] \longrightarrow [R]}{\vdots}}{[\Gamma], A_2, A_3, A_4 \longrightarrow [R]}}{[\Gamma], A_2, A_3 \wedge^+ A_4 \longrightarrow [R]}}{\dfrac{[\Gamma], A_1 \vee A_2, A_3 \wedge^+ A_4 \longrightarrow [R]}{[\Gamma], (A_1 \vee A_2) \wedge^+ (A_3 \wedge^+ A_4) \longrightarrow [R]}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\Gamma, A_1, A_3, A_4] \longrightarrow [R]}{\vdots}}{[\Gamma], A_1, A_3, A_4 \longrightarrow [R]} \quad \dfrac{\dfrac{[\Gamma, A_2, A_3, A_4] \longrightarrow [R]}{\vdots}}{[\Gamma], A_2, A_3, A_4 \longrightarrow [R]}}{[\Gamma], A_1 \vee A_2, A_3, A_4 \longrightarrow [R]}}{\dfrac{[\Gamma], A_1 \vee A_2, A_3 \wedge^+ A_4 \longrightarrow [R]}{[\Gamma], (A_1 \vee A_2) \wedge^+ (A_3 \wedge^+ A_4) \longrightarrow [R]}}$$

These combine into a single macro-rule for the asynchronous phase:

$$\dfrac{[\Gamma, A_1, A_3, A_4] \longrightarrow [R] \qquad [\Gamma, A_2, A_3, A_4] \longrightarrow [R]}{[\Gamma], (A_1 \vee A_2) \wedge^+ (A_3 \wedge^+ A_4) \longrightarrow [R]}$$

In the rest of this report, we consider two LJF derivations with the same macro-rule derivation as the same derivation. We view formulas like $A_1 \vee (A_2 \wedge^+ A_3)$ as a *synthetic connective* with the introduction rules given by (*). This identification of derivations is needed to get full completeness of open derivations for the different systems.

Like the original focused proof system for linear logic, LLF [5], assignment of polarity to atoms does not affect provability. Therefore, LJF is sound and complete with respect to intuitionistic logic. In the following and in the rest of this work $\vdash_{\mathrm{I}}$ will stand for *provability* in an unspecified intuitionistic system with the domain of discourse (the atom structure) as our system. We write A° (resp. Γ°) for the unpolerised version of A (resp. Γ), ($\wedge^+, \wedge^-$ is replaced with $\wedge$).

Theorem 6.2.1. *For all LJF formulas A and for any assignment of polarities to atoms:*

$$\vdash_{\mathrm{I}} A^{\circ} \quad \textit{if and only if} \quad [\,] \longrightarrow A$$

Proof. The proof is available in the original article [52]. □

In the encodings of the proof systems, we will use the bracketed context to hold the encoded rules. The following corollary allows us to use the connection to intuitionistic logic for sequents with non-empty contexts.

Corollary 6.2.2. *If A is an atom or a positive formula, and Γ consists of atoms and negative formulas, then:*

$$\Gamma^{\circ} \vdash_{\mathrm{I}} A^{\circ} \quad \textit{if and only if} \quad [\Gamma] \longrightarrow [A]$$

Proof. If $\Gamma = \{A_1, \ldots, A_n\}$ then by invertability of implication in intuitionistic logic we have that:

$$\Gamma^{\circ} \vdash_{\mathrm{I}} A^{\circ} \quad \text{if and only if} \quad \vdash_{\mathrm{I}} {A_1}^{\circ} \supset ({A_2}^{\circ} \supset \cdots ({A_n}^{\circ} \supset A^{\circ}) \cdots)$$

which by Theorem 6.2.1 means that:

$$\Gamma^{\circ} \vdash_{\mathrm{I}} A^{\circ} \quad \text{if and only if} \quad [\,] \longrightarrow A_1 \supset (A_2 \supset \cdots (A_n \supset A) \cdots)$$

which means that we have:

$$\Gamma^{\circ} \vdash_{\mathrm{I}} A^{\circ} \quad \text{if and only if} \quad [\,], \Gamma \longrightarrow A$$

and because A is an atom or a positive formula, and Γ consists of atoms and negative formulas then:

$$\Gamma^{\circ} \vdash_{\mathrm{I}} A^{\circ} \quad \text{if and only if} \quad [\Gamma] \longrightarrow [A]$$

□

6.3 Encoding in LJF

The encodings of object-level formulas inside LJF atoms are the same as in Nigam and Miller's paper, and we shortly give an overview here.

Our object-level system will be variations of a standard single sorted first-order logic. There are two object-level syntactic categories: object-level terms, and object-level formulas. The object-level formulas uses standard syntax (negation is only used in some of the object systems):

$$A ::= \bot \mid \top \mid A_1 \Rightarrow A_2 \mid A_1 \vee A_2 \mid A_1 \wedge A_2 \mid \exists x A \mid \forall x A \; (\mid \neg A).$$

To encode a proof system for such a logic using LJF, we need to specify the primitive sorts, s, the predicates, p, and the constants c.

The primitive sorts, s, consists of a sort for object-level terms, i, and one for object-level formulas, *form*. The constants are the object level constructors, so c includes e.g. $\wedge_{\mathrm{obj}}$ and $\forall_{\mathrm{obj}}$ with the types:

$$\wedge_{\mathrm{obj}} : \mathit{form} \to \mathit{form} \to \mathit{form},$$
$$\forall_{\mathrm{obj}} : (i \to \mathit{form}) \to \mathit{form}.$$

The predicates, p, contain two unary predicates: $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$, both over terms of sort *form* (i.e., $\lfloor\cdot\rfloor : \mathit{form} \to o$). The first one is used to encode hypotheses in the object-level sequent, and the second one is used to encode conclusions.

As an example consider an object-logic with zero, addition and equality, that is

$$\begin{aligned} &\text{zero} : i, \\ &\text{plus} : i \to i \to i, \\ &\quad \text{eq} : i \to i \to \mathit{form}. \end{aligned}$$

The object-level hypothesis formula:

$$\forall x.\ x + 0 = x,$$

can be represented as a meta-level formula as follows:

$$\lfloor \forall_{\text{obj}}(\lambda x : i.\ \text{eq}\ (\text{plus}\ x\ \text{zero})\ x) \rfloor$$

We omit the subscripted 'obj' in concrete cases. Both predicates can be applied to sets of object-level formulas generating sets of meta-level formulas in a straightforward manner. In general, an intuitionistic sequent $\Gamma \vdash A$ will be encoded as the sequent $[\lfloor\Gamma\rfloor, \mathcal{L}] \longrightarrow \lceil A\rceil$, and the classical sequent $\Gamma \vdash \Delta$ as $[\lfloor\Gamma\rfloor, \lceil\Delta\rceil, \mathcal{L}] \longrightarrow \mathbf{false}$. $\mathcal{L}$ is a system dependent encoding of the proof rules. We will give more details on the encodings for each system in the next sections.

6.3.1 Sequent calculus

In this section we consider a proof system for intuitionistic sequent calculus (LJ) and classical sequent calculus (LK) [34]. These systems are given by the rules in Figure 6.2 and 6.3. The system LJ is a little different from the system LJ in the Nigam and Miller paper. We use a version of LJ where there are no empty right sides of the sequents. We use this version to give a simpler encoding, but we will return to the version with empty right sides in Section 6.3.4.

For the intuitionistic sequent calculus LJ, we use a straightforward encoding. All propositions on the left of the LJ sequent are encoded using the $\lfloor\cdot\rfloor$ predicate to the left side of the LJF sequent. The conclusion is encoded on the right side of the LJF sequent using the $\lceil\cdot\rceil$ predicate.

For the left rules we see that they can only be applied, if there exists a formula with the given connective. This means that we must not release focus from atoms in the right focused sequent, and therefore $\lfloor\cdot\rfloor$ atoms have to be positive. The opposite is true for the right rules and therefore $\lceil\cdot\rceil$ atoms have to be negative.

The proof rules of the object logic (LJ or LK) are encoded as a collection of named LJF formulas. For example we represent the implication left introduction rule as the formula:

$$(\Rightarrow_{\mathsf{L}}) \quad \forall A{:}\mathit{form}.\, \forall B{:}\mathit{form}.\, \lfloor A \Rightarrow B\rfloor \supset (\lceil A\rceil \supset \lfloor B\rfloor)$$

For simplicity, we will use a shorthand when writing these formulas and omit the leading universals. Therefore we will write the above formula as:

$$\lfloor A \Rightarrow B\rfloor \supset (\lceil A\rceil \supset \lfloor B\rfloor).$$

Similarly, x will range over object-level terms, so we write:

$$(\forall_{\mathsf{R}}) \quad \lceil \forall x A\rceil \subset \forall x \lceil A\rceil$$

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}} A_1 \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{LJ}} C}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}} C} \Rightarrow_{\mathsf{L}} \quad \frac{\Gamma, A_1 \vdash_{\mathrm{LJ}} A_2}{\Gamma \vdash_{\mathrm{LJ}} A_1 \Rightarrow A_2} \Rightarrow_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \wedge A_2, A_i \vdash_{\mathrm{LJ}} C}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{LJ}} C} \wedge_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}} A_1 \qquad \Gamma \vdash_{\mathrm{LJ}} A_2}{\Gamma \vdash_{\mathrm{LJ}} A_1 \wedge A_2} \wedge_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \vee A_2, A_1 \vdash_{\mathrm{LJ}} C \qquad \Gamma, A_1 \vee A_2, A_2 \vdash_{\mathrm{LJ}} C}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{LJ}} C} \vee_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}} A_i}{\Gamma \vdash_{\mathrm{LJ}} A_1 \vee A_2} \vee_{\mathsf{R}}$$

$$\frac{\Gamma, \forall x A, A[t/x] \vdash_{\mathrm{LJ}} C}{\Gamma, \forall x A \vdash_{\mathrm{LJ}} C} \forall_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}} A[a/x]}{\Gamma \vdash_{\mathrm{LJ}} \forall x A} \forall_{\mathsf{R}}{}^{a}$$

$$\frac{\Gamma, \exists x A, A[a/x] \vdash_{\mathrm{LJ}} C}{\Gamma, \exists x A \vdash_{\mathrm{LJ}} C} \exists_{\mathsf{L}}{}^{a} \quad \frac{\Gamma \vdash_{\mathrm{LJ}} A[t/x]}{\Gamma \vdash_{\mathrm{LJ}} \exists x A} \exists_{\mathsf{R}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{LJ}} A} \mathsf{I} \quad \frac{\Gamma \vdash_{\mathrm{LJ}} A \qquad \Gamma, A \vdash_{\mathrm{LJ}} C}{\Gamma \vdash_{\mathrm{LJ}} C} \mathsf{Cut}$$

$$\frac{}{\Gamma, \bot \vdash_{\mathrm{LJ}} C} \bot_{\mathsf{L}} \quad \frac{}{\Gamma \vdash_{\mathrm{LJ}} \top} \top_{\mathsf{R}}$$

Figure 6.2: The proof system LJ. a is not free in Γ or C. $i \in \{1, 2\}$.

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LK}} A_1, \Delta \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{LK}} \Delta}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LK}} \Delta} \Rightarrow_{\mathsf{L}}$$

$$\frac{\Gamma, A_1 \vdash_{\mathrm{LK}} A_1 \Rightarrow A_2, A_2, \Delta}{\Gamma \vdash_{\mathrm{LK}} A_1 \Rightarrow A_2, \Delta} \Rightarrow_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \wedge A_2, A_i \vdash_{\mathrm{LK}} \Delta}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{LK}} \Delta} \wedge_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LK}} A_1 \wedge A_2, A_1, \Delta \qquad \Gamma \vdash_{\mathrm{LK}} A_1 \wedge A_2, A_2, \Delta}{\Gamma \vdash_{\mathrm{LK}} A_1 \wedge A_2, \Delta} \wedge_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \vee A_2, A_1 \vdash_{\mathrm{LK}} \Delta \qquad \Gamma, A_1 \vee A_2, A_2 \vdash_{\mathrm{LK}} \Delta}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{LK}} \Delta} \vee_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LK}} A_i, A_1 \vee A_2, \Delta}{\Gamma \vdash_{\mathrm{LK}} A_1 \vee A_2, \Delta} \vee_{\mathsf{R}}$$

$$\frac{\Gamma, \forall x A, A[t/x] \vdash_{\mathrm{LK}} \Delta}{\Gamma, \forall x A \vdash_{\mathrm{LK}} \Delta} \forall_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LK}} A[a/x], \forall x A, \Delta}{\Gamma \vdash_{\mathrm{LK}} \forall x A, \Delta} \forall_{\mathsf{R}}{}^{a}$$

$$\frac{\Gamma, \exists x A, A[a/x] \vdash_{\mathrm{LK}} \Delta}{\Gamma, \exists x A \vdash_{\mathrm{LK}} \Delta} \exists_{\mathsf{L}}{}^{a} \quad \frac{\Gamma \vdash_{\mathrm{LK}} A[t/x], \exists x A, \Delta}{\Gamma \vdash_{\mathrm{LK}} \exists x A, \Delta} \exists_{\mathsf{R}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{LK}} A, \Delta} \mathsf{I} \quad \frac{\Gamma \vdash_{\mathrm{LK}} A, \Delta \qquad \Gamma, A \vdash_{\mathrm{LK}} \Delta}{\Gamma \vdash_{\mathrm{LK}} \Delta} \mathsf{Cut}$$

$$\frac{}{\Gamma, \bot \vdash_{\mathrm{LK}} \Delta} \bot_{\mathsf{L}} \quad \frac{}{\Gamma \vdash_{\mathrm{LK}} \top, \Delta} \top_{\mathsf{R}}$$

Figure 6.3: The proof system LK. a is not free in Γ or Δ. $i \in \{1, 2\}$.

$(\Rightarrow_{\mathsf{L}})$	$\lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \supset \lfloor B \rfloor)$	$(\Rightarrow_{\mathsf{R}})$	$\lceil A \Rightarrow B \rceil \subset (\lfloor A \rfloor \supset \lceil B \rceil)$
$(\wedge_{\mathsf{L}})$	$\lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{-} \lfloor B \rfloor)$	$(\wedge_{\mathsf{R}})$	$\lceil A \wedge B \rceil \subset (\lceil A \rceil \wedge^{-} \lceil B \rceil)$
$(\vee_{\mathsf{L}})$	$\lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \vee \lfloor B \rfloor)$	$(\vee_{\mathsf{R}})$	$\lceil A \vee B \rceil \subset (\lceil A \rceil \vee \lceil B \rceil)$
$(\forall_{\mathsf{L}})$	$\lfloor \forall x A \rfloor \supset \forall x \lfloor A \rfloor$	$(\forall_{\mathsf{R}})$	$\lceil \forall x A \rceil \subset \forall x \lceil A \rceil$
$(\exists_{\mathsf{L}})$	$\lfloor \exists x A \rfloor \supset \exists x \lfloor A \rfloor$	$(\exists_{\mathsf{R}})$	$\lceil \exists x A \rceil \subset \exists x \lceil A \rceil$
(I)	$\lfloor A \rfloor \supset \lceil A \rceil$	(Cut)	$\lfloor A \rfloor \subset \lceil A \rceil$
$(\bot_{\mathsf{L}})$	$\lfloor \bot \rfloor \supset \mathbf{false}$	$(\top_{\mathsf{R}})$	$\lceil \top \rceil \subset \mathbf{true}$

Figure 6.4: Intuitionistic sequent calculus, $\mathcal{L}_{\mathrm{LJ}}$.

$(\Rightarrow_{\mathsf{L}})$	$\lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \vee \lfloor B \rfloor)$	$(\Rightarrow_{\mathsf{R}})$	$\lceil A \Rightarrow B \rceil \supset (\lfloor A \rfloor \wedge^{+} \lceil B \rceil)$
$(\wedge_{\mathsf{L}})$	$\lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{-} \lfloor B \rfloor)$	$(\wedge_{\mathsf{R}})$	$\lceil A \wedge B \rceil \supset (\lceil A \rceil \vee \lceil B \rceil)$
$(\vee_{\mathsf{L}})$	$\lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \vee \lfloor B \rfloor)$	$(\vee_{\mathsf{R}})$	$\lceil A \vee B \rceil \supset (\lceil A \rceil \wedge^{-} \lceil B \rceil)$
$(\forall_{\mathsf{L}})$	$\lfloor \forall x A \rfloor \supset \forall x \lfloor A \rfloor$	$(\forall_{\mathsf{R}})$	$\lceil \forall x A \rceil \supset \exists x \lceil A \rceil$
$(\exists_{\mathsf{L}})$	$\lfloor \exists x A \rfloor \supset \exists x \lfloor A \rfloor$	$(\exists_{\mathsf{R}})$	$\lceil \exists x A \rceil \supset \forall x \lceil A \rceil$
(I)	$(\lfloor A \rfloor \wedge^{+} \lceil A \rceil) \supset \mathbf{false}$	(Cut)	$\lfloor A \rfloor \vee \lceil A \rceil$
$(\bot_{\mathsf{L}})$	$\lfloor \bot \rfloor \supset \mathbf{false}$	$(\top_{\mathsf{R}})$	$\lceil \top \rceil \supset \mathbf{false}$

Figure 6.5: Classical sequent calculus, $\mathcal{L}_{\mathrm{LK}}$.

instead of

$$\forall A{:}i \to \mathit{form}.\, (\forall x{:}i.\, \lceil A \rceil) \supset \lfloor \forall x A \rfloor$$

The encoding of the rules for LJ is given in Figure 6.4.

We use $\mathcal{L}_{\mathrm{LJ}}$ to stand for the set of all encoded rules from Figure 6.4; similarly we will later use $\mathcal{L}_{\mathrm{X}}$ to stand for the set of all encoded rules for some system, X. Putting all this together, we get that the sequent $\Gamma \vdash_{\mathrm{LJ}} C$ is encoded as $[\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$.

The encodings of the different rules are very straightforward, as the object-level connectives are encoded by the same meta-level connectives. For the left rules, the principal formula implies the rest of the formulas, and for the right rules, the principal formula is implied by the rest of the formulas.

The encoding encodes LJ on the full level of completeness between (open) derivations, as shown by the following proposition:

Proposition 6.3.1. *Let $\Gamma \cup \{C\}$ be a set of LJ formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{LJ}} C \quad \textit{and} \quad [\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

Proof. This is the first proof in a series of rather similar proofs of full completeness. Formally, the proof goes by induction on the derivation in both directions but we consider both directions in one go. An observation is that the macro-rules in LJF corresponds exactly to focusing on one of the different formulas from $\mathcal{L}_{\mathrm{LJ}}$. What we then have to show is that each rule in LJ corresponds exactly to focusing on the corresponding formula in $\mathcal{L}_{\mathrm{LJ}}$.

So to prove completeness, one goes through all the rules and check correspondence. In each of these proofs, we will show a couple of cases and leave the rest for the reader.

$(\Rightarrow_{\mathsf{L}})$:

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}} A_1 \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{LJ}} C}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}} C} \Rightarrow_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor A_1 \Rightarrow A_2 \rfloor}\!\!\rightarrow} \mathsf{I_R} \quad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \rceil}\!\!\rightarrow} \mathsf{R_R}, []_{\mathsf{R}} \quad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [\lceil C \rceil]}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor} [\lceil C \rceil]} \mathsf{R_L}, []_{\mathsf{L}}}{} 2 \times \supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \supset \lfloor A_2 \rfloor)} [\lceil C \rceil]}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}$$

($\Rightarrow_{\mathsf{R}}$):

$$\dfrac{\Gamma, A_1 \vdash_{\mathrm{LJ}} A_2}{\Gamma \vdash_{\mathrm{LJ}} A_1 \Rightarrow A_2} \Rightarrow_{\mathsf{R}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil} [\lceil A_1 \Rightarrow A_2 \rceil]} \mathsf{I_L} \quad \dfrac{\dfrac{[\mathcal{K}, \lfloor A_1 \rfloor] \longrightarrow [\lceil A_2 \rceil]}{[\mathcal{K}], \lfloor A_1 \rfloor \longrightarrow \lceil A_2 \rceil} []_{\mathsf{L}}, []_{\mathsf{R}}}{[\mathcal{K}] -_{\lfloor A_1 \rfloor \supset \lceil A_2 \rceil}\!\!\rightarrow} \mathsf{R_R}, \supset_{\mathsf{R}}}{} \supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil \subset (\lfloor A_1 \rfloor \supset \lceil A_2 \rceil)} [\lceil A_1 \Rightarrow A_2 \rceil]}{[\mathcal{K}] \longrightarrow [\lceil A_1 \Rightarrow A_2 \rceil]} \mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}$$

($\forall_{\mathsf{L}}$):

$$\dfrac{\Gamma, \forall x A, A[t/x] \vdash_{\mathrm{LJ}} C}{\Gamma, \forall x A \vdash_{\mathrm{LJ}} C} \forall_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \forall x A \rfloor\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor \forall x A \rfloor}\!\!\rightarrow} \mathsf{I_R} \quad \dfrac{\dfrac{[\mathcal{K}, \lfloor A[t/x] \rfloor] \longrightarrow [\lceil C \rceil]}{[\mathcal{K}] \xrightarrow{\lfloor A[t/x] \rfloor} [\lceil C \rceil]} \mathsf{R_L}, []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\forall x \lfloor A \rfloor} [\lceil C \rceil]} \forall_{\mathsf{L}}}{} \supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor \forall x A \rfloor \supset \forall x \lfloor A \rfloor} [\lceil C \rceil]}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}, \forall_{\mathsf{L}}}$$

(I):

$$\dfrac{}{\Gamma, A \vdash_{\mathrm{LJ}} A} \mathsf{I}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A \rfloor\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor A \rfloor}\!\!\rightarrow} \mathsf{I_R} \quad \dfrac{}{[\mathcal{K}] \xrightarrow{\lceil A \rceil} [\lceil A \rceil]} \mathsf{I_L}}{} \supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A \rfloor \supset \lceil A \rceil} [\lceil A \rceil]}{[\mathcal{K}] \longrightarrow [\lceil A \rceil]} \mathsf{L_F}, \forall_{\mathsf{L}}}$$

($\bot_{\mathsf{L}}$):

$$\dfrac{}{\Gamma, \bot \vdash_{\mathrm{LJ}} C} \bot_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\text{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$)

$$\dfrac{\dfrac{}{[\mathcal{K}] -\!\!\!\rightarrow_{\lfloor \bot \rfloor}} \mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [\lceil C \rceil]}\, \mathsf{R_L}, \mathbf{false_L}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{false}} [\lceil C \rceil]}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]}\, \mathsf{L_F}}\, \supset_{\mathsf{L}}$$

□

To encode LK we use a well-known encoding [79], where both Γ and Δ are on the left side of the LJF sequent, and **false** is on the right side. The hypotheses from Γ will be encoded using the $\lfloor \cdot \rfloor$ predicate, and the conclusions will be encoded using the $\lceil \cdot \rceil$ predicate. As for LJ, the rules need to force a specific connective to be used, therefore we must not release focus on either $\lfloor \cdot \rfloor$ or $\lceil \cdot \rceil$ atoms on the right. This means that both sets of atoms must be positive.

The encoding of the rules for LK is given in Figure 6.5. The sequent $\Gamma \vdash_{\text{LK}} \Delta$ is encoded as $[\mathcal{L}_{\text{LK}}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil] \longrightarrow [\mathbf{false}]$.

The encoding of the rules for LK have several differences from the encoding of the rules in LJ. First, the principal formula always implies the rest of the formulas, because both the hypotheses and the conclusions are placed on the left of the LJF sequent. This also means that the right rules are negated ,which is why $\vee$ is encoded using $\wedge$, and vice-versa. The classical implication is encoded using $\vee$, same as the Cut rule. The I rule is encoded as an implication of false, because of the **false** in the encoding of the sequent.

This encoding encode LK on the full level of completeness between (open) derivations, as shown by the following proposition:

Proposition 6.3.2. *Let Γ and Δ be sets of LK formulas. Then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\text{LK}} \Delta \quad \textit{and} \quad [\mathcal{L}_{\text{LK}}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil] \longrightarrow [\mathbf{false}]$$

Proof. This proof follows in the same way as the proof for LJ, we show some cases here:
($\Rightarrow_{\mathsf{L}}$):

$$\dfrac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\text{LK}} A_1, \Delta \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\text{LK}} \Delta}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\text{LK}} \Delta}\, \Rightarrow_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\text{LK}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor\}$)

$$\dfrac{\dfrac{}{[\mathcal{K}] -\!\!\!\rightarrow_{\lfloor A_1 \Rightarrow A_2 \rfloor}} \mathsf{I_R} \qquad \dfrac{\dfrac{\dfrac{[\mathcal{K}, \lceil A_1 \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lceil A_1 \rceil \longrightarrow [\mathbf{false}]}\, []_{\mathsf{L}} \qquad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A_2 \rfloor \longrightarrow [\mathbf{false}]}\, []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lceil A_1 \rceil \vee \lfloor A_2 \rfloor} [\mathbf{false}]}\, \mathsf{R_L}, \vee_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \vee \lfloor A_2 \rfloor)} [\mathbf{false}]}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}\, \mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}\, \supset_{\mathsf{L}}$$

($\Rightarrow_{\mathsf{R}}$):

$$\dfrac{\Gamma, A_1 \vdash_{\text{LK}} A_1 \Rightarrow A_2, A_2, \Delta}{\Gamma \vdash_{\text{LK}} A_1 \Rightarrow A_2, \Delta}\, \Rightarrow_{\mathsf{R}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LK}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil \cup \{\lceil A_1 \Rightarrow A_2 \rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lceil A_1 \Rightarrow A_2 \rceil \rightarrow}\,\mathsf{I_R} \qquad \dfrac{\dfrac{[\mathcal{K}, \lfloor A_1 \rfloor, \lceil A_2 \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A_1 \rfloor, \lceil A_2 \rceil \longrightarrow [\mathbf{false}]}\,2 \times []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \rfloor \wedge^+ \lceil A_2 \rceil} [\mathbf{false}]}\,\mathsf{R_L}, \wedge^+_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil \supset (\lfloor A_1 \rfloor \wedge^+ \lceil A_2 \rceil)} [\mathbf{false}]}\,\supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}\,\mathsf{L_F}, 2 \times \forall_{\mathsf{L}}$$

(I):

$$\dfrac{}{\Gamma, A \vdash_{\mathrm{LK}} A, \Delta}\,\mathsf{I}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LK}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil \cup \{\lfloor A \rfloor, \lceil A \rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lfloor A \rfloor \rightarrow}\,\mathsf{I_R} \quad \dfrac{}{[\mathcal{K}] - \lceil A \rceil \rightarrow}\,\mathsf{I_R}}{[\mathcal{K}] - \lfloor A \rfloor \wedge^+ \lceil A \rceil \rightarrow}\,\wedge^+_{\mathsf{R}} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [\mathbf{false}]}\,\mathsf{R_L}, \mathbf{false_L}}{[\mathcal{K}] \xrightarrow{(\lfloor A \rfloor \wedge^+ \lceil A \rceil) \supset \mathbf{false}} [\mathbf{false}]}\,\supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}\,\mathsf{L_F}, \forall_{\mathsf{L}}$$

(Cut):

$$\dfrac{\Gamma \vdash_{\mathrm{LK}} A, \Delta \qquad \Gamma, A \vdash_{\mathrm{LK}} \Delta}{\Gamma \vdash_{\mathrm{LK}} \Delta}\,\mathsf{Cut}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LK}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}, \lfloor A \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \qquad \dfrac{[\mathcal{K}, \lceil A \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lceil A \rceil \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}}}{}\,\mathsf{R_L}, \vee_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A \rfloor \vee \lceil A \rceil} [\mathbf{false}]}\,\mathsf{L_F}, \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

□

6.3.2 Natural deduction

In this section we consider an intuitionistic fragment of the system of natural deduction NJ, based on the system by Sieg and Byrnes [91]. We use a formulation of the rules given by Pfenning [80]. The rules are presented in Figure 6.6. We use sequents on the form $\Gamma \vdash_{\mathrm{NJ}} C \uparrow$ when C is obtained in a bottom-up way (reasoning on the conclusion), and sequents on the form $\Gamma \vdash_{\mathrm{NJ}} C \downarrow$ when C is obtained in a top-down way (reasoning from the hypotheses). When a rule has a $\uparrow(\downarrow)$, it means that the arrow can be either $\uparrow$ or $\downarrow$, but that all arrows in the same rule instance must be the same. Our formulation of NJ is slightly different from NJ in the Nigam and Miller paper. In our version of the $(\bot_{\mathsf{E}})$ rule, the bottom sequent can either be a down arrow or an up arrow sequent. The arrows can be used to differentiate *normal proofs*, but we do not consider those here, except for mentioning that the different $(\bot_{\mathsf{E}})$ in Nigam and Miller's paper is due to the fact that only their version is allowed in normal proofs.

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \Rightarrow A_2 \downarrow \qquad \Gamma \vdash_{\mathrm{NJ}} A_1 \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_2 \downarrow} \Rightarrow_{\mathsf{E}} \qquad \frac{\Gamma, A_1 \vdash_{\mathrm{NJ}} A_2 \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_1 \Rightarrow A_2 \uparrow} \Rightarrow_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \wedge A_2 \downarrow}{\Gamma \vdash_{\mathrm{NJ}} A_i \downarrow} \wedge_{\mathsf{E}} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \uparrow \qquad \Gamma \vdash_{\mathrm{NJ}} A_2 \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_1 \wedge A_2 \uparrow} \wedge_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \vee A_2 \downarrow \qquad \Gamma, A_1 \vdash_{\mathrm{NJ}} C \uparrow(\downarrow) \qquad \Gamma, A_2 \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)} \vee_{\mathsf{E}} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A_i \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_1 \vee A_2 \uparrow} \vee_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{NJ}} \forall x A \downarrow}{\Gamma \vdash_{\mathrm{NJ}} A[t/x] \downarrow} \forall_{\mathsf{E}} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A[a/x] \uparrow}{\Gamma \vdash_{\mathrm{NJ}} \forall x A \uparrow} \forall_{\mathsf{I}}{}^{a}$$

$$\frac{\Gamma \vdash_{\mathrm{NJ}} \exists x A \downarrow \qquad \Gamma, A[a/x] \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)} \exists_{\mathsf{E}}{}^{a} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A[t/x] \uparrow}{\Gamma \vdash_{\mathrm{NJ}} \exists x A \uparrow} \exists_{\mathsf{I}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{NJ}} A \downarrow} \mathsf{I} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A \downarrow}{\Gamma \vdash_{\mathrm{NJ}} A \uparrow} \mathsf{M} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A \downarrow} \mathsf{S} \qquad \frac{}{\Gamma \vdash_{\mathrm{NJ}} \top \uparrow} \top_{\mathsf{I}} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} \bot \downarrow}{\Gamma \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)} \bot_{\mathsf{E}}$$

Figure 6.6: The proof system NJ. a is not free in Γ or C. $i \in \{1, 2\}$.

$(\Rightarrow_{\mathsf{E}})$	$\lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \supset \lfloor B \rfloor)$	$(\Rightarrow_{\mathsf{I}})$	$\lceil A \Rightarrow B \rceil \subset (\lfloor A \rfloor \supset \lceil B \rceil)$
$(\wedge_{\mathsf{E}})$	$\lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{-} \lfloor B \rfloor)$	$(\wedge_{\mathsf{I}})$	$\lceil A \wedge B \rceil \subset (\lceil A \rceil \wedge^{-} \lceil B \rceil)$
$(\vee_{\mathsf{E}})$	$\lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \vee \lfloor B \rfloor)$	$(\vee_{\mathsf{I}})$	$\lceil A \vee B \rceil \subset (\lceil A \rceil \vee \lceil B \rceil)$
$(\forall_{\mathsf{E}})$	$\lfloor \forall x A \rfloor \supset \forall x \lfloor A \rfloor$	$(\forall_{\mathsf{I}})$	$\lceil \forall x A \rceil \subset \forall x \lceil A \rceil$
$(\exists_{\mathsf{E}})$	$\lfloor \exists x A \rfloor \supset \exists x \lfloor A \rfloor$	$(\exists_{\mathsf{I}})$	$\lceil \exists x A \rceil \subset \exists x \lfloor A \rfloor$
(M)	$\lfloor A \rfloor \supset \lceil A \rceil$	(S)	$\lfloor A \rfloor \subset \lceil A \rceil$
$(\bot_{\mathsf{E}})$	$\lfloor \bot \rfloor \supset \mathbf{false}$	$(\top_{\mathsf{I}})$	$\lceil \top \rceil \subset \mathbf{true}$

Figure 6.7: Natural deduction, $\mathcal{L}_{\mathrm{NJ}}$.

We wish to encode NJ, using $\lfloor \cdot \rfloor$ for the hypotheses and for the $\downarrow$ conclusions. We use $\lceil \cdot \rceil$ for the $\uparrow$ conclusions. Furthermore, we encode the hypotheses on the left side of the LJF sequent and the conclusions on the right side, because NJ admits weakening on the left, but not on the right.

The introduction rules dictate, like for the sequent calculus, that $\lceil \cdot \rceil$ atoms have to be negative. And because we want to lose focus for the elimination rules $\lfloor \cdot \rfloor$ atoms have to be negative as well. We could possible use a delay construction like $\mathbf{true} \supset \lfloor A \rfloor$ to lose focus but that would clutter the rules unnecessarily.

The encodings of the rules are given in Figure 6.7. Interestingly, the rules are the same as for LJ; the only difference is the change of polarity. The sequent $\Gamma \vdash_{\mathrm{NJ}} A \downarrow$ is encoded as $[\mathcal{L}_{\mathrm{NJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lfloor A \rfloor]$. The sequent $\Gamma \vdash_{\mathrm{NJ}} A \uparrow$ is encoded as $[\mathcal{L}_{\mathrm{NJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil A \rceil]$.

This encoding encodes NJ on the full level of completeness:

Proposition 6.3.3. *Let $\Gamma \cup \{C\}$ be a set of NJ formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{NJ}} C \downarrow \quad \textit{and} \quad [\mathcal{L}_{\mathrm{NJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lfloor C \rfloor]$$

$$\Gamma \vdash_{\mathrm{NJ}} C \uparrow \quad \textit{and} \quad [\mathcal{L}_{\mathrm{NJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

Proof. The proof is similar to the proofs for LJ and LK, one remark is that the initial rule for NJ does not have a formula in $\mathcal{L}_{\mathrm{NJ}}$, but the proof goes through because focusing

on an atom succeeds exactly when the conclusion is the same atom, corresponding to the initial rule in NJ. We show a couple of the remaining cases here:

($\Rightarrow_{\mathsf{E}}$):

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \Rightarrow A_2 \downarrow \qquad \Gamma \vdash_{\mathrm{NJ}} A_1 \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_2 \downarrow} \Rightarrow_{\mathsf{E}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{NJ}} \cup \lfloor \Gamma \rfloor$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lfloor A_1 \Rightarrow A_2 \rfloor]}{[\mathcal{K}] -_{\lfloor A_1 \Rightarrow A_2 \rfloor}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \qquad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \rceil}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor} \lfloor A_2 \rfloor} \mathsf{I}_{\mathsf{L}}}{2 \times \supset_{\mathsf{L}}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \supset \lfloor A_2 \rfloor)} [\lfloor A_2 \rfloor]}{[\mathcal{K}] \longrightarrow [\lfloor A_2 \rfloor]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}$$

($\vee_{\mathsf{E}}$):

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_1 \vee A_2 \downarrow \qquad \Gamma, A_1 \vdash_{\mathrm{NJ}} C \uparrow(\downarrow) \qquad \Gamma, A_2 \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)} \vee_{\mathsf{E}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{NJ}} \cup \lfloor \Gamma \rfloor$ and F either $\lfloor C \rfloor$ or $\lceil C \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lfloor A_1 \vee A_2 \rfloor]}{[\mathcal{K}] -_{\lfloor A_1 \vee A_2 \rfloor}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \qquad \dfrac{\dfrac{[\mathcal{K}, \lfloor A_1 \rfloor] \longrightarrow [F]}{[\mathcal{K}], \lfloor A_1 \rfloor \longrightarrow [F]} []_{\mathsf{L}} \qquad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [F]}{[\mathcal{K}], \lfloor A_2 \rfloor \longrightarrow [F]} []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \rfloor \vee \lfloor A_2 \rfloor} [F]} \mathsf{R}_{\mathsf{L}}, \vee_{\mathsf{L}}}{\supset_{\mathsf{L}}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A_1 \vee A_2 \rfloor \supset (\lfloor A_1 \rfloor \vee \lfloor A_2 \rfloor)} [F]}{[\mathcal{K}] \longrightarrow [F]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}$$

($\vee_{\mathsf{I}}$):

$$\frac{\Gamma \vdash_{\mathrm{NJ}} A_i \uparrow}{\Gamma \vdash_{\mathrm{NJ}} A_1 \vee A_2 \uparrow} \vee_{\mathsf{I}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{NJ}} \cup \lfloor \Gamma \rfloor$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] \xrightarrow{\lceil A_1 \vee A_2 \rceil} [\lceil A_1 \vee A_2 \rceil]} \mathsf{I}_{\mathsf{L}} \qquad \dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lceil A_i \rceil]}{[\mathcal{K}] -_{\lceil A_i \rceil}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}}}{[\mathcal{K}] -_{\lceil A_1 \rceil \vee \lceil A_2 \rceil}\!\!\rightarrow} \vee_{\mathsf{R}}}{\supset_{\mathsf{L}}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lceil A_1 \vee A_2 \rceil \subset (\lceil A_1 \rceil \vee \lceil A_2 \rceil)} [\lceil A_1 \vee A_2 \rceil]}{[\mathcal{K}] \longrightarrow [\lceil A_1 \vee A_2 \rceil]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}$$

($\bot_{\mathsf{E}}$):

$$\frac{\Gamma \vdash_{\mathrm{NJ}} \bot \downarrow}{\Gamma \vdash_{\mathrm{NJ}} C \uparrow(\downarrow)} \bot_{\mathsf{E}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{NJ}} \cup \lfloor \Gamma \rfloor$ and F either $\lfloor C \rfloor$ or $\lceil C \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lfloor \bot \rfloor]}{[\mathcal{K}] -_{\lfloor \bot \rfloor} \rightarrow} \mathsf{R_R}, []_\mathsf{R} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [F]} \mathsf{R_L}, \mathbf{false}_\mathsf{L}}{[\mathcal{K}] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{false}} [F]} \supset_\mathsf{L}}{[\mathcal{K}] \longrightarrow [F]} \mathsf{L_F}$$

□

6.3.3 Generalized elimination rules

In this section we consider another system of natural deduction, where the form of elimination rules used for $\vee, \exists$ are used for all connectives. The system is similar to systems by Schroeder-Heister [89] and Von Plato [100]. We consider a form with sequents annotated with $\uparrow$ and $\downarrow$ (called GEA), like for NJ, and a form without (called GE). The rules are given in Figure 6.8 (not annotated) and Figure 6.9 (annotated). For GEA our formulation is slightly different from the formulation in the Nigam and Miller paper. The difference is the same difference the NJ system had. (The $(\bot_\mathsf{E})$ rule.)

For GE we have that $\lfloor \cdot \rfloor$ atoms are positive and $\lceil \cdot \rceil$ atoms are negative, the choice is dictated by the I rule, which must be able to focus on both A's. The encodings of the rules are given in Figure 6.10 and the sequent $\Gamma \vdash_{\mathrm{GE}} A$ is encoded as $[\mathcal{L}_{\mathrm{GE}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil A \rceil]$.

Besides from the missing cut rule, the rules are different from the LJ rules in one way: the elimination rules use $\lceil \cdot \rceil$ instead of $\lfloor \cdot \rfloor$. $\lceil \cdot \rceil$ is used, because there are no arrows in the sequent.

This encoding encodes GE on the full level of completeness:

Proposition 6.3.4. *Let $\Gamma \cup \{C\}$ be a set of GE formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{GE}} C \quad \textit{and} \quad [\mathcal{L}_{\mathrm{GE}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

Proof. The proof is similar to the earlier proofs, we show a single case:
$(\Rightarrow_{\mathsf{GE}})$:

$$\dfrac{\Gamma \vdash_{\mathrm{GE}} A_1 \Rightarrow A_2 \qquad \Gamma \vdash_{\mathrm{GE}} A_1 \qquad \Gamma, A_2 \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{GE}} C} \Rightarrow_{\mathsf{GE}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{GE}} \cup \lfloor \Gamma \rfloor$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \Rightarrow A_2 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \Rightarrow A_2 \rceil} \rightarrow} \mathsf{R_R}, []_\mathsf{R} \quad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \rceil} \rightarrow} \mathsf{R_R}, []_\mathsf{R} \quad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [\lceil C \rceil]}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor} [\lceil C \rceil]} \mathsf{R_L}, []_\mathsf{L}}{\dfrac{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil \supset (\lceil A_1 \rceil \supset \lfloor A_2 \rfloor)} [\lceil C \rceil]}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}, 2 \times \forall_\mathsf{L}} 2 \times \supset_\mathsf{L}$$

□

For GEA we have that $\lfloor \cdot \rfloor$ atoms are negative and $\lceil \cdot \rceil$ atoms are also negative, corresponding to the situation for the (annotated) NJ. The encodings of the rules are given in Figure 6.11, and the sequent $\Gamma \vdash_{\mathrm{GEA}} A \downarrow$ is encoded as $[\mathcal{L}_{\mathrm{GEA}}, \lfloor \Gamma \rfloor] \longrightarrow [\lfloor A \rfloor]$, and $\Gamma \vdash_{\mathrm{GEA}} A \uparrow$ is encoded as $[\mathcal{L}_{\mathrm{GEA}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil A \rceil]$,

The encoded rules are, besides from $\Rightarrow_{\mathsf{GE}}$ and $\forall_{\mathsf{GE}}$, the same as for NJ. These two rules are encoded using a positive conjunction and **true**. We need that construction,

$$\frac{\Gamma \vdash_{\mathrm{GE}} A_1 \Rightarrow A_2 \quad \Gamma \vdash_{\mathrm{GE}} A_1 \quad \Gamma, A_2 \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{GE}} C} \Rightarrow_{\mathsf{GE}} \qquad \frac{\Gamma, A_1 \vdash_{\mathrm{GE}} A_2}{\Gamma \vdash_{\mathrm{GE}} A_1 \Rightarrow A_2} \Rightarrow_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GE}} A_1 \wedge A_2 \quad \Gamma, A_1, A_2 \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{NJ}} C} \wedge_{\mathsf{GE}} \qquad \frac{\Gamma \vdash_{\mathrm{GE}} A_1 \quad \Gamma \vdash_{\mathrm{GE}} A_2}{\Gamma \vdash_{\mathrm{GE}} A_1 \wedge A_2} \wedge_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GE}} A_1 \vee A_2 \quad \Gamma, A_1 \vdash_{\mathrm{GE}} C \quad \Gamma, A_2 \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{GE}} C} \vee_{\mathsf{GE}} \qquad \frac{\Gamma \vdash_{\mathrm{GE}} A_i}{\Gamma \vdash_{\mathrm{GE}} A_1 \vee A_2} \vee_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GE}} \forall x A \quad \Gamma, A[t/x] \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{GE}} C} \forall_{\mathsf{GE}} \qquad \frac{\Gamma \vdash_{\mathrm{GE}} A[a/x]}{\Gamma \vdash_{\mathrm{GE}} \forall x A} \forall_{\mathsf{I}}{}^{a}$$

$$\frac{\Gamma \vdash_{\mathrm{GE}} \exists x A \quad \Gamma, A[a/x] \vdash_{\mathrm{GE}} C}{\Gamma \vdash_{\mathrm{GE}} C} \exists_{\mathsf{GE}}{}^{a} \qquad \frac{\Gamma \vdash_{\mathrm{GE}} A[t/x]}{\Gamma \vdash_{\mathrm{GE}} \exists x A} \exists_{\mathsf{I}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{GE}} A} \mathsf{I} \qquad \frac{}{\Gamma \vdash_{\mathrm{GE}} \top} \top_{\mathsf{I}} \qquad \frac{\Gamma \vdash_{\mathrm{GE}} \bot}{\Gamma \vdash_{\mathrm{GE}} C} \bot_{\mathsf{E}}$$

Figure 6.8: The proof system GE. a is not free in Γ or C. $i \in \{1, 2\}$.

$$\frac{\Gamma \vdash_{\mathrm{GEA}} A_1 \Rightarrow A_2 \downarrow \quad \Gamma \vdash_{\mathrm{GEA}} A_1 \uparrow \quad \Gamma, A_2 \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \Rightarrow_{\mathsf{GE}}$$

$$\frac{\Gamma, A_1 \vdash_{\mathrm{GEA}} A_2 \uparrow}{\Gamma \vdash_{\mathrm{GEA}} A_1 \Rightarrow A_2 \uparrow} \Rightarrow_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GEA}} A_1 \wedge A_2 \downarrow \quad \Gamma, A_1, A_2 \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \wedge_{\mathsf{GE}} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} A_1 \uparrow \quad \Gamma \vdash_{\mathrm{GEA}} A_2 \uparrow}{\Gamma \vdash_{\mathrm{GEA}} A_1 \wedge A_2 \uparrow} \wedge_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GEA}} A_1 \vee A_2 \downarrow \quad \Gamma, A_1 \vdash_{\mathrm{GEA}} C \uparrow(\downarrow) \quad \Gamma, A_2 \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \vee_{\mathsf{GE}}$$

$$\frac{\Gamma \vdash_{\mathrm{GEA}} A_i \uparrow}{\Gamma \vdash_{\mathrm{GEA}} A_1 \vee A_2 \uparrow} \vee_{\mathsf{I}}$$

$$\frac{\Gamma \vdash_{\mathrm{GEA}} \forall x A \downarrow \quad \Gamma, A[t/x] \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \forall_{\mathsf{GE}} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} A[a/x] \uparrow}{\Gamma \vdash_{\mathrm{GEA}} \forall x A \uparrow} \forall_{\mathsf{I}}{}^{a}$$

$$\frac{\Gamma \vdash_{\mathrm{GEA}} \exists x A \downarrow \quad \Gamma, A[a/x] \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \exists_{\mathsf{GE}}{}^{a} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} A[t/x] \uparrow}{\Gamma \vdash_{\mathrm{GEA}} \exists x A \uparrow} \exists_{\mathsf{I}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{GEA}} A \downarrow} \mathsf{I} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} A \downarrow}{\Gamma \vdash_{\mathrm{GEA}} A \uparrow} \mathsf{M} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} A \uparrow}{\Gamma \vdash_{\mathrm{GEA}} A \downarrow} \mathsf{S}$$

$$\frac{}{\Gamma \vdash_{\mathrm{GEA}} \top \uparrow} \top_{\mathsf{I}} \qquad \frac{\Gamma \vdash_{\mathrm{GEA}} \bot \downarrow}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \bot_{\mathsf{E}}$$

Figure 6.9: The proof system GEA (annotated GE). a is not free in Γ or C. $i \in \{1, 2\}$.

$$
\begin{array}{llll}
(\Rightarrow_{\mathsf{GE}}) & \lceil A \Rightarrow B \rceil \supset (\lceil A \rceil \supset \lfloor B \rfloor) & (\Rightarrow_{\mathsf{I}}) & \lceil A \Rightarrow B \rceil \subset (\lfloor A \rfloor \supset \lceil B \rceil) \\
(\wedge_{\mathsf{GE}}) & \lceil A \wedge B \rceil \supset (\lfloor A \rfloor \wedge^{+} \lfloor B \rfloor) & (\wedge_{\mathsf{I}}) & \lceil A \wedge B \rceil \subset (\lceil A \rceil \wedge^{-} \lceil B \rceil) \\
(\vee_{\mathsf{GE}}) & \lceil A \vee B \rceil \supset (\lfloor A \rfloor \vee \lfloor B \rfloor) & (\vee_{\mathsf{I}}) & \lceil A \vee B \rceil \subset (\lceil A \rceil \vee \lceil B \rceil) \\
(\forall_{\mathsf{GE}}) & \lceil \forall x A \rceil \supset \forall x \lfloor A \rfloor & (\forall_{\mathsf{I}}) & \lceil \forall x A \rceil \subset \forall x \lceil A \rceil \\
(\exists_{\mathsf{GE}}) & \lceil \exists x A \rceil \supset \exists x \lfloor A \rfloor & (\exists_{\mathsf{I}}) & \lceil \exists x A \rceil \subset \exists x \lceil A \rceil \\
(\mathsf{I}) & \lfloor A \rfloor \supset \lceil A \rceil & & \\
(\bot_{\mathsf{E}}) & \lfloor \bot \rfloor \supset \mathbf{false} & (\top_{\mathsf{I}}) & \lceil \top \rceil \subset \mathbf{true}
\end{array}
$$

Figure 6.10: Generalized elimination rules, $\mathcal{L}_{\mathrm{GE}}$.

$$
\begin{array}{llll}
(\Rightarrow_{\mathsf{GE}}) & \lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \supset (\lfloor B \rfloor \wedge^{+} \mathbf{true})) & (\Rightarrow_{\mathsf{I}}) & \lceil A \Rightarrow B \rceil \subset (\lfloor A \rfloor \supset \lceil B \rceil) \\
(\wedge_{\mathsf{GE}}) & \lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{+} \lfloor B \rfloor) & (\wedge_{\mathsf{I}}) & \lceil A \wedge B \rceil \subset (\lceil A \rceil \wedge^{-} \lceil B \rceil) \\
(\vee_{\mathsf{GE}}) & \lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \vee \lfloor B \rfloor) & (\vee_{\mathsf{I}}) & \lceil A \vee B \rceil \subset (\lceil A \rceil \vee \lceil B \rceil) \\
(\forall_{\mathsf{GE}}) & \lfloor \forall x A \rfloor \supset \forall x (\lfloor A \rfloor \wedge^{+} \mathbf{true}) & (\forall_{\mathsf{I}}) & \lceil \forall x A \rceil \subset \forall x \lceil A \rceil \\
(\exists_{\mathsf{GE}}) & \lceil \exists x A \rceil \supset \exists x \lfloor A \rfloor & (\exists_{\mathsf{I}}) & \lceil \exists x A \rceil \subset \exists x \lceil A \rceil \\
(\mathsf{M}) & \lfloor A \rfloor \supset \lceil A \rceil & (\mathsf{S}) & \lfloor A \rfloor \subset \lceil A \rceil \\
(\bot_{\mathsf{E}}) & \lfloor \bot \rfloor \supset \mathbf{false} & (\top_{\mathsf{I}}) & \lceil \top \rceil \subset \mathbf{true}
\end{array}
$$

Figure 6.11: Generalized elimination rules (annotated), $\mathcal{L}_{\mathrm{GEA}}$.

because the right side of the implication needs to be positive to loose focus. That is also why we use the positive conjunction for the $\wedge_{\mathsf{GE}}$ encoding. We note that the generalized rules from NJ $\vee_{\mathsf{GE}}$ and $\exists_{\mathsf{GE}}$ already have positive connectives, and therefore their encoding is the same.

This encoding encodes GEA on the full level of completeness:

Proposition 6.3.5. *Let $\Gamma \cup \{C\}$ be a set of GEA formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{GEA}} C \downarrow \quad \textit{and} \quad [\mathcal{L}_{\mathrm{GEA}}, \lfloor \Gamma \rfloor] \longrightarrow [\lfloor C \rfloor]$$

$$\Gamma \vdash_{\mathrm{GEA}} C \uparrow \quad \textit{and} \quad [\mathcal{L}_{\mathrm{GEA}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

Proof. The proof is similar to the earlier proofs, we show a single case:

($\Rightarrow_{\mathsf{GE}}$):

$$\frac{\Gamma \vdash_{\mathrm{GEA}} A_1 \Rightarrow A_2 \downarrow \qquad \Gamma \vdash_{\mathrm{GEA}} A_1 \uparrow \qquad \Gamma, A_2 \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)}{\Gamma \vdash_{\mathrm{GEA}} C \uparrow(\downarrow)} \Rightarrow_{\mathsf{GE}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{GEA}} \cup \lfloor \Gamma \rfloor$ and F either $\lfloor C \rfloor$ or $\lceil C \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\lfloor A_1 \Rightarrow A_2 \rfloor]}{[\mathcal{K}] -_{\lfloor A_1 \Rightarrow A_2 \rfloor}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \qquad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \rceil}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \qquad \dfrac{\dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [F]}{[\mathcal{K}], \lfloor A_2 \rfloor, \mathbf{true} \longrightarrow [F]} \mathbf{true}_{\mathsf{L}}, []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor \wedge^{+} \mathbf{true}} [F]} \mathsf{R}_{\mathsf{L}}, \wedge^{+}_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \supset (\lfloor A_2 \rfloor \wedge^{+} \mathbf{true}))} [F]} 2 \times \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [F]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}$$

□

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}'} A_1 \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{LJ}'} F}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}'} F} \Rightarrow_{\mathsf{L}} \quad \frac{\Gamma, A_1 \vdash_{\mathrm{LJ}'} A_2}{\Gamma \vdash_{\mathrm{LJ}'} A_1 \Rightarrow A_2} \Rightarrow_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \wedge A_2, A_i \vdash_{\mathrm{LJ}'} F}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{LJ}'} F} \wedge_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} A_1 \qquad \Gamma \vdash_{\mathrm{LJ}'} A_2}{\Gamma \vdash_{\mathrm{LJ}'} A_1 \wedge A_2} \wedge_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \vee A_2, A_1 \vdash_{\mathrm{LJ}'} F \qquad \Gamma, A_1 \vee A_2, A_2 \vdash_{\mathrm{LJ}'} F}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{LJ}'} F} \vee_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} A_i}{\Gamma \vdash_{\mathrm{LJ}'} A_1 \vee A_2} \vee_{\mathsf{R}}$$

$$\frac{\Gamma, \forall x A, A[t/x] \vdash_{\mathrm{LJ}'} F}{\Gamma, \forall x A \vdash_{\mathrm{LJ}'} F} \forall_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} A[a/x]}{\Gamma \vdash_{\mathrm{LJ}'} \forall x A} \forall_{\mathsf{R}}{}^{a}$$

$$\frac{\Gamma, \exists x A, A[a/x] \vdash_{\mathrm{LJ}'} F}{\Gamma, \exists x A \vdash_{\mathrm{LJ}'} F} \exists_{\mathsf{L}}{}^{a} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} A[t/x]}{\Gamma \vdash_{\mathrm{LJ}'} \exists x A} \exists_{\mathsf{R}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{LJ}'} A} \mathsf{I} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} A \qquad \Gamma, A \vdash_{\mathrm{LJ}'} F}{\Gamma \vdash_{\mathrm{LJ}'} F} \mathsf{Cut}$$

$$\frac{}{\Gamma, \bot \vdash_{\mathrm{LJ}'} \cdot} \bot_{\mathsf{L}} \quad \frac{\Gamma \vdash_{\mathrm{LJ}'} \cdot}{\Gamma \vdash_{\mathrm{LJ}'} C} \mathsf{W}_{\mathsf{R}} \quad \frac{}{\Gamma \vdash_{\mathrm{LJ}'} \top} \top_{\mathsf{R}}$$

Figure 6.12: The proof system LJ'. F is either $\cdot$ or a proper formula C. a is not free in Γ or F. $i \in \{1, 2\}$.

6.3.4 LJ with empty right sides

As we saw in Section 6.3.1, the version of LJ with non empty right sides could be encoded in a nice way using the meta-level **false**. In this section, we turn to the version of LJ, which uses the empty right sides, we call this version LJ'. The rules of LJ' are given in Figure 6.12. Except for making explicit the places where the right side can be empty (the left rules), the rules are identical to the rules from the Nigam and Miller paper. We use F to stand for either a formula or the empty right side.

To encode LJ', we cannot use the same encoding of the $\bot_{\mathsf{L}}$, rule because this version is too strong to be used in the system. The reason is that this encoding would allow us to derive $\Gamma, \bot \vdash_{\mathrm{LJ}'} C$ for any C, and the object-level rule only allows us to derive the empty right side.

To overcome this problem, we introduce a new atomic proposition **empty** (with type o) in the LJF meta-logic. This atom is assigned negative polarity. The rest of the encoding (including the polarities) follows the encoding for LJ, except for the conclusion where the empty conclusion is encoded as **empty**.

To summarize: the sequent $\Gamma \vdash_{\mathrm{LJ}'} C$ is encoded as $[\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$ and $\Gamma \vdash_{\mathrm{LJ}'} \cdot$ as $[\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\mathbf{empty}]$. The encoding of the rules for LJ' is given in Figure 6.13.

The encoding encodes LJ' on the full level of completeness between (open) derivations, as shown by the following proposition:

Proposition 6.3.6. *Let $\Gamma \cup \{C\}$ be a set of LJ' formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{LJ}'} C \quad \textit{and} \quad [\mathcal{L}_{\mathrm{LJ}'}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

and

$$\Gamma \vdash_{\mathrm{LJ}'} \cdot \quad \textit{and} \quad [\mathcal{L}_{\mathrm{LJ}'}, \lfloor \Gamma \rfloor] \longrightarrow [\mathbf{empty}]$$

Proof. This proof is very similar to the proof for LJ and we show a couple of cases. Below F in the LJF derivations stand for either **empty** or $\lceil C \rceil$.

$(\Rightarrow_{\mathsf{L}})$	$\lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \supset \lfloor B \rfloor)$	$(\Rightarrow_{\mathsf{R}})$	$\lceil A \Rightarrow B \rceil \subset (\lfloor A \rfloor \supset \lceil B \rceil)$
$(\wedge_{\mathsf{L}})$	$\lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{-} \lfloor B \rfloor)$	$(\wedge_{\mathsf{R}})$	$\lceil A \wedge B \rceil \subset (\lceil A \rceil \wedge^{-} \lceil B \rceil)$
$(\vee_{\mathsf{L}})$	$\lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \vee \lfloor B \rfloor)$	$(\vee_{\mathsf{R}})$	$\lceil A \vee B \rceil \subset (\lceil A \rceil \vee \lceil B \rceil)$
$(\forall_{\mathsf{L}})$	$\lfloor \forall x A \rfloor \supset \forall x \lfloor A \rfloor$	$(\forall_{\mathsf{R}})$	$\lceil \forall x A \rceil \subset \forall x \lceil A \rceil$
$(\exists_{\mathsf{L}})$	$\lfloor \exists x A \rfloor \supset \exists x \lfloor A \rfloor$	$(\exists_{\mathsf{R}})$	$\lceil \exists x A \rceil \subset \exists x \lceil A \rceil$
(I)	$\lfloor A \rfloor \supset \lceil A \rceil$	(Cut)	$\lfloor A \rfloor \subset \lceil A \rceil$
$(\bot_{\mathsf{L}})$	$\lfloor \bot \rfloor \supset \mathbf{empty}$	(W_R)	$\mathbf{empty} \supset \lceil A \rceil$
		$(\top_{\mathsf{R}})$	$\lceil \top \rceil \subset \mathbf{true}$

Figure 6.13: Intuitionistic sequent calculus with empty right sides, $\mathcal{L}_{\mathrm{LJ}'}$.

$(\Rightarrow_{\mathsf{L}})$:

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}'} A_1 \qquad \Gamma, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{LJ}'} F}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{LJ}'} F} \Rightarrow_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor A_1 \Rightarrow A_2 \rfloor}\!\!\rightarrow} \mathsf{I}_{\mathsf{R}} \quad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] -_{\lceil A_1 \rceil}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \quad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [F]}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor} [F]} \mathsf{R}_{\mathsf{L}}, []_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \supset \lfloor A_2 \rfloor)} [F]} 2 \times \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [F]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}$$

$(\bot_{\mathsf{L}})$:

$$\frac{}{\Gamma, \bot \vdash_{\mathrm{LJ}'} \cdot} \bot_{\mathsf{L}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor \bot \rfloor}\!\!\rightarrow} \mathsf{I}_{\mathsf{R}} \quad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{empty}} [\mathbf{empty}]} \mathsf{I}_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{empty}} [\mathbf{empty}]} \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{empty}]} \mathsf{L}_{\mathsf{F}}$$

$(\mathsf{W}_{\mathsf{R}})$:

$$\frac{\Gamma \vdash_{\mathrm{LJ}'} \cdot}{\Gamma \vdash_{\mathrm{LJ}'} C} \mathsf{W}_{\mathsf{R}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}] \longrightarrow [\mathbf{empty}]}{[\mathcal{K}] -_{\mathbf{empty}}\!\!\rightarrow} \mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \quad \dfrac{}{[\mathcal{K}] \xrightarrow{\lceil C \rceil} [\lceil C \rceil]} \mathsf{I}_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\mathbf{empty} \supset \lceil C \rceil} [\lceil C \rceil]} \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L}_{\mathsf{F}}, \forall_{\mathsf{L}}$$

□

$$\frac{\Gamma \vdash_{\mathrm{FD}} A_1 \Rightarrow A_2, \Delta \quad \Gamma \vdash_{\mathrm{FD}} A_1, \Delta \quad \Gamma, A_2 \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \Rightarrow_{\mathsf{GE}}$$

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{FD}} \Delta \quad \Gamma, A_1 \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \Rightarrow_{\mathsf{GI}1} \qquad \frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{FD}} \Delta \quad \Gamma \vdash_{\mathrm{FD}} A_2, \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \Rightarrow_{\mathsf{GI}2}$$

$$\frac{\Gamma \vdash_{\mathrm{FD}} A_1 \wedge A_2, \Delta \quad \Gamma, A_i \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \wedge_{\mathsf{GE}}$$

$$\frac{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{FD}} \Delta \quad \Gamma \vdash_{\mathrm{FD}} A_1, \Delta \quad \Gamma \vdash_{\mathrm{FD}} A_2, \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \wedge_{\mathsf{GI}}$$

$$\frac{\Gamma \vdash_{\mathrm{FD}} A_1 \vee A_2, \Delta \quad \Gamma, A_1 \vdash_{\mathrm{FD}} \Delta \quad \Gamma, A_2 \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \vee_{\mathsf{GE}}$$

$$\frac{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{FD}} \Delta \quad \Gamma \vdash_{\mathrm{FD}} A_i, \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \vee_{\mathsf{GI}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{FD}} A, \Delta} \mathsf{I}$$

$$\frac{\Gamma, \neg A \vdash_{\mathrm{FD}} \Delta \quad \Gamma, A \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \neg_{\mathsf{GI}1} \qquad \frac{\Gamma, \vdash_{\mathrm{FD}} \neg A, \Delta \quad \Gamma \vdash_{\mathrm{FD}} A, \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \neg_{\mathsf{GI}2}$$

Figure 6.14: The proof system FD. $i \in \{1, 2\}$.

$(\Rightarrow_{\mathsf{GE}})$	$\lceil A \Rightarrow B \rceil \vee \lceil A \rceil \vee \lfloor B \rfloor$	$(\Rightarrow_{\mathsf{GI}1})$	$\lfloor A \Rightarrow B \rfloor \vee \lfloor A \rfloor$
		$(\Rightarrow_{\mathsf{GI}2})$	$\lfloor A \Rightarrow B \rfloor \vee \lceil B \rceil$
$(\wedge_{\mathsf{GE}1})$	$\lceil A \wedge B \rceil \vee \lfloor A \rfloor$	$(\wedge_{\mathsf{GI}})$	$\lfloor A \wedge B \rfloor \vee \lceil A \rceil \vee \lceil B \rceil$
$(\wedge_{\mathsf{GE}2})$	$\lceil A \wedge B \rceil \vee \lfloor B \rfloor$		
$(\vee_{\mathsf{GE}})$	$\lceil A \vee B \rceil \vee \lfloor A \rfloor \vee \lfloor B \rfloor$	$(\vee_{\mathsf{GI}1})$	$\lfloor A \vee B \rfloor \vee \lceil A \rceil$
		$(\vee_{\mathsf{GI}2})$	$\lfloor A \vee B \rfloor \vee \lceil B \rceil$
$(\neg_{\mathsf{GI}1})$	$\lfloor \neg A \rfloor \vee \lfloor A \rfloor$	$(\neg_{\mathsf{GI}2})$	$\lceil \neg A \rceil \vee \lceil A \rceil$
(I)	$(\lfloor A \rfloor \wedge^{+} \lceil A \rceil) \supset \mathbf{false}$		

Figure 6.15: Free deduction, $\mathcal{L}_{\mathrm{FD}}$.

6.3.5 Free deduction

In this section we consider an additive version of the system of free deduction FD [76], the rules are given in Figure 6.14. This system covers only the propositional fragment of classical logic and uses an explicit negation $\neg$.

FD will be encoded in the same way as LK, so for FD we have that both $\lfloor \cdot \rfloor$ atoms and $\lceil \cdot \rceil$ atoms are positive. The encodings of the rules are given in Figure 6.15, and the sequent $\Gamma \vdash_{\mathrm{FD}} \Delta$ is encoded as $[\mathcal{L}_{\mathrm{FD}}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil] \longrightarrow [\mathbf{false}]$.

The encodings of the rules are similar to the rules for LK, but where LK uses implication, FD uses disjunction. The disjunction is used, because the principal formula is above the inference line, and therefore we must not focus on it. It might be possible to use a construction, like the one for GEA, to loose focus explicitly, but we prefer this version.

This encoding encodes FD on the full level of completeness:

Proposition 6.3.7. *Let Γ and Δ be sets of FD formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{FD}} \Delta \quad \textit{and} \quad [\mathcal{L}_{\mathrm{FD}}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil] \longrightarrow [\mathbf{false}]$$

Proof. The proof is similar to the earlier proofs, but the rules are quite different so we show a couple of cases:

($\Rightarrow_{\mathsf{GE}}$):

$$\frac{\Gamma \vdash_{\mathrm{FD}} A_1 \Rightarrow A_2, \Delta \qquad \Gamma \vdash_{\mathrm{FD}} A_1, \Delta \qquad \Gamma, A_2 \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \Rightarrow_{\mathsf{GE}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{FD}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}, \lceil A_1 \Rightarrow A_2 \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lceil A_1 \Rightarrow A_2 \rceil \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \quad \dfrac{[\mathcal{K}, \lceil A_1 \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lceil A_1 \rceil \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \quad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A_2 \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}}}{\qquad}\,\mathsf{R_L}, 2 \times \vee_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil \vee \lceil A_1 \rceil \vee \lfloor A_2 \rfloor} [\mathbf{false}]}\,\mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

($\Rightarrow_{\mathsf{GI1}}$):

$$\frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{FD}} \Delta \qquad \Gamma, A_1 \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \Rightarrow_{\mathsf{GI1}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{FD}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}, \lfloor A_1 \Rightarrow A_2 \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A_1 \Rightarrow A_2 \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \quad \dfrac{[\mathcal{K}, \lfloor A_1 \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A_1 \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}}}{\qquad}\,\mathsf{R_L}, \vee_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \vee \lfloor A_1 \rfloor} [\mathbf{false}]}\,\mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

($\neg_{\mathsf{GI1}}$):

$$\frac{\Gamma, \neg A \vdash_{\mathrm{FD}} \Delta \qquad \Gamma, A \vdash_{\mathrm{FD}} \Delta}{\Gamma \vdash_{\mathrm{FD}} \Delta} \neg_{\mathsf{GI1}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{FD}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil$)

$$\dfrac{\dfrac{\dfrac{[\mathcal{K}, \lfloor \neg A \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor \neg A \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \quad \dfrac{[\mathcal{K}, \lfloor A \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}], \lfloor A \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}}}{\qquad}\,\mathsf{R_L}, \vee_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor \neg A \rfloor \vee \lfloor A \rfloor} [\mathbf{false}]}\,\mathsf{L_F}, \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

□

6.3.6 Tableaux

In this section we consider a system of tableaux KE [20], the rules are given in Figure 6.16. Like FD, this system covers only the propositional fragment, again including an explicit negation.

KE will be encoded in the same way as LK, so for KE we have that both $\lfloor \cdot \rfloor$ atoms and $\lceil \cdot \rceil$ atoms are positive. The encodings of the rules are given in Figure 6.17 and the sequent $\Gamma \vdash_{\mathrm{KE}} \Delta$ is encoded as $[\mathcal{L}_{\mathrm{KE}}, \lfloor \Gamma \rfloor, \lceil \Delta \rceil] \longrightarrow [\mathbf{false}]$.

The rules that correspond to LK rules have the same encoding ($\Rightarrow_{\mathsf{R}}$,$\wedge_{\mathsf{L}}$,$\vee_{\mathsf{R}}$), the rules, which are split into two rules, have two encodings each, using an implication to focus on either A_1 or A_2 in the rule, corresponding to the formula that needs to be under the inference line.

This encoding encodes KE on the full level of completeness:

$$\frac{\Gamma, A_1, A_1 \Rightarrow A_2, A_2 \vdash_{\mathrm{KE}} \Delta}{\Gamma, A_1, A_1 \Rightarrow A_2 \vdash_{\mathrm{KE}} \Delta} \Rightarrow_{\mathsf{L}1} \quad \frac{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{KE}} A_1, A_2, \Delta}{\Gamma, A_1 \Rightarrow A_2 \vdash_{\mathrm{KE}} A_2, \Delta} \Rightarrow_{\mathsf{L}2}$$

$$\frac{\Gamma, A_1 \vdash_{\mathrm{KE}} A_1 \Rightarrow A_2, A_2, \Delta}{\Gamma, \vdash_{\mathrm{KE}} A_1 \Rightarrow A_2, \Delta} \Rightarrow_{\mathsf{R}}$$

$$\frac{\Gamma, A_1 \wedge A_2, A_1, A_2 \vdash_{\mathrm{KE}} \Delta}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{KE}} \Delta} \wedge_{\mathsf{L}} \quad \frac{\Gamma, A_1 \vdash_{\mathrm{KE}} A_1 \wedge A_2, A_2, \Delta}{\Gamma, A_1 \vdash_{\mathrm{KE}} A_1 \wedge A_2, \Delta} \wedge_{\mathsf{R}1}$$

$$\frac{\Gamma, A_2 \vdash_{\mathrm{KE}} A_1 \wedge A_2, A_1, \Delta}{\Gamma, A_2 \vdash_{\mathrm{KE}} A_1 \wedge A_2, \Delta} \wedge_{\mathsf{R}2}$$

$$\frac{\Gamma, A_1 \vee A_2, A_2 \vdash_{\mathrm{KE}} A_1, \Delta}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{KE}} A_1, \Delta} \vee_{\mathsf{L}1} \quad \frac{\Gamma, A_1 \vee A_2, A_1 \vdash_{\mathrm{KE}} A_2, \Delta}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{KE}} A_2, \Delta} \vee_{\mathsf{L}2}$$

$$\frac{\Gamma \vdash_{\mathrm{KE}} A_1 \vee A_2, A_1, A_2, \Delta}{\Gamma \vdash_{\mathrm{KE}} A_1 \vee A_2, \Delta} \vee_{\mathsf{R}}$$

$$\frac{\Gamma, \neg A \vdash_{\mathrm{KE}} A, \Delta}{\Gamma, \neg A \vdash_{\mathrm{KE}} \Delta} \neg_{\mathsf{L}} \quad \frac{\Gamma, A \vdash_{\mathrm{KE}} \neg A, \Delta}{\Gamma \vdash_{\mathrm{KE}} \neg A, \Delta} \neg_{\mathsf{R}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{KE}} A, \Delta} \mathsf{I} \quad \frac{\Gamma, A \vdash_{\mathrm{KE}} \Delta \qquad \Gamma \vdash_{\mathrm{KE}} A, \Delta}{\Gamma \vdash_{\mathrm{KE}} \Delta} \mathsf{Cut}$$

Figure 6.16: The proof system KE.

$(\Rightarrow_{\mathsf{L}1})$	$\lfloor A \Rightarrow B \rfloor \supset (\lfloor A \rfloor \supset \lfloor B \rfloor)$	$(\Rightarrow_{\mathsf{R}})$	$\lceil A \Rightarrow B \rceil \supset (\lfloor A \rfloor \wedge^{+} \lceil B \rceil)$
$(\Rightarrow_{\mathsf{L}2})$	$\lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \subset \lceil B \rceil)$		
$(\wedge_{\mathsf{L}})$	$\lfloor A \wedge B \rfloor \supset (\lfloor A \rfloor \wedge^{+} \lfloor B \rfloor)$	$(\wedge_{\mathsf{R}1})$	$\lceil A \wedge B \rceil \supset (\lfloor A \rfloor \supset \lceil B \rceil)$
		$(\wedge_{\mathsf{R}2})$	$\lceil A \wedge B \rceil \supset (\lceil A \rceil \subset \lfloor B \rfloor)$
$(\vee_{\mathsf{L}1})$	$\lfloor A \vee B \rfloor \supset (\lceil A \rceil \supset \lfloor B \rfloor)$	$(\vee_{\mathsf{R}})$	$\lceil A \vee B \rceil \supset (\lceil A \rceil \wedge^{+} \lceil B \rceil)$
$(\vee_{\mathsf{L}2})$	$\lfloor A \vee B \rfloor \supset (\lfloor A \rfloor \subset \lceil B \rceil)$		
$(\neg_{\mathsf{L}})$	$\lfloor \neg A \rfloor \supset \lceil A \rceil$	$(\neg_{\mathsf{R}})$	$\lceil \neg A \rceil \supset \lfloor A \rfloor$
(I)	$(\lfloor A \rfloor \wedge^{+} \lceil A \rceil) \supset \mathbf{false}$	(Cut)	$\lfloor A \rfloor \vee \lceil A \rceil$

Figure 6.17: Classical tableaux, $\mathcal{L}_{\mathrm{KE}}$.

Proposition 6.3.8. *Let Γ and Δ be sets of KE formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{KE}} \Delta \quad \textit{and} \quad [\mathcal{L}_{\mathrm{KE}}, \lfloor\Gamma\rfloor, \lceil\Delta\rceil] \longrightarrow [\mathbf{false}]$$

Proof. The proof is similar to the previous proofs, we show a couple of cases:
($\vee_{\mathsf{L}1}$):

$$\frac{\Gamma, A_1 \vee A_2, A_2 \vdash_{\mathrm{KE}} A_1, \Delta}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{KE}} A_1, \Delta} \vee_{\mathsf{L}1}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{KE}} \cup \lfloor\Gamma\rfloor \cup \lceil\Delta\rceil \cup \{\lfloor A_1 \vee A_2\rfloor, \lceil A_1\rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -\!\!\!\rightarrow_{\lfloor A_1 \vee A_2\rfloor}} \mathsf{I}_{\mathsf{R}} \quad \dfrac{}{[\mathcal{K}] -\!\!\!\rightarrow_{\lceil A_1\rceil}} \mathsf{I}_{\mathsf{R}} \quad \dfrac{[\mathcal{K}, \lfloor A_2\rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}] \xrightarrow{\lfloor A_2\rfloor} [\mathbf{false}]} \mathsf{R}_{\mathsf{L}}, []_{\mathsf{L}}}{\lfloor A_1 \vee A_2\rfloor \supset (\lceil A_1\rceil \supset \lfloor A_2\rfloor)} 2 \times \supset_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \vee A_2\rfloor \supset (\lceil A_1\rceil \supset \lfloor A_2\rfloor)} [\mathbf{false}]} \mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

($\neg_{\mathsf{R}}$):

$$\frac{\Gamma, A \vdash_{\mathrm{KE}} \neg A, \Delta}{\Gamma \vdash_{\mathrm{KE}} \neg A, \Delta} \neg_{\mathsf{R}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{KE}} \cup \lfloor\Gamma\rfloor \cup \lceil\Delta\rceil \cup \{\lceil\neg A\rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -\!\!\!\rightarrow_{\lceil\neg A\rceil}} \mathsf{I}_{\mathsf{R}} \quad \dfrac{[\mathcal{K}, \lfloor A\rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}] \xrightarrow{\lfloor A\rfloor} [\mathbf{false}]} \mathsf{R}_{\mathsf{L}}, []_{\mathsf{L}}}{\lceil\neg A\rceil \supset \lfloor A\rfloor} \supset_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lceil\neg A\rceil \supset \lfloor A\rfloor} [\mathbf{false}]} \mathsf{L}_{\mathsf{F}}, \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

□

6.3.7 Analytic cut

In this section we consider Smullyan's proof system for analytic cuts AC [92], the rules are given in Figure 6.18. As in Nigam and Miller's paper, the side condition on cut is dropped. Only propositional formulas are included, and we use an explicit negation.

AC will be encoded in the same way as LK, so for AC we have that both $\lfloor\cdot\rfloor$ atoms and $\lceil\cdot\rceil$ atoms are positive. The encodings of the rules are given in Figure 6.19 and the sequent $\Gamma \vdash_{\mathrm{AC}} \Delta$ is encoded as $[\mathcal{L}_{\mathrm{AC}}, \lfloor\Gamma\rfloor, \lceil\Delta\rceil] \longrightarrow [\mathbf{false}]$.

The encoded rules are pretty verbose, because every left and right rule must be able to end the proof and therefore prove **false**. The right side of the implication in the encodings of the rules is basically the double negation of the right side for LK. We note that some of the rules can be joined into one formula in the encoding.

This encoding encodes AC on the full level of completeness:

Proposition 6.3.9. *Let Γ and Δ be sets of AC formulas, then there is a bijective correspondence between the open derivations of the following sequents:*

$$\Gamma \vdash_{\mathrm{AC}} \Delta \quad \textit{and} \quad [\mathcal{L}_{\mathrm{AC}}, \lfloor\Gamma\rfloor, \lceil\Delta\rceil] \longrightarrow [\mathbf{false}]$$

$$\frac{}{\Gamma, A_1, A_1 \Rightarrow A_2 \vdash_{\mathrm{AC}} A_2, \Delta}\,{\Rightarrow_{\mathsf{L}}} \qquad \frac{}{\Gamma \vdash_{\mathrm{AC}} A_1, A_1 \Rightarrow A_2, \Delta}\,{\Rightarrow_{\mathsf{R}1}}$$

$$\frac{}{\Gamma, A_2 \vdash_{\mathrm{AC}} A_1 \Rightarrow A_2, \Delta}\,{\Rightarrow_{\mathsf{R}2}}$$

$$\frac{}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{AC}} A_1, \Delta}\,{\wedge_{\mathsf{L}1}} \qquad \frac{}{\Gamma, A_1 \wedge A_2 \vdash_{\mathrm{AC}} A_2, \Delta}\,{\wedge_{\mathsf{L}2}} \qquad \frac{}{\Gamma, A_1, A_2 \vdash_{\mathrm{AC}} A_1 \wedge A_2, \Delta}\,{\wedge_{\mathsf{R}}}$$

$$\frac{}{\Gamma, A_1 \vee A_2 \vdash_{\mathrm{AC}} A_1, A_2, \Delta}\,{\vee_{\mathsf{L}}} \qquad \frac{}{\Gamma, A_1 \vdash_{\mathrm{AC}} A_1 \vee A_2, \Delta}\,{\vee_{\mathsf{R}1}} \qquad \frac{}{\Gamma, A_2 \vdash_{\mathrm{AC}} A_1 \vee A_2, \Delta}\,{\vee_{\mathsf{R}2}}$$

$$\frac{}{\Gamma, \neg A, A \vdash_{\mathrm{AC}} \Delta}\,{\neg_{\mathsf{L}}} \qquad \frac{}{\Gamma \vdash_{\mathrm{AC}} \neg A, A, \Delta}\,{\neg_{\mathsf{R}}}$$

$$\frac{}{\Gamma, A \vdash_{\mathrm{AC}} A, \Delta}\,\mathsf{I} \qquad \frac{\Gamma, A \vdash_{\mathrm{AC}} \Delta \qquad \Gamma \vdash_{\mathrm{AC}} A, \Delta}{\Gamma \vdash_{\mathrm{AC}} \Delta}\,\mathsf{Cut}$$

Figure 6.18: The proof system AC.

$(\Rightarrow_{\mathsf{L}})$ $\lfloor A \Rightarrow B \rfloor \supset ((\lfloor A \rfloor \wedge^{+} \lceil B \rceil) \supset \mathbf{false})$	$(\Rightarrow_{\mathsf{R}})$ $\lceil A \Rightarrow B \rceil \supset ((\lceil A \rceil \vee \lfloor B \rfloor) \supset \mathbf{false})$
$(\wedge_{\mathsf{L}})$ $\lfloor A \wedge B \rfloor \supset ((\lceil A \rceil \vee \lceil B \rceil) \supset \mathbf{false})$	$(\wedge_{\mathsf{R}})$ $\lceil A \wedge B \rceil \supset ((\lfloor A \rfloor \wedge^{+} \lfloor B \rfloor) \supset \mathbf{false})$
$(\vee_{\mathsf{L}})$ $\lfloor A \vee B \rfloor \supset ((\lceil A \rceil \wedge^{+} \lceil B \rceil) \supset \mathbf{false})$	$(\vee_{\mathsf{R}})$ $\lceil A \vee B \rceil \supset ((\lfloor A \rfloor \vee \lfloor B \rfloor) \supset \mathbf{false})$
$(\neg_{\mathsf{L}})$ $\lfloor \neg A \rfloor \supset (\lfloor A \rfloor \supset \mathbf{false})$	$(\neg_{\mathsf{R}})$ $\lceil \neg A \rceil \supset (\lceil A \rceil \supset \mathbf{false})$
(I) $(\lfloor A \rfloor \wedge^{+} \lceil A \rceil) \supset \mathbf{false}$	(Cut) $\lfloor A \rfloor \vee \lceil A \rceil$

Figure 6.19: Smullyan's analytic cut, $\mathcal{L}_{\mathrm{AC}}$.

Proof. The proof is similar to the previous proofs, and we show a couple of cases. We note that in those cases where two rules are joined into one formula, each rule corresponds exactly to one branch in the LJF derivation (one of the different available macro-rules):

$(\Rightarrow_{\mathsf{L}})$:

$$\frac{}{\Gamma, A_1, A_1 \Rightarrow A_2 \vdash_{\mathrm{AC}} A_2, \Delta}\,{\Rightarrow_{\mathsf{L}}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{AC}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor, \lfloor A_1 \rfloor, \lceil A_2 \rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lfloor A_1 \Rightarrow A_2 \rfloor \rightarrow}\,\mathsf{I_R} \qquad \dfrac{\dfrac{}{[\mathcal{K}] - \lfloor A_1 \rfloor \rightarrow}\,\mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}] - \lceil A_2 \rceil \rightarrow}\,\mathsf{I_R}}{[\mathcal{K}] - \lfloor A_1 \rfloor \wedge^{+} \lceil A_2 \rceil \rightarrow}\,\wedge^{+}_{\mathsf{R}} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [\mathbf{false}]}\,\mathsf{R_L}, \mathbf{false_L}}{}\,2 \times \supset_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset ((\lfloor A_1 \rfloor \wedge^{+} \lceil A_2 \rceil) \supset \mathbf{false})} [\mathbf{false}]}\,\mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

$(\Rightarrow_{\mathsf{R}1})$:

$$\frac{}{\Gamma \vdash_{\mathrm{AC}} A_1, A_1 \Rightarrow A_2, \Delta}\,{\Rightarrow_{\mathsf{R}1}}$$

corresponds to (with $\mathcal{K} = \mathcal{L}_{\mathrm{AC}} \cup \lfloor \Gamma \rfloor \cup \lceil \Delta \rceil \cup \{\lceil A_1 \Rightarrow A_2 \rceil, \lceil A_1 \rceil\}$)

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lceil A_1 \Rightarrow A_2 \rceil \rightarrow}\,\mathsf{I_R} \qquad \dfrac{\dfrac{}{[\mathcal{K}] - \lceil A_1 \rceil \rightarrow}\,\mathsf{I_R}}{[\mathcal{K}] - \lceil A_1 \rceil \vee \lfloor A_2 \rfloor \rightarrow}\,\vee_{\mathsf{R}} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [\mathbf{false}]}\,\mathsf{R_L}, \mathbf{false_L}}{}\,2 \times \supset_{\mathsf{L}}}{[\mathcal{K}] \xrightarrow{\lceil A_1 \Rightarrow A_2 \rceil \supset ((\lceil A_1 \rceil \vee \lfloor A_2 \rfloor) \supset \mathbf{false})} [\mathbf{false}]}\,\mathsf{L_F}, 2 \times \forall_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{false}]}$$

□

6.4 Relative completeness

This section is concerned with the different relative completeness theorems. We show how to prove correspondence between the different systems, using the formalizations from the previous section. The proofs illustrate, how easy it is to come with a new system and show that it is complete with respect to intuitionistic or classical logic.

There are several ways to prove relative completeness. In this work we use three different methods.

1. LJF method: we exploit the completeness theorems from the previous section and do the proof by induction on the focused derivations of the encoded sequents in LJF.

2. Intuitionistic method: We use the completeness theorems and Corollary 6.2.2 to map the encoded sequents to intuitionistic logic and reason directly in intuitionistic logic. If we can show that each set of rules follows from the other, this approach gives the simplest proof.

   If we can show that the rules in one system follows from the rules in the other we call it proof by *rule implication*, and if we can show both directions, we call it proof by *rule equivalence*.

3. Object-level method: we reason directly in the object-level systems, using induction on the derivations. This method does not use the encodings, so we use this method as little as possible.

Most of the proofs in this section use option two. The cases wherein we need the other methods are the cases where cut elimination is needed, where we use a transformation on the formulas, or when we try to relate structurally different systems (e.g. LJ and LJ' or intuitionistic and classical logic).

6.4.1 Intuitionistic systems

In this section we compare the different intuitionistic systems. A general comment about the encoded systems is that their relative completeness with regard to each other is in most of the cases trivial, as the systems mostly have the same rules, save for polarities, which does not effect provability.

The first proposition shows correspondence between LJ and NJ:

Proposition 6.4.1. *Let $\Gamma \cup \{C\}$ be a set of object formulas, then:*

$$\Gamma \vdash_{\mathrm{LJ}} C \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{NJ}} C \uparrow$$

Proof. The idea behind the proof is to use the completeness results from Section 6.3 and then reason in intuitionistic logic.

Using Proposition 6.3.1 and Proposition 6.3.3, we need to show that (keep in mind the different polarities on the left and the right side):

$$[\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil] \quad \text{if and only if} \quad [\mathcal{L}_{\mathrm{NJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

which by Corollary 6.2.2 is the same as showing that:

$$\mathcal{L}_{\mathrm{LJ}}{}^{\circ}, \lfloor \Gamma \rfloor \vdash_{\mathrm{I}} \lceil C \rceil \quad \text{if and only if} \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor \Gamma \rfloor \vdash_{\mathrm{I}} \lceil C \rceil$$

And if we can show that:

$$\forall F \in \mathcal{L}_{\mathrm{NJ}}{}^{\circ} \quad \mathcal{L}_{\mathrm{LJ}}{}^{\circ} \vdash_{\mathrm{I}} F$$
$$\forall F \in \mathcal{L}_{\mathrm{LJ}}{}^{\circ} \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ} \vdash_{\mathrm{I}} F$$

the result follows, because we can cut each formula in the logic encoding, as the following derivation shows (for the "only if"-direction):

$$\dfrac{\mathcal{L}_{\mathrm{NJ}}{}^{\circ} \vdash_{\mathrm{I}} F_1^{\mathrm{LJ}} \quad \vdots \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor \vdash_{\mathrm{I}} F_1^{\mathrm{LJ}} \qquad \dfrac{\mathcal{L}_{\mathrm{NJ}}{}^{\circ} \vdash_{\mathrm{I}} F_2^{\mathrm{LJ}} \quad \vdots \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor, F_1^{\mathrm{LJ}} \vdash_{\mathrm{I}} F_2^{\mathrm{LJ}} \qquad \dfrac{\lfloor\Gamma\rfloor, \mathcal{L}_{\mathrm{LJ}}{}^{\circ} \vdash_{\mathrm{I}} \lceil C\rceil \quad \vdots \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor, \mathcal{L}_{\mathrm{LJ}}{}^{\circ} \vdash_{\mathrm{I}} \lceil C\rceil \quad \vdots \quad \mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor, F_1^{\mathrm{LJ}}, F_2^{\mathrm{LJ}} \vdash_{\mathrm{I}} \lceil C\rceil}{\mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor, F_1^{\mathrm{LJ}} \vdash_{\mathrm{I}} \lceil C\rceil}}{\mathcal{L}_{\mathrm{NJ}}{}^{\circ}, \lfloor\Gamma\rfloor \vdash_{\mathrm{I}} \lceil C\rceil}$$

What remains is to show that all the formulas in the first logic follows from the rules in the second logic, and vice-versa. In this case this is highly trivial, as $\mathcal{L}_{\mathrm{LJ}}{}^{\circ} = \mathcal{L}_{\mathrm{NJ}}{}^{\circ}$. □

For the systems with generalized elimination rules, the situation is the same as in the Nigam and Miller paper: for GE we can only prove one direction of equivalence to LJ, because of the missing cut rule. If we remove the cut rule from LJ, we can prove full correspondence, which is witnessed by the following two propositions:

Proposition 6.4.2. *Let $\Gamma \cup \{C\}$ be a set of object formulas, then:*

$$\textit{If} \quad \Gamma \vdash_{\mathrm{GE}} C \quad \textit{then} \quad \Gamma \vdash_{\mathrm{LJ}} C$$

Proof. Like in the proof for relative completeness of LJ and NJ, we take a detour to intuitionistic logic using Propositions 6.3.1, 6.3.3 and Corollary 6.2.2. Therefore, it comes down to showing that:

$$\text{If} \quad \mathcal{L}_{\mathrm{GE}}{}^{\circ}, \lfloor\Gamma\rfloor \vdash_{\mathrm{I}} \lceil C\rceil \quad \text{then} \quad \mathcal{L}_{\mathrm{LJ}}{}^{\circ}, \lfloor\Gamma\rfloor \vdash_{\mathrm{I}} \lceil C\rceil$$

which follows, if we can show that:

$$\forall F \in \mathcal{L}_{\mathrm{GE}}{}^{\circ} \quad \mathcal{L}_{\mathrm{LJ}}{}^{\circ} \vdash_{\mathrm{I}} F$$

We will show one of these consequences and leave the rest for the reader. In the proof of these consequences, we will not give a full derivation in $\vdash_{\mathrm{I}}$, but rather in a less verbose way; it should be clear, how to convert our proof into a derivation.

($\Rightarrow_{\mathsf{GE}}$): We need to show:

$$\mathcal{L}_{\mathrm{LJ}}{}^{\circ} \vdash_{\mathrm{I}} \forall A{:}\mathit{form}.\, \forall B{:}\mathit{form}.\, \lceil A \Rightarrow B\rceil \supset (\lceil A\rceil \supset \lfloor B\rfloor)$$

Let A and B be given and assume $\lceil A \Rightarrow B\rceil$. Using Cut (i.e., $\forall C{:}\mathit{form}.\, \lceil C\rceil \supset \lfloor C\rfloor$) with $C \equiv A \Rightarrow B$ we get:

$$\lfloor A \Rightarrow B\rfloor$$

and the result follows from $\Rightarrow_L$ (i.e., $\lfloor A \Rightarrow B\rfloor \supset (\lceil A\rceil \supset \lfloor B\rfloor)$).

□

Proposition 6.4.3. *Let $\Gamma \cup \{C\}$ be a set of object formulas, and let LJ^{f} be the proof system LJ without the cut rule, then:*

$$\Gamma \vdash_{\mathrm{LJ^f}} C \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{GE}} C$$

Proof. We use the same method as the in former proofs in this section, strictly speaking we have not proved completeness for $\mathrm{LJ^f}$, but completeness is proven in exactly the same way as for LJ. When the cut rule is removed, the result follows, like for LJ and NJ. We show a single case:

($\Rightarrow_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{GE}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \supset \lfloor B \rfloor)$$

So assume $\lfloor A \Rightarrow B \rfloor$ by (I) we have that:

$$\lceil A \Rightarrow B \rceil$$

and the result follows from ($\Rightarrow_{\mathsf{GE}}$).

□

As a side remark note that because LJ satisfies cut elimination , we have that provability in GE is equivalent to provability in LJ. But in this work we are more interested in the method than the (well-known) result.

For GEA we can prove the full correspondence:

Proposition 6.4.4. *Let $\Gamma \cup \{C\}$ be a set of object formulas then:*

$$\Gamma \vdash_{\mathrm{LJ}} C \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{GEA}} C\uparrow$$

Proof. Again we use the same method, and the only rules that are different are $\Rightarrow_{\mathsf{GE}}$, and $\forall_{\mathsf{GE}}$, these are clearly equivalent to the corresponding LJ rules. □

Last, we look at how to connect the two versions of sequent calculus LJ and LJ'.

Proposition 6.4.5. *Let $\Gamma \cup \{C\}$ be a set of formulas then:*

$$\Gamma \vdash_{\mathrm{LJ}} C \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{LJ'}} C$$

Proof. This proof is different from the other proofs of relative completeness, because instead of going all the way to intuitionistic logic, we stay in LJF and prove the correspondence there. The reason why we cannot use equivalence between the rules is because the encoded sequents are of a different format and therefore not directly related.

So using Propositions 6.3.1 and 6.3.6, what we wish to show in one direction is:

$$\text{If} \quad [\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil] \quad \text{then} \quad [\mathcal{L}_{\mathrm{LJ'}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil]$$

and to go in the other direction we generalize slightly, and want to show:

$$\begin{aligned} \text{If} \quad [\mathcal{L}_{\mathrm{LJ'}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil] \quad \text{then} \quad [\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil] \\ \text{If} \quad [\mathcal{L}_{\mathrm{LJ'}}, \lfloor \Gamma \rfloor] \longrightarrow [\mathbf{empty}] \quad \text{then} \quad [\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil \bot \rceil] \end{aligned}$$

The first direction is proved by induction on the focused derivations, and the second direction is proved by mutual induction on focused derivations. We note that the only

way the sequents can be provable is by focusing on one of the formulas in $\mathcal{L}_{\mathrm{LJ}}$, so those are the only cases we consider.

We only show the cases where $\bot$ is involved, as the rest follows straightforwardly and are very similar to what one would do, if one was working with the derivations in the object-level systems directly. First, we consider the first direction.

(Focus on ($\bot_{\mathsf{L}}$)):

The derivation on the left must be (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$):

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor \bot \rfloor} \!\!\rightarrow} \mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{false}} [\lceil C \rceil]} \mathsf{R_L}, \mathbf{false_L}}{[\mathcal{K}] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{false}} [\lceil C \rceil]} \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}$$

With $\mathcal{K}' = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$ we construct the needed derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\mathcal{K}'] -_{\lfloor \bot \rfloor} \!\!\rightarrow} \mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}'] \xrightarrow{\mathbf{empty}} \mathbf{empty}} \mathsf{I_L}}{[\mathcal{K}'] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{empty}} [\mathbf{empty}]} \supset_{\mathsf{L}}}{[\mathcal{K}'] \longrightarrow [\mathbf{empty}]} \mathsf{L_F} \qquad \dfrac{}{[\mathcal{K}'] \xrightarrow{\lceil C \rceil} [\lceil C \rceil]} \mathsf{I_L}}{[\mathcal{K}'] -_{\mathbf{empty}} \!\!\rightarrow} \mathsf{R_R}, []_{\mathsf{R}}}{[\mathcal{K}'] \xrightarrow{\mathbf{empty} \supset \lceil C \rceil} [\lceil C \rceil]} \supset_{\mathsf{L}}}{[\mathcal{K}'] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}, \forall_{\mathsf{L}}$$

Next we consider the second direction.

(Focus on ($\bot_{\mathsf{L}}$)):

The derivation on the left must be (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$):

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] -_{\lfloor \bot \rfloor} \!\!\rightarrow} \mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\mathbf{empty}} [\mathbf{empty}]} \mathsf{I_L}}{[\mathcal{K}] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{empty}} [\mathbf{empty}]} \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\mathbf{empty}]} \mathsf{L_F}$$

With $\mathcal{K}' = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor \bot \rfloor\}$ we construct the needed derivation:

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}'] -_{\lfloor \bot \rfloor} \!\!\rightarrow} \mathsf{I_R} \qquad \dfrac{}{[\mathcal{K}'] \xrightarrow{\mathbf{false}} [\lceil \bot \rceil]} \mathsf{R_L}, \mathbf{false_L}}{[\mathcal{K}'] \xrightarrow{\lfloor \bot \rfloor \supset \mathbf{false}} [\lceil \bot \rceil]} \supset_{\mathsf{L}}}{[\mathcal{K}'] \longrightarrow [\lceil \bot \rceil]} \mathsf{L_F}$$

(Focus on ($\mathsf{W_R}$)):

The derivation on the left must be (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}'} \cup \lfloor \Gamma \rfloor$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] \longrightarrow [\mathbf{empty}]} \mathsf{R_R}, []_{\mathsf{R}}}{[\mathcal{K}] -_{\mathbf{empty}} \!\!\rightarrow} \qquad \dfrac{}{[\mathcal{K}] \xrightarrow{\lceil C \rceil} [\lceil C \rceil]} \mathsf{I_L}}{[\mathcal{K}] \xrightarrow{\mathbf{empty} \supset \lceil C \rceil} [\lceil C \rceil]} \supset_{\mathsf{L}}}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]} \mathsf{L_F}, \forall_{\mathsf{L}}$$

With $\mathcal{K}' = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor$ we get by IH:

$$[\mathcal{K}'] \longrightarrow [\lceil \bot \rceil]$$

from which the result follows by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}'] \longrightarrow [\lceil \bot \rceil]}\,\mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}}}{[\mathcal{K}'] -_{\lceil \bot \rceil}\!\rightarrow}\qquad \dfrac{\dfrac{\dfrac{}{[\mathcal{K}', \lfloor \bot \rfloor] -_{\lfloor \bot \rfloor}\!\rightarrow}\,\mathsf{I}_{\mathsf{R}} \qquad \dfrac{}{[\mathcal{K}', \lfloor \bot \rfloor] \xrightarrow{\lceil C \rceil} [\lceil C \rceil]}\,\mathsf{I}_{\mathsf{L}}}{[\mathcal{K}', \lfloor \bot \rfloor] \longrightarrow [\lceil C \rceil]}\,\mathsf{L}_{\mathsf{F}}, \supset_{\mathsf{L}}}{[\mathcal{K}'] \xrightarrow{\lfloor \bot \rfloor} [\lceil C \rceil]}\,\mathsf{R}_{\mathsf{L}}, []_{\mathsf{L}}}{[\mathcal{K}'] \xrightarrow{\lceil \bot \rceil \supset \lfloor \bot \rfloor} [\lceil C \rceil]}\,\supset_{\mathsf{L}}}{[\mathcal{K}'] \longrightarrow [\lceil C \rceil]}\,\mathsf{L}_{\mathsf{F}}, \forall_{\mathsf{L}}$$

□

6.4.2 Classical systems

In this section we consider the classical systems. Because only LK deals with quantifiers, $\bot$ and $\top$, we shall restrict our view to propositional formulas without $\bot$ and $\top$. So in the following, Γ, Δ and C will represent propositional formulas without $\bot$ and $\top$.

Another remark is that the LK does not have an explicit negation, and therefore we must use a transformation to relate the later systems to LK. Because of the absence of negation, we cannot use a proof of equivalence between the rules like for the intuitionistic cases. So instead we introduce LK with negation (called LK$\neg$), prove that LK$\neg$ is equivalent to LK and then relate LK$\neg$ to FD, KE and AC.

The rules for LK$\neg$ are the same as for the propositional part of LK except $\bot_{\mathsf{L}}$ and $\top_{\mathsf{R}}$ plus the following two rules:

$$\frac{\Gamma, \neg A \vdash_{\mathrm{LK}\neg} A, \Delta}{\Gamma, \neg A \vdash_{\mathrm{LK}\neg} \Delta}\,\neg_{\mathsf{L}} \qquad \frac{\Gamma, A \vdash_{\mathrm{LK}\neg} \neg A, \Delta}{\Gamma \vdash_{\mathrm{LK}\neg} \neg A, \Delta}\,\neg_{\mathsf{R}}$$

For the encoding, we get a complete encoding of $\mathcal{L}_{\mathrm{LK}\neg}$ by adding the following (and removing the $\bot, \top, \forall, \exists$ part):

$$(\neg_L) \quad \lfloor \neg A \rfloor \supset \lceil A \rceil \qquad (\neg_R) \quad \lceil \neg A \rceil \supset \lfloor A \rfloor$$

For the correspondence, the function φ is defined as in the Nigam and Miller paper as:

$$\begin{aligned} \varphi(A) &= A && A \text{ atomic} \\ \varphi(\neg C) &= \varphi(C) \Rightarrow \bot \\ \varphi(C_1 \star C_2) &= \varphi(C_1) \star \varphi(C_2) && \star \in \{\Rightarrow, \wedge, \vee\} \end{aligned}$$

φ is extended to sets in the obvious way.

Using φ we can prove correspondence between LK and LK$\neg$:

Lemma 6.4.6. *Let Γ and Δ be sets of object formulas with negation then:*

$$\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(\Delta) \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{LK}\neg} \Delta$$

Proof. As the rules for LK and LK$\neg$ are almost the same, the only things to prove are the cases with the negation, which can be on either the left or the right side. The proof goes by induction on the height of the derivations.

For the "only if"-direction the interesting cases are $\Rightarrow_{\mathsf{L}}$ and $\Rightarrow_{\mathsf{R}}$ where the principal formulas are $A \Rightarrow \bot$.

($\Rightarrow_{\mathsf{L}}$):

$$\frac{\varphi(\Gamma), \varphi(A) \Rightarrow \bot \vdash_{\mathrm{LK}} \varphi(A), \varphi(\Delta) \qquad \varphi(\Gamma), \varphi(A) \Rightarrow \bot, \bot \vdash_{\mathrm{LK}} \varphi(\Delta)}{\varphi(\Gamma), \varphi(A) \Rightarrow \bot \vdash_{\mathrm{LK}} \varphi(\Delta)}$$

By IH we have that:

$$\Gamma, \neg A \vdash_{\mathrm{LK}\neg} A, \Delta$$

which gives the result by the $\neg_{\mathsf{L}}$ rule.

($\Rightarrow_{\mathsf{R}}$):

$$\frac{\varphi(\Gamma), \varphi(A) \vdash_{\mathrm{LK}} \bot, \varphi(A) \Rightarrow \bot, \varphi(\Delta)}{\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(A) \Rightarrow \bot, \varphi(\Delta)}$$

We can then obtain a proof of the same height of the following (removing the $\bot$ on the right - seen by straightforward induction):

$$\varphi(\Gamma), \varphi(A) \vdash_{\mathrm{LK}} \varphi(A) \Rightarrow \bot, \varphi(\Delta)$$

and then by IH we have that:

$$\Gamma, A \vdash_{\mathrm{LK}\neg} \neg A, \Delta$$

which gives the result by the $\neg_{\mathsf{R}}$ rule.

For the "if"-direction we consider the cases $\neg_{\mathsf{L}}$ and $\neg_{\mathsf{R}}$:

($\neg_{\mathsf{L}}$):

$$\frac{\Gamma, \neg A \vdash_{\mathrm{LK}\neg} A, \Delta}{\Gamma, \neg A \vdash_{\mathrm{LK}\neg} \Delta}\ \neg_{\mathsf{L}}$$

By IH we have that:

$$\varphi(\Gamma), \varphi(A) \Rightarrow \bot \vdash_{\mathrm{LK}} \varphi(A), \varphi(\Delta)$$

and the result follows from $\Rightarrow_{\mathsf{L}}$ and $\bot_{\mathsf{L}}$.

($\neg_{\mathsf{R}}$):

$$\frac{\Gamma, A \vdash_{\mathrm{LK}\neg} \neg A, \Delta}{\Gamma \vdash_{\mathrm{LK}\neg} \neg A, \Delta}\ \neg_{\mathsf{R}}$$

By IH we have that:

$$\varphi(\Gamma), \varphi(A) \vdash_{\mathrm{LK}} \varphi(A) \Rightarrow \bot, \varphi(\Delta)$$

and the result follows from the following derivation:

$$\frac{\dfrac{\dfrac{}{\varphi(\Gamma), \varphi(A) \vdash_{\mathrm{LK}} \varphi(A), \bot, \varphi(A) \Rightarrow \bot, \varphi(\Delta)}}{\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(A), \varphi(A) \Rightarrow \bot, \varphi(\Delta)} \qquad \varphi(\Gamma), \varphi(A) \vdash_{\mathrm{LK}} \varphi(A) \Rightarrow \bot, \varphi(\Delta)}{\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(A) \Rightarrow \bot, \varphi(\Delta)}$$

□

Using LK$\neg$ in the proofs, we can prove the correspondence from LK to FD, KE and AC. For FD we run into the same problem as for GE, namely that we can only prove one direction of the equivalence because of the missing cut rule. If we remove the cut rule from LK, we can prove the correspondence in the other way (note that we use the cut rule in the first direction, so we do not get a full equivalence for the cut-free LK).

Proposition 6.4.7. *Let* Γ *and* Δ *be sets of object formulas with negation then:*

$$\textit{If} \quad \Gamma \vdash_{\mathrm{FD}} \Delta \quad \textit{then} \quad \varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(\Delta)$$

Proof. We use that provability in LK is equivalent to provability in LK$\neg$ so we need to prove that:

$$\text{If} \quad \Gamma \vdash_{\mathrm{FD}} \Delta \quad \text{then} \quad \Gamma \vdash_{\mathrm{LK}\neg} \Delta$$

which we prove by proving that (using the corresponding completeness propositions):

$$\forall F \in \mathcal{L}_{\mathrm{FD}}{}^{\circ} \quad \mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} F$$

($\Rightarrow_{\mathsf{E}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lceil A \Rightarrow B \rceil \vee \lceil A \rceil \vee \lfloor B \rfloor$$

By using (Cut) for LK$\neg$ we get that:

$$\lfloor A \Rightarrow B \rfloor \vee \lceil A \Rightarrow B \rceil$$

In the second disjunct, we are done so assuming the first disjunct, we can then use ($\Rightarrow_{\mathsf{L}}$) to prove that:

$$\lceil A \rceil \supset \lfloor B \rfloor \tag{6.1}$$

and then we can use (Cut) again to show that

$$\lfloor A \rfloor \vee \lceil A \rceil$$

again we are done in the second disjunct, and in the first disjunct, the result follows from the implication in (6.1).

($\Rightarrow_{\mathsf{I1}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \vee \lfloor A \rfloor$$

By using (Cut) for LK$\neg$ we get that:

$$\lfloor A \Rightarrow B \rfloor \vee \lceil A \Rightarrow B \rceil$$

In the first disjunct we are done, so assuming the second disjunct we can then use ($\Rightarrow_{\mathsf{R}}$) to prove that:

$$\lfloor A \rfloor \wedge \lceil B \rceil$$

which proves the result.

($\neg_{\mathsf{I1}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \lfloor \neg A \rfloor \vee \lfloor A \rfloor$$

By using (Cut) for LK¬ we get that:

$$\lfloor \neg A \rfloor \vee \lceil \neg A \rceil$$

In the first disjunct we are done, so assuming the second disjunct we can then use ($\neg_{\mathsf{R}}$) to prove that:

$$\lfloor A \rfloor$$

which proves the original proposition.

□

Proposition 6.4.8. *Let* Γ *and* Δ *be sets of object formulas with negation, and let* LK^{f} *be LK without the cut rule, then:*

$$\textit{If} \quad \varphi(\Gamma) \vdash_{\mathrm{LK^f}} \varphi(\Delta) \quad \textit{then} \quad \Gamma \vdash_{\mathrm{FD}} \Delta$$

Proof. We take the detour through $\mathrm{LK^f}\neg$ (LK¬ without cut) and into intuitionistic logic, so we need to prove that:

$$\forall F \in \mathcal{L}_{\mathrm{LK^f}\neg}{}^{\circ} \quad \mathcal{L}_{\mathrm{FD}}{}^{\circ} \vdash_{\mathrm{I}} F$$

Again we show a couple of cases:

($\Rightarrow_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{FD}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \vee \lfloor B \rfloor)$$

Assume $\lfloor A \Rightarrow B \rfloor$ by ($\Rightarrow_{\mathsf{E}}$) we get that:

$$\lceil A \Rightarrow B \rceil \vee \lceil A \rceil \vee \lfloor B \rfloor$$

In the first disjunct we can use the assumption and (I) to conclude **false**, and in the second disjunction we are done.

($\Rightarrow_{\mathsf{R}}$): We need to show:

$$\mathcal{L}_{\mathrm{FD}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lceil A \Rightarrow B \rceil \supset (\lfloor A \rfloor \wedge \lceil B \rceil)$$

Assume $\lceil A \Rightarrow B \rceil$ by ($\Rightarrow_{\mathsf{I1}}$) and ($\Rightarrow_{\mathsf{I2}}$) we get that:

$$\lfloor A \Rightarrow B \rfloor \vee \lfloor A \rfloor$$
$$\lfloor A \Rightarrow B \rfloor \vee \lceil B \rceil$$

If we in either formula have $\lfloor A \Rightarrow B \rfloor$, then we can use (I) again to conclude **false**. So we must have both $\lfloor A \rfloor$ and $\lceil B \rceil$, and we are done.

($\neg_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{FD}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \lfloor \neg A \rfloor \supset \lceil A \rceil$$

Assume $\lfloor \neg A \rfloor$ by ($\neg_{\mathsf{I2}}$) we get that:

$$\lceil \neg A \rceil \vee \lceil A \rceil$$

and we can either conclude **false** from (I) or the conclusion, so we are done.

□

Again, because of cut elimination for LK, we have that provability in FD is equivalent to provability in LK.

For KE and AC, we can prove the full equivalence.

Proposition 6.4.9. *Let* Γ *and* Δ *be sets of object formulas with negation then:*

$$\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(\Delta) \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{KE}} \Delta$$

Proof. As for FD we use LK$\neg$ and intuitionistic logic to prove equivalence, which means we must show:

$$\forall F \in \mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \quad \mathcal{L}_{\mathrm{KE}}{}^{\circ} \vdash_{\mathrm{I}} F$$
$$\forall F \in \mathcal{L}_{\mathrm{KE}}{}^{\circ} \quad \mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} F$$

We first consider the first direction, and show a single case:

($\Rightarrow_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{KE}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \vee \lfloor B \rfloor)$$

Assume $\lfloor A \Rightarrow B \rfloor$ by using ($\Rightarrow_{\mathsf{L1}}$) we get that:

$$\lfloor A \rfloor \supset \lfloor B \rfloor \tag{6.2}$$

By (Cut) we get that:

$$\lfloor A \rfloor \vee \lceil A \rceil$$

If the first disjunct is the case, the result follows from (6.2); if the second disjunct is the case, the result follows immediately.

Next we consider the second direction, and show a single case:

($\Rightarrow_{\mathsf{L1}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset (\lfloor A \rfloor \supset \lfloor B \rfloor)$$

Assume $\lfloor A \Rightarrow B \rfloor$ and $\lfloor A \rfloor$ by using ($\Rightarrow_{\mathsf{L}}$) on the first, we get that:

$$\lceil A \rceil \vee \lfloor B \rfloor$$

If the first disjunct is the case, then we can derive **false** from (I), for the second disjunct the result follows immediately. □

Proposition 6.4.10. *Let* Γ *and* Δ *be sets of object formulas with negation then:*

$$\varphi(\Gamma) \vdash_{\mathrm{LK}} \varphi(\Delta) \quad \textit{if and only if} \quad \Gamma \vdash_{\mathrm{AC}} \Delta$$

Proof. This proofs is similar all the other proofs in this section, and again it comes down to showing that:

$$\forall F \in \mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \quad \mathcal{L}_{\mathrm{AC}}{}^{\circ} \vdash_{\mathrm{I}} F$$
$$\forall F \in \mathcal{L}_{\mathrm{AC}}{}^{\circ} \quad \mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} F$$

We first consider the first direction, and show a couple of cases:

($\Rightarrow_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{AC}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset (\lceil A \rceil \vee \lfloor B \rfloor)$$

Assume $\lfloor A \Rightarrow B \rfloor$ by using ($\Rightarrow_{\mathsf{L}}$), we get that:

$$(\lfloor A \rfloor \wedge \lceil B \rceil) \supset \mathbf{false} \tag{6.3}$$

By (Cut) we get that:

$$\lfloor A \rfloor \vee \lceil A \rceil$$

If the second disjunct is the case, the result follows immediately. So assume $\lceil A \rceil$, by using (Cut) again, we get that:

$$\lfloor B \rfloor \vee \lceil B \rceil$$

If the first disjunct is the case, the result follows immediately. So by assuming $\lceil B \rceil$ we can derive **false** from (6.3).

($\Rightarrow_{\mathsf{R}}$): We need to show:

$$\mathcal{L}_{\mathrm{AC}}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lceil A \Rightarrow B \rceil \supset (\lfloor A \rfloor \wedge \lceil B \rceil)$$

Assume $\lceil A \Rightarrow B \rceil$ by using ($\Rightarrow_{\mathsf{R}}$) we get that:

$$(\lceil A \rceil \vee \lfloor B \rfloor) \supset \mathbf{false} \tag{6.4}$$

By (Cut) we get that:

$$\lfloor A \rfloor \vee \lceil A \rceil$$

If the second disjunct is the case, then we can derive **false** from (6.4). So assume $\lfloor A \rfloor$, by using (Cut) again, we get that:

$$\lfloor B \rfloor \vee \lceil B \rceil$$

If the first disjunct is the case, we can again derive **false** from (6.4). So by assuming $\lceil B \rceil$ we get the conclusion.

Next we consider the second direction, and show a couple of cases:

($\Rightarrow_{\mathsf{L}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lfloor A \Rightarrow B \rfloor \supset ((\lfloor A \rfloor \wedge \lceil B \rceil) \supset \mathbf{false})$$

Assume $\lfloor A \Rightarrow B \rfloor$, $\lfloor A \rfloor$ and $\lceil B \rceil$ by using ($\Rightarrow_{\mathsf{L}}$) we get that:

$$\lceil A \rceil \vee \lfloor B \rfloor$$

and for each disjunct we can prove **false** from an assumption and (I).

($\Rightarrow_{\mathsf{R}}$): We need to show:

$$\mathcal{L}_{\mathrm{LK}\neg}{}^{\circ} \vdash_{\mathrm{I}} \forall A \forall B \lceil A \Rightarrow B \rceil \supset ((\lceil A \rceil \vee \lfloor B \rfloor) \supset \mathbf{false})$$

Assume $\lceil A \Rightarrow B \rceil$ and $\lceil A \rceil \vee \lfloor B \rfloor$ by using ($\Rightarrow_{\mathsf{R}}$) we get that:

$$\lfloor A \rfloor \wedge \lceil B \rceil \tag{6.5}$$

and for each disjunct in the assumption we can prove **false** from a conjunct in (6.5) and (I).

□

6.4.3 Intuitionistic and classical systems

In this section we look at how to relate LJ to LK. The result is the well-known fact that intuitionistic provability implies classical provability:

Proposition 6.4.11. *Let $\Gamma \cup \{C\}$ be a set of object formulas then:*

$$\textit{If} \quad \Gamma \vdash_{\mathrm{LJ}} C \quad \textit{then} \quad \Gamma \vdash_{\mathrm{LK}} C$$

Proof. Like in the proof of relative completeness for LJ and LJ', we use LJF directly and prove correspondence for focused derivations.

Using Propositions 6.3.1 and 6.3.2, what we need to show is (notice that there are different polarities on each side):

$$\text{If} \quad [\mathcal{L}_{\mathrm{LJ}}, \lfloor \Gamma \rfloor] \longrightarrow [\lceil C \rceil] \quad \text{then} \quad [\mathcal{L}_{\mathrm{LK}}, \lfloor \Gamma \rfloor, \lceil C \rceil] \longrightarrow [\mathbf{false}]$$

which we prove by induction on the focused derivation. As for LJ and LJ', the only way the sequent, can be provable is by focusing on one of the formulas in $\mathcal{L}_{\mathrm{LJ}}$, so those are the only cases we consider.

(Focus on $(\Rightarrow_{\mathsf{L}})$):

The derivation on the left must be (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor\}$):

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lfloor A_1 \Rightarrow A_2 \rfloor \rightarrow}\,\mathsf{I}_{\mathsf{R}} \quad \dfrac{[\mathcal{K}] \longrightarrow [\lceil A_1 \rceil]}{[\mathcal{K}] - \lceil A_1 \rceil \rightarrow}\,\mathsf{R}_{\mathsf{R}}, []_{\mathsf{R}} \quad \dfrac{[\mathcal{K}, \lfloor A_2 \rfloor] \longrightarrow [\lceil C \rceil]}{[\mathcal{K}] \xrightarrow{\lfloor A_2 \rfloor} [\lceil C \rceil]}\,\mathsf{R}_{\mathsf{L}}, []_{\mathsf{L}}}{}\,2 \times \supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \supset \lfloor A_2 \rfloor)} [\lceil C \rceil]}{[\mathcal{K}] \longrightarrow [\lceil C \rceil]}\,\mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}$$

Now let $\mathcal{K}'$ be $\mathcal{L}_{\mathrm{LK}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A_1 \Rightarrow A_2 \rfloor, \lceil C \rceil\}$.

By IH and weakening (in the second case) we get that:

$$[\mathcal{K}', \lceil A_1 \rceil] \longrightarrow [\mathbf{false}]$$
$$[\mathcal{K}', \lfloor A_2 \rfloor] \longrightarrow [\mathbf{false}]$$

We can now construct the needed derivation:

$$\dfrac{\dfrac{}{[\mathcal{K}'] - \lfloor A_1 \Rightarrow A_2 \rfloor \rightarrow}\,\mathsf{I}_{\mathsf{R}} \quad \dfrac{\dfrac{\dfrac{[\mathcal{K}', \lceil A_1 \rceil] \longrightarrow [\mathbf{false}]}{[\mathcal{K}'], \lceil A_1 \rceil \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}} \quad \dfrac{[\mathcal{K}', \lfloor A_2 \rfloor] \longrightarrow [\mathbf{false}]}{[\mathcal{K}'], \lfloor A_2 \rfloor \longrightarrow [\mathbf{false}]}\,[]_{\mathsf{L}}}{}\,\mathsf{R}_{\mathsf{L}}, \vee_{\mathsf{L}}}{[\mathcal{K}'] \xrightarrow{\lceil A_1 \rceil \vee \lfloor A_2 \rfloor} [\mathbf{false}]}\,\supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}'] \xrightarrow{\lfloor A_1 \Rightarrow A_2 \rfloor \supset (\lceil A_1 \rceil \vee \lfloor A_2 \rfloor)} [\mathbf{false}]}{[\mathcal{K}'] \longrightarrow [\mathbf{false}]}\,\mathsf{L}_{\mathsf{F}}, 2 \times \forall_{\mathsf{L}}}$$

(Focus on (I)):

The derivation on the left must be (with $\mathcal{K} = \mathcal{L}_{\mathrm{LJ}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A \rfloor\}$):

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}] - \lfloor A \rfloor \rightarrow}\,\mathsf{I}_{\mathsf{R}} \quad \dfrac{}{[\mathcal{K}] \xrightarrow{\lceil A \rceil} [\lceil A \rceil]}\,\mathsf{I}_{\mathsf{L}}}{}\,\supset_{\mathsf{L}}}{\dfrac{[\mathcal{K}] \xrightarrow{\lfloor A \rfloor \supset \lceil A \rceil} [\lceil A \rceil]}{[\mathcal{K}] \longrightarrow [\lceil A \rceil]}\,\mathsf{L}_{\mathsf{F}}, \forall_{\mathsf{L}}}$$

With $\mathcal{K}' = \mathcal{L}_{\mathrm{LK}} \cup \lfloor \Gamma \rfloor \cup \{\lfloor A \rfloor, \lceil A \rceil\}$) the needed derivation follows easily:

$$\dfrac{\dfrac{\dfrac{}{[\mathcal{K}'] - \lfloor A \rfloor \rightarrow}\,\mathsf{I_R} \quad \dfrac{}{[\mathcal{K}'] - \lceil A \rceil \rightarrow}\,\mathsf{I_R}}{[\mathcal{K}'] - \lfloor A \rfloor \wedge^{+} \lceil A \rceil \rightarrow}\,\wedge^{+}_{\mathsf{R}} \quad \dfrac{}{[\mathcal{K}'] \xrightarrow{\mathbf{false}} [\mathbf{false}]}\,\mathsf{R_L}, \mathbf{false_L}}{[\mathcal{K}'] \xrightarrow{(\lfloor A \rfloor \wedge^{+} \lceil A \rceil) \supset \mathbf{false}} [\mathbf{false}]}\,\supset_{\mathsf{L}}}{[\mathcal{K}'] \longrightarrow [\mathbf{false}]}\,\mathsf{L_F}, \forall_{\mathsf{L}}$$

□

6.5 Comparison of LJF and LLF

In this section we compare the use of LJF as a framework to the use of LLF as a framework. To compare the frameworks, we look at how easy or 'natural' the encodings are, how hard the completeness proofs are, and how well we can reason within the framework, for instance how hard the relative completeness theorems are.

First we look at the different encodings. Here it is clear that all the different intuitionistic systems have a natural encoding in LJF, which is also seen by the similar encodings they share. It seems very elegant that just by changing polarities we go from LJ to NJ. Except the GE encoding, only a few of the rule encodings ($\Rightarrow_{\mathsf{GE}}$ and $\forall_{\mathsf{GE}}$ for GEA, $\mathsf{W_R}$ and $\bot_{\mathsf{L}}$ for LJ') have a different encoding from the rest. Furthermore, the encodings are very simple, in the sense that the object-level connectives are mostly encoded using the corresponding meta-level connective, one exception here being the use of the atom **empty** in the encoding of LJ', which gives a less elegant encoding.

On the other hand, it seems that the encodings of the intuitionistic systems in LLF are more different to each other, than the encodings in LJF are, although the biggest differences are mostly in which exponentials are used. Two observations are interesting: the first is that the encodings of NJ, GE, GEA need an extra formula ($\lfloor \bot \rfloor^{\bot}$ or $\lceil \bot \rceil$), which we do not need in our encoding. This makes the LJF encoding a little closer to the object-level systems, but for NJ and GEA, the extra formula can also be used to describe normal-form proofs for the $\Rightarrow, \wedge, \forall, \bot, \top$ fragment. The second observation is that for LJ with empty right sides, the LLF encoding can still use the meta-level **false**, whereas the LJF encoding needs to create a new meta-level atom. The reason is that the LLF encoding can exploit linear logic's strict structural rules and prevent the $\bot_{\mathsf{L}}$ rule from being used on proper formulas.

The encodings of the classical systems in LJF are not as natural as the encodings of the intuitionistic systems. But for most of the systems, the encodings are still fairly simple and easy to come up with, although the encoding of AC is a little more complicated than the others. The specific encoding of sequents using **false** is also different from the intuitionistic sequents, which make the encoding of the rules different. For LLF the situation is almost the same as for intuitionistic logic, although the different classical encodings have more differences than the intuitionistic encodings have.

A very nice feature of LLF is that the different systems stem from the same generic system with the selective inclusion of structural rules. We have not found a way to create a generic system for LJF, as there is no control of the structural properties. This means that it would be impossible to encode nicely in LJF, linear systems or systems with multiple conclusions that do not allow weakening. As LLF uses linear logic as a base, these systems should be possible to encode in LLF.

The proofs of full completeness are very similar in the two systems. In both cases we have to look at the derivations and see how they match the rules. In that aspect, we have not met any difficulties in using LJF instead of LLF.

For the LJF encodings, the proofs of relative completeness between the intuitionistic systems and between the classical systems are mostly very easy. We can use proof by rule equivalence, except for the cases where one of the systems is missing a cut, rule, or when there are syntactic differences. This is good property, as the proofs of rule equivalence are shallow and suitable for automation. Compared to proving relative completeness using the object-level systems, it seems easier to prove relative completeness using the LJF approach, as the needed induction is hidden, whereas in the object-level proof we would need to use a new induction every time.

In the cases where the cut rule is missing in one system, we are not able to prove rule equivalence. It should still be possible to prove relative completeness in those cases, maybe following the work by Miller and Pimentel [60], but we have not further investigated that here.

Because the intuitionistic and classical systems are represented in a different way, we cannot prove rule equivalence for LJ and LJ' or rule implication for LJ and LK, and therefore we need to resort to induction over focused derivations, which takes the same or slightly more work, than an object-level proof would in this case.

In LLF the proofs of relative completeness usually use rule equivalence in one direction and induction on focused derivations in the other direction. This means that it is easier to prove relative completeness for some of the systems in the LJF setting. But when mixing the different encodings in LJF, like we do for LJ and LK, then rule implication cannot be used. In LLF the encodings of the sequents are the same, and we think that the proof in LLF could go through using rule implication, therefore making it simpler than the LJF proof.

In conclusion, it seems that while LLF is more general and allowing more systems to be represented in a more streamlined way, LJF is more 'specialized' in some ways, leading to, in some cases, easier proofs.

6.6 Summary and related work

In this work we have shown how to encode a wide range of different intuitionistic and classical proof systems, using the focused intuitionistic logic LJF. For each encoding, we prove that the open derivations of sequents in the object-level system are in bijective correspondence with the open derivations of the encoded sequents in LJF. The features of the focused proofs are crucial in reducing the variety of proofs in the meta-logic, and the strong equivalences are based on that.

Using the encodings, we have proven equivalences of provability between the different intuitionistic systems and between the different classical systems. Only in the cases were cut elimination or a syntactic transformation were needed, have we used a different approach.

The methods used in this work are not new and are based on recent work by Nigam and Miller [73]. Our works show that focusing is the important part, and that linearity is not needed, when the object-level systems have unrestricted structural rules. Furthermore, there may be advantages in using LJF, as some of the encodings seem easier, and some of the proofs seem shorter. Although, in other cases the more streamlined encodings in LLF may be preferred.

Related to the work by Nigam and Miller, and therefore also to this work, is the work by Miller and Pimentel [59, 60, 61, 82, 83], in which classical linear logic is used as a meta-logic for different proof systems including both representation of and reasoning about the object-level systems. Other people have been using other frameworks for representing different proof systems The approach by Harper, Honsell and Plotkin is more general allowing other derivation-based systems to be encoded (e.g, operational semantics). But they need a more complicated meta-system (based on dependant types), and do not have the easy equivalence proof, which we have shown here (the proofs by rule equivalence or implication).

Chapter 7

Focusing and contraction

Abstract

Existing focused proof systems for classical and intuitionistic logic allow contraction for exactly those formulas chosen for focus. For proof-search applications, contraction is undesirable, as we risk traversing the same path multiple times. We present here a contraction-free focused sequent calculus for classical propositional logic, called $\mathrm{LKF}_{\mathrm{CF}}$, which is a modification of the recently developed proof system LKF. We prove that our system is sound and complete with respect to LKF, and therefore it is also sound and complete with respect to propositional classical logic. LKF can be justified with a compilation into focused proofs for linear logic; in this work, we show how to do a similar compilation for $\mathrm{LKF}_{\mathrm{CF}}$, but into focused proofs for linear logic with subexponentials instead. We use two subexponentials, neither allowing contraction but one allowing weakening. We show how the focused proofs for linear logic can then simulate proofs in $\mathrm{LKF}_{\mathrm{CF}}$. Returning to proof-search, we end this work with a small experimental study, showing that a proof-search implementation based on $\mathrm{LKF}_{\mathrm{CF}}$ performs well compared to implementations based on $\mathsf{lean}T^{A}P$ and several variants and optimizations on LK and LKF.

7.1 Introduction

The sequent calculus is a general framework, well suited for expressing proof systems for different logics, but for a given sequent many different proofs might exist. In the sequent calculus, structural rules can always be applied, and introduction rules can be applied, when there is a matching formula. Furthermore, when rules are applied in a backwards fashion, there is nothing which connects the different rules; one rule might apply to one formula, while the next rule could apply to an entirely different formula independent from the first one, leading to a large number of proofs.

Focusing. *Focusing*, introduced by Andreoli for linear logic [5], is a method for limiting this freedom and, thereby, reducing the number of different proofs. A focused proof alternates between two *phases*. In the *negative* or *asynchronous* phase, all applicable rules are invertible and can therefore be applied in any order. In the *positive* or *synchronous* phase, all rules are applied only to a specific formula and various of its subformulas. Such a formula is called a *focus*. This limits the freedom in the proof to some specific introduction rules in the positive phase (e.g. which branch to prove in a disjunction) and to deciding upon which formula to focus. By reducing the number of proofs, focused proofs serve as a useful normal form for cut-free proofs. As cut-free proofs are used in

a variety of different applications in the foundations of computation, such normal forms can prove useful for applications in logic programming or in functional programming.

Although originally introduced for linear logic, focused proof systems have been developed for other logics as well. The flexible focused proof systems, LJF and LKF, for intuitionistic and classical logics respectively, recently presented by Liang and Miller [52], are examples of this. Each connective is classified as having positive or negative *polarity*, depending on whether its introduction and elimination rules apply in the positive or negative phase. An important aspect of focusing is that this classification extends to the atoms. For provability, the assignment can be arbitrary, but a given polarity assignment can have a great effect on the shape of the possible proofs. Consider trying to prove D from the assumptions: $A, A \supset B, (A \wedge B) \supset C, (A \wedge B \wedge C) \supset D$. If all atoms are given negative polarity, the only possible proof is a backwards chaining proof on D. On the other hand, if all atoms are given positive polarity, then a significantly smaller forward chaining proof on A exists. This allows a focused proof system to uniformly capture both bottom-up proofs (like traditionally done for Datalog) or top-down proofs (like in Prolog). In functional programming, Curien and Herbelin [19] note the connection between call-by-value and forward chaining, and between call-by-name and backward chaining.

The usefulness of focusing is further attested by several other applications. For proof-search, focusing can help by reducing the nondeterminism in the choice of which formula to decompose. Recent theorem provers by Chaudhuri and Pfenning [15], by McLaughlin and Pfenning [56], and by Baelde, Miller and Snow [7] are examples of this kind of application. Focused proof systems have also been used directly as a framework for hosting other proof systems. The reduced number of proofs in a focused proof system allows the object-level proof systems to be encoded at the maximal level of adequacy; the level of open derivations. Nigam and Miller [73] show how to use the focused proof system for linear logic, LLF, to host several different classical and intuitionistic proof systems, and the Author [37][1] shows how to use the focused intuitionistic proof system LJF to host the same systems.

Contraction. We now turn our attention to focused proof systems for classical and intuitionistic logic only. An interesting property of existing systems is that even though contraction-free systems exist, like the system by Dyckhoff [27] for intuitionistic logic and the system G3c [94] for classical logic, contraction is still used when focusing on a formula. The presence of contraction introduces a form of redundancy in the focused proofs, as the same formula can be considered for focus several times. Unfortunately, even in classical logic, this use of contraction is crucial to completeness. For backwards proof-search, contraction is also an undesirable property; when a formula is duplicated with contraction, proof-search risks having to consider the same formula and possibly apply the same rules again. Therefore, when contraction is present, expensive and non-logical methods like loop detection are needed for a proof search implementation. It is therefore valuable to ask, whether contraction can be removed.

In this work, we show how to remove contraction from the propositional part of the focused proof system for classical logic LKF, while still including the positive connectives. Even though the intuitionistic system LJF might have been a more interesting system from the viewpoint of removing contraction, we believe that starting with a system, where contraction is easier to remove in the unfocused case, gives insights on how

[1] Included in a reformatted and slightly edited form as Chapter 6.

to remove contraction from a focused system, and therefore also insights in how to approach a focused and contraction-free system for intuitionistic logic. Essentially, LKF is modified in two ways: when a formula is chosen for focus, it is removed from the context; and when a disjunction is decomposed in the positive phase, instead of discarding the non-chosen branch, it is added back into the context. The second modification is critical in order to retain completeness. We show that the new proof system, called $\mathrm{LKF}_{\mathrm{CF}}$ for contraction-free LKF, is sound and complete with respect to LKF or equivalently with respect to classical logic.

An interesting property of the developed system is that it allows a proof-theoretic notion of proof restart. Consider trying to prove a disjunction $A \vee B \vee C$: one needs to make a choice that might be wrong, for example consider choosing A. There are usually two ways of addressing such a wrong choice: 1. Backtrack, erase the proof built above the wrong choice, and then resume with the other choices. This is not captured by the proof theory, as the wrong choice is erased; 2. Do a restart: keep the proof that has been built so far, and simply reselect the formula $(A \vee B \vee C)$ and make a choice again. This is a more proof theoretical method, since this restart can be understood as a rule of inference [33]. The problem with the restart process is that there is no (proof theoretic) support for making sure that a different formula is selected (B or C). The system proposed here allows restart of only the remaining possible choices.

Linear logic compilation. LKF as well as LJF, the intuitionistic system used to derive LKF, can be justified with a compositional compilation into a focused proof system for linear logic. This allows an implementation of a focused proof system for linear logic to simulate both LKF and LJF faithfully. By a faithful simulation we mean that the proofs in the two systems are in bijective correspondence, so that nothing is lost from a proof theoretical perspective. The proof system, $\mathrm{LKF}_{\mathrm{CF}}$, that we develop here, is a bit unusual for sequent calculus systems, so in order to understand how unusual it is, we attempted a similar natural compilation into linear logic. This did not seem possible to do; so to further investigate the subject, we looked at a known extension to linear logic: adding *subexponentials* [71, 72]. With two additional operators, it is possible to achieve the compilation, allowing a focused proof system for linear logic with subexponentials to simulate $\mathrm{LKF}_{\mathrm{CF}}$ faithfully.

Proof-search implementation. To gain experience with using $\mathrm{LKF}_{\mathrm{CF}}$ for proof-search, we have constructed several simple λProlog [66] proof-search implementations, one based on Fitting's implementation [29] of $\mathsf{lean}\, T^A P$ [9], and the others based on variants and optimizations of LK, LKF and $\mathrm{LKF}_{\mathrm{CF}}$. We compare their relative performance by running them on a set of propositional classical theorems, originating from problems proposed by Pelletier [78].

Related work. Most related work have already been discussed above. Contraction elimination has been studied for several other systems as well. Negri [70] presents contraction-free systems for classical and intuitionistic logic extended to geometric theories. Hirschowitz et al. [44] present a system for linear logic, where contraction can be eliminated using a modified tensor rule. At the present time, the Author is not aware of any system that tries to remove contraction from focused systems. In the theorem-proving community, a great deal of work has gone into deciding satisfiability (and therefore also validity) of propositional classical logic, and therefore highly opti-

$$\frac{}{\mapsto [\Theta, \neg P], P}\ Id\ (P \text{ literal})$$

$$\frac{\mapsto [\Theta, P], P}{\vdash [\Theta, P]}\ Focus \qquad \frac{\vdash [\Theta, C], \Gamma}{\vdash [\Theta], \Gamma, C}\ [] \qquad \frac{\vdash [\Theta], N}{\mapsto [\Theta], N}\ Release$$

$$\frac{}{\vdash [\Theta], \Gamma, \mathcal{T}^-}\ Absurd \qquad \frac{\vdash [\Theta], \Gamma}{\vdash [\Theta], \Gamma, \mathcal{F}^-}\ Trivial \qquad \frac{}{\mapsto [\Theta], \mathcal{T}^+}\ \mathcal{T}^+$$

$$\frac{\vdash [\Theta], \Gamma, A, B}{\vdash [\Theta], \Gamma, A \vee^- B}\ \vee^- \qquad \frac{\vdash [\Theta], \Gamma, A \qquad \vdash [\Theta], \Gamma, B}{\vdash [\Theta], \Gamma, A \wedge^- B}\ \wedge^-$$

$$\frac{\mapsto [\Theta], A}{\mapsto [\Theta], A \vee^+ B}\ \vee_1^+ \qquad \frac{\mapsto [\Theta], B}{\mapsto [\Theta], A \vee^+ B}\ \vee_2^+ \qquad \frac{\mapsto [\Theta], A \qquad \mapsto [\Theta], B}{\mapsto [\Theta], A \wedge^+ B}\ \wedge^+$$

Figure 7.1: The propositional fragment of LKF. P is a positive formula, C is a literal or a positive formula. N is a negative formula.

mized provers based on (extensions to) DPLL [22, 23] exist, e.g. zChaff [85]. Our work is not meant to compete with these efficient provers. We focus on the proof theoretical aspects of proof-search; we try to act as a stepping stone into other non-classical logics, e.g. intuitionistic logic, where theorem proving is not so well-developed.

7.2 Focused classical logic

We start by briefly introducing LKF and then move on to LKF_{CF}.

7.2.1 The focused system LKF

A *literal* is either an atom or a negated atom. All formulas are assumed to be in *negation normal form* (NNF); meaning that there are no implications, and negation is only applied to atoms. The connectives in LKF come in two *polarizations*: the negative connectives are $\mathcal{F}^-, \mathcal{T}^-, \wedge^-, \vee^-$ and the positive connectives are $\mathcal{F}^+, \mathcal{T}^+, \wedge^+, \vee^+$. The negative and the positive version of the same connective has the same provability, but different (focused) proofs. The two versions of each connective stem from linear logic, where the negative connectives correspond to $\top, \bot, \&, \parr$ and the positive connectives correspond to $\mathbf{0}, \mathbf{1}, \otimes, \oplus$. Note that Liang and Miller [52] write $\neg\mathcal{T}$ and $\neg\mathcal{F}$ for $\mathcal{F}^-$ and $\mathcal{T}^-$, and no explicit polarity on the positive units. Atoms are arbitrarily assigned either negative or positive polarity; a negated atom is assigned the opposite polarity of the atom itself. We write A° for the transformation, which removes all polarizations from the connectives of A. LKF uses two types of sequents: the unfocused sequent $\vdash [\Theta], \Gamma$ corresponding to the negative phase, and the focused sequent $\mapsto [\Theta], A$ corresponding to the positive phase. For these sequents we have that Γ is a list of formulas, A is a single formula and Θ is a multiset of literals and positive formulas. In the following we will refer to Θ as the context, and to Γ as the working list. Figure 7.1 shows the propositional fragment of LKF.

Before moving on to LKF_{CF}, we note that a trivial way to remove contraction from LKF is to recognize that classical propositional logic can be organized, so that all connectives have invertible rules, corresponding to using negative polarity for all connectives. This organization comes at a cost, though; consider e.g. conjunction: the conjunction rule is invertible, because it copies everything. If conjunction is interpreted negatively,

then the copying rule must be applied immediately, but if conjunction is interpreted positively, one can choose in the proof, when to apply it (including not applying it at all). Consider trying to prove $(B \vee A) \vee \neg A$, where B is very large. If only positive connectives are used, we can ignore B and get a simple proof; on the other hand, if only negative connectives are used, one must produce a disjunctive normal form of the entire formula (which could be large depending on B). There are no short proofs for this formula using only negative connectives.

The focus rule of LKF expresses that when a given formula is chosen for focus, it is kept in the context for later refocus. An important insight is that the only reason for keeping it in the context is the rules for positive disjunction, as they risk throwing a needed subformula away. As an example, consider trying to prove $A \vee^+ \neg A$ where A is a positive atom. The only possible proof involves focusing on $A \vee^+ \neg A$ twice; once to put $\neg A$ into the context and then once to end the proof with focus on A. Realising that the troublesome rules for removing contraction are the rules for the positive disjunction is the starting point of our contraction-free system.

7.2.2 The contraction-free system $\mathrm{LKF}_{\mathrm{CF}}$

To simplify the presentation, we start out by considering only the positive fragment $(\mathcal{F}^+, \mathcal{T}^+, \vee^+, \wedge^+)$ of LKF. The first change is to remove the focused formula from the context, resulting in the following rule:

$$\frac{\mapsto [\Theta], P}{\vdash [\Theta, P]}$$

But, as mentioned previously, this rule change alone breaks completeness, because of the rule for the positive disjunction, so we realize that when following a path through one of the disjuncts, we do not need to restart with the full formula, but only with the non-chosen disjunct. So to regain completeness we add the non-taken branch to the context, by replacing $\vee_1^+$ with the following rule:

$$\frac{\mapsto [\Theta, B], A}{\mapsto [\Theta], A \vee^+ B}$$

These modifications are sufficient to get a sound and complete contraction-free system for the positive fragment of LKF. We only mention this result and then move on to the full fragment.

For the full fragment, we want to do something similar, but there is a problem with the positive disjunction rule as stated above. The non-taken branch could be a non-atomic negative formula, which would then be added to the context. A design goal of LKF was to make certain that non-atomic negative formulas are never allowed in the context: we feel that it is important to maintain this choice here as well. To avoid the problem, we change the system to take the negative formulas into account by decomposing them before adding them to the context. There are basically three ways of doing this, either decompose before continuing with the chosen branch, decompose in parallel with the chosen branch, or decompose after the chosen branch is finished. We have picked the first option, as we believe it gives the most natural system. Instead of introducing a new relation expressing the decomposition directly, we allow the negative phase to have a saved focus; this also handles branching, when negative conjunctions are encountered.

$$\frac{}{\mapsto [\Theta, \neg P], P}\ Id\ (P \text{ literal})$$

$$\frac{\mapsto [\Theta], A}{\vdash A, [\Theta]}\ Focus_1 \qquad \frac{\mapsto [\Theta], P}{\vdash \cdot, [\Theta, P]}\ Focus_2$$

$$\frac{\vdash \xi, [\Theta, C], \Gamma}{\vdash \xi, [\Theta], \Gamma, C}\ [] \qquad \frac{\vdash \cdot, [\Theta], N}{\mapsto [\Theta], N}\ Release$$

$$\frac{}{\vdash \xi, [\Theta], \Gamma, \mathcal{T}^-}\ Absurd \qquad \frac{\vdash \xi, [\Theta], \Gamma}{\vdash \xi, [\Theta], \Gamma, \mathcal{F}^-}\ Trivial \qquad \frac{}{\mapsto [\Theta], \mathcal{T}^+}\ \mathcal{T}^+$$

$$\frac{\vdash \xi, [\Theta], \Gamma, A, B}{\vdash \xi, [\Theta], \Gamma, A \vee^- B}\ \vee^- \qquad \frac{\vdash \xi, [\Theta], \Gamma, A \qquad \vdash \xi, [\Theta], \Gamma, B}{\vdash \xi, [\Theta], \Gamma, A \wedge^- B}\ \wedge^-$$

$$\frac{\vdash A, [\Theta], B}{\mapsto [\Theta], A \vee^+ B}\ \vee_1^+ \qquad \frac{\vdash B, [\Theta], A}{\mapsto [\Theta], A \vee^+ B}\ \vee_2^+ \qquad \frac{\mapsto [\Theta], A \qquad \mapsto [\Theta], B}{\mapsto [\Theta], A \wedge^+ B}\ \wedge^+$$

Figure 7.2: The contraction-free focused system $\mathrm{LKF}_{\mathrm{CF}}$. P is a positive formula, C is a positive formula or a literal. N is a negative formula.

Formalizing these ideas, we arrive at the proof system $\mathrm{LKF}_{\mathrm{CF}}$. The basic formalism is the same for LKF and $\mathrm{LKF}_{\mathrm{CF}}$, and that means we use the same definitions as given above for LKF. As in LKF, we have two types of sequents. The focused sequent $\mapsto [\Theta], A$ is very similar to the focused sequent in LKF, but the unfocused sequent $\vdash \xi, [\Theta], \Gamma$ is modified to include an extra zone ξ, which can be either empty $\cdot$ or a proper formula A. The meaning of this new sequent is that the formulas in Γ are decomposed using the rules for the negative fragment. When Γ becomes empty, if ξ is also empty, then a positive formula from Θ is picked for focus, but if ξ contains a proper formula, A (from a positive disjunction), then the only applicable rule continues with focus on A. This restriction of focus to a potential saved focus is crucial in ensuring that we treat the positive disjunction positively. The full system for $\mathrm{LKF}_{\mathrm{CF}}$ is given in Fig. 7.2.

The rules for the negative phase are similar to their LKF counterpart: a saved focus is kept until all negative formulas are decomposed, at which point it is refocused on. The focus rule of LKF is split into two rules, where one rule allows a saved focus to be resumed, and the other rule applies in the case where there is no saved focus. The rules for the positive phase are similar to the LKF rules. The only exception is the rules for the positive disjunction. In $\mathrm{LKF}_{\mathrm{CF}}$ the chosen formula is moved to the saved focus position, and the non-chosen disjunct is decomposed in the following negative phase.

The changed disjunction rule allows $\mathrm{LKF}_{\mathrm{CF}}$ to have less redundant proofs, when several parts of the same formula are used. Consider tying to prove $A \wedge^+ (F \vee^+ \neg F)$, where A is a big provable formula, and F is an atom. The only LKF proofs involve proving A twice, whereas the $\mathrm{LKF}_{\mathrm{CF}}$ proofs can get away with only proving A once.

One might be worried that the way we handle the positive disjunction gives it a negative flavour, as we keep both formulas after a choice is made. If the non-chosen disjunct is positive, it is immediately added to the context and focus is resumed; in this case the disjunction is strictly positive. On the other hand, if the non-chosen disjunct is negative, we have to decompose it, before it is added to the context, similar to how a negative disjunction is treated. This seems to be the price we pay, for handling the disjunction in the chosen way. An alternative could be to add a delay to a negative formula (e.g. transforming N to $N \wedge^+ \mathcal{T}^+$), and then put it directly into the context, instead of decomposing it.

Soundness of $\mathrm{LKF}_{\mathrm{CF}}$

LKF is sound and complete with respect to classical logic, regardless of the polarization for the connectives and the atoms. The proof of soundness and completeness can be found in the work by Liang and Miller [52]. Therefore, to prove that $\mathrm{LKF}_{\mathrm{CF}}$ is sound, with respect to LKF, we just need to prove that it is sound with respect to any classical system, e.g. LK, which can be proven by straightforward mutual induction:

Lemma 7.2.1. *For all A, Θ, ξ and Γ,*

1. *If $\mapsto [\Theta], A$ then $\vdash_{\mathrm{LK}} \Theta^\circ, A^\circ$.*
2. *If $\vdash \xi, [\Theta], \Gamma$ then $\vdash_{\mathrm{LK}} \xi^\circ, \Theta^\circ, \Gamma^\circ$.*

Soundness for $\mathrm{LKF}_{\mathrm{CF}}$ now follows from completeness of LKF with respect to LK.

Theorem 7.2.2 (Soundness of $\mathrm{LKF}_{\mathrm{CF}}$)**.** *If A is provable in $\mathrm{LKF}_{\mathrm{CF}}$, then A is provable in LKF.*

Completeness of $\mathrm{LKF}_{\mathrm{CF}}$

Completeness of $\mathrm{LKF}_{\mathrm{CF}}$ is not as simple as soundness. The main difficulty is proving that the focused formula is not needed in the context after a focus rule. But in order to prove such a lemma, we need to generalize the statement. The reason for this is that the subformulas of a focused formula might be negative, and therefore it is not the full formulas, but subformulas, that are present in the context. Furthermore, there might also be conjunctions in the subformulas, which may cause branching. To express subformulas and to handle the branching, we define a relation, $\uparrow$, between formulas and multisets of formulas as follows:

1. $C \uparrow C$, if C is positive or a negative literal.
2. $\mathcal{T}^- \uparrow \cdot$.
3. $(A \vee^- B) \uparrow \varphi_1, \varphi_2$, if $A \uparrow \varphi_1$ and $B \uparrow \varphi_2$.
4. $(A \wedge^- B) \uparrow \varphi$, if $A \uparrow \varphi$.
5. $(A \wedge^- B) \uparrow \varphi$, if $B \uparrow \varphi$.

This relation is derived from the similar relation in the work by Liang and Miller [53]. The idea behind the relation is that it captures exactly what happens in the negative phase. A couple of simple lemmas, which can be proven by structural induction, show how to use this relation:

Lemma 7.2.3. *If $\vdash \xi, [\Theta], \Gamma, A$ and $A \uparrow \varphi$ then $\vdash \xi, [\Theta, \varphi], \Gamma$. Furthermore, the derivation without A has strictly smaller height than the derivation with A.*

This lemma expresses that if we have a derivation of some sequent in the negative phase with A in the waiting list, then whenever $A \uparrow \varphi$, then there is a subderivation of the original derivation, where φ is in the context and A is not in the working list.

Lemma 7.2.4. *Given A, if we for all φ such that $A \uparrow \varphi$ have that $\vdash \xi, [\Theta, \varphi], \Gamma$ then $\vdash \xi, [\Theta], \Gamma, A$.*

This lemma expresses that $\uparrow$ captures all the subderivations needed to prove the given negative sequent.

The hardest parts of the main lemma (removing the focused formula from the context) are concerned with the positive conjunction and disjunction; below we prove two lemmas, which allow us to decompose those positive connectives inside the context. These lemmas follow from mutual induction on the derivations. The interesting cases are those where focus selects the given connective from the context; we show one of those cases:

Lemma 7.2.5. *For all A, B, C, Θ, ξ and Γ:*

1. *Suppose $\mapsto [\Theta, A \vee^+ B], C$. If $A \uparrow \varphi_1$ and $B \uparrow \varphi_2$ then $\mapsto [\Theta, \varphi_1, \varphi_2], C$.*
2. *Suppose $\vdash \xi, [\Theta, A \vee^+ B], \Gamma$. If $A \uparrow \varphi_1$ and $B \uparrow \varphi_2$ then $\vdash \xi, [\Theta, \varphi_1, \varphi_2], \Gamma$.*

Proof. Consider the case for 2., where the derivation has the following form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash A, [\Theta], B}}{\mapsto [\Theta], A \vee^+ B}}{\vdash \cdot, [\Theta, A \vee^+ B]}$$

Lemma 7.2.3 applied to the top derivation gives a derivation of $\vdash A, [\Theta, \varphi_2]$, which must end in $Focus_1$, so we get a derivation of:

$$\mapsto [\Theta, \varphi_2], A \ . \tag{7.1}$$

Now if A is a positive formula, then $\varphi_1 = A$ and the result follows from focus on A. If A is a negative formula, then focus is lost following (7.1), and the result follows from application of Lemma 7.2.3. The case where B is chosen is symmetric. □

Lemma 7.2.6. *For all A, B, C, Θ, ξ, and Γ:*

1. *If $\mapsto [\Theta, A \wedge^+ B], C$ and $A \uparrow \varphi_1$, then $\mapsto [\Theta, \varphi_1], C$.*
2. *If $\vdash \xi, [\Theta, A \wedge^+ B], \Gamma$ and $A \uparrow \varphi_1$, then $\vdash \xi, [\Theta, \varphi_1], \Gamma$.*
3. *If $\mapsto [\Theta, A \wedge^+ B], C$ and $B \uparrow \varphi_2$, then $\mapsto [\Theta, \varphi_2], C$.*
4. *If $\vdash \xi, [\Theta, A \wedge^+ B], \Gamma$ and $B \uparrow \varphi_2$, then $\vdash \xi, [\Theta, \varphi_2], \Gamma$.*

With these two lemmas, we can prove the main lemma.

Lemma 7.2.7. *Given A, if we for all φ such that $A \uparrow \varphi$ have that $\mapsto [\Theta, \varphi], A$ then $\mapsto [\Theta], A$.*

Proof. The proof is by structural induction on A. The interesting case is the positive disjunction, which we show here. Consider $A = B \vee^+ C$. By assumption we have that (there are two cases, here we consider one):

$$\dfrac{\vdash B, [\Theta, B \vee^+ C], C}{\mapsto [\Theta, B \vee^+ C], B \vee^+ C}$$

First, assume that $C \uparrow \varphi_2$ (we wish to use Lemma 7.2.4, and therefore prove that $\vdash B, [\Theta, \varphi_2]$). Using Lemma 7.2.3 we get $\vdash B, [\Theta, B \vee^+ C, \varphi_2]$, which could only have been derived by Focus_1, so we get a derivation of:

$$\mapsto [\Theta, B \vee^+ C, \varphi_2], B \ . \tag{7.2}$$

Now assume that $B \uparrow \varphi_1$ (here we wish to use the induction hypothesis, so we want to prove that $\mapsto [\Theta', \varphi_1, \varphi_2], B$). Now by Lemma 7.2.5 applied to (7.2), we get $\mapsto [\Theta, \varphi_1, \varphi_2, \varphi_2], B$ on which the induction hypothesis applies, and we get:

$$\mapsto [\Theta, \varphi_2, \varphi_2], B \ . \tag{7.3}$$

We now want to prove by inner mutual induction that for any $D \in \varphi_2$:

1. If $\vdash \xi, [\Theta', D, D], \Gamma$ then $\vdash \xi, [\Theta', D], \Gamma$.
2. If $\mapsto [\Theta', D, D], E$ then $\mapsto [\Theta', D], E$.

All cases, except the focus rule where D is selected for focus, apply the inner induction hypothesis straightforwardly. If D is selected for focus, we have the following derivation:

$$\frac{\mapsto [\Theta', D], D}{\vdash \cdot, [\Theta', D, D]}$$

Because D is positive, we have that $D \uparrow D$, and as D is a subformula of C or equal to C, the outer induction hypothesis applies, and we get $\mapsto [\Theta'], D$. The inner proof is then completed with Focus_2. We apply the result once for every formula in φ_2 to (7.3) and get:

$$\mapsto [\Theta, \varphi_2], B \ . \tag{7.4}$$

By applying Focus_1 and then Lemma 7.2.4 to (7.4) we get $\vdash B, [\Theta], C$, from which the result follows with $\vee_1^+$. □

By using this lemma in the focus rule case, it is now straightforward to prove completeness of $\mathrm{LKF}_{\mathrm{CF}}$ with respect to LKF, by mutual induction on the derivations.

Theorem 7.2.8 (Completeness of $\mathrm{LKF}_{\mathrm{CF}}$)**.** *For all* Θ, Γ *and* A*:*

1. *If* $\vdash_{\mathrm{LKF}} [\Theta], \Gamma$ *then* $\vdash \cdot, [\Theta], \Gamma$.
2. *If* $\mapsto_{\mathrm{LKF}} [\Theta], A$ *then* $\mapsto [\Theta], A$.

Having completeness with respect to LKF also means that $\mathrm{LKF}_{\mathrm{CF}}$ is complete with respect to propositional classical logic, regardless of the polarization of the atoms and the connectives.

7.3 Compilation into linear logic

When attempting a compilation of $\mathrm{LKF}_{\mathrm{CF}}$ into linear logic similar to the compilation of LJF by Liang and Miller [52], one is faced with two challenges: first, our context admits weakening but not contraction, so we cannot immediately use the unrestricted context of linear logic; secondly, we need some mechanism to ensure that if a formula is saved, it is selected for focus before the formulas in the context are selected. Fortunately, linear logic with subexponentials can be used to overcome both challenges.

7.3.1 Linear logic with subexponentials

The exponentials in linear logic are not canonical, meaning that if we have two sets of exponentials $(?^1, !^1, ?^2, !^2)$ with the same set of introduction rules, they cannot be proven equivalent. Danos et al. [21] proposed a linear logic system with the possibility of having several non-canonical exponentials. These exponentials may or may not admit weakening or contraction, so the exponential equivalences may not hold, and therefore Nigam and Miller [71, 72] call them subexponentials. The basic notion is a *subexponential signature* $\Sigma = \langle \mathcal{I}, \preceq, \mathcal{W}, \mathcal{C} \rangle$, where $\mathcal{I}$ is a set of subexponential indexes, $\preceq$ is a preorder relation over $\mathcal{I}$. The subexponentials indexed by $\mathcal{W} \subseteq \mathcal{I}$ admit weakening, the subexponentials indexed by $\mathcal{C} \subseteq \mathcal{I}$ admit contraction. All subexponentials admit the rule of dereliction (here $a \in \mathcal{I}$):

$$\frac{\vdash A, \Delta}{\vdash ?^a A, \Delta}$$

The subexponentials where $w \in \mathcal{W}$ admit weakning, and the subexponentials where $c \in \mathcal{C}$ admit contraction:

$$\frac{\vdash \Delta}{\vdash ?^w A, \Delta} \qquad \frac{\vdash ?^c A, ?^c A, \Delta}{\vdash ?^c A, \Delta}$$

The promotion rule is given for all subexponentials $a \in \mathcal{I}, x_i \in \mathcal{I}$:

$$\frac{\vdash ?^{x_1} A_1, \ldots, ?^{x_n} A_n, A}{\vdash ?^{x_1} A_1, \ldots, ?^{x_n} A_n, !^a A} \, (a \preceq x_1 \wedge \ldots \wedge a \preceq x_n)$$

For our compilation, we use the subexponential structure $\Sigma = \langle \{x, y\}, \preceq, \{y\}, \emptyset \rangle$ where $x \preceq y$. The ordering of x and y will be exploited to make sure that a potential saved focus is resumed before a new formula is chosen for focus.

7.3.2 Focused linear logic with subexponentials

Nigam [71] proposes a focused linear logic proof system with subexponentials, $SELLF_\Sigma$, with the proviso that $\mathcal{C} \subseteq \mathcal{W}$. We will use that system with the sequents specialized to our signature given above. Instead of using an indexed context for the question-mark prefixed formulas, we will use two exponential contexts, one for x and one for y; this will make the syntax closer to that of Andreoli's LLF. The basic properties are the same as for $SELLF_\Sigma$: we consider formulas in negation normal form; the following connectives are positive: $\mathbf{0}, \mathbf{1}, \otimes, \oplus, !$; the following connectives are negative: $\top, \bot, \&, \parr, ?$; and atoms are assigned an arbitrary polarity. For more details, we refer to the work by Nigam [71].

The system has two kinds of sequents, the focused sequents $\Xi : \Theta : \Delta \Downarrow A$, and the unfocused sequents $\Xi : \Theta : \Delta \Uparrow \Gamma$. The first zone is the subexponential zone for x, the second zone is the subexponential zone for y, and the third zone is the normal linear context. The focused sequent corresponds to the following sequent in unfocused linear logic: $\vdash ?^x \Xi, ?^y \Theta, \Delta, A$ and the unfocused sequent to the following: $\vdash ?^x \Xi, ?^y \Theta, \Delta, \Gamma$. The rules (specialized from the rules by Nigam [71]), are given in Fig. 7.3. The rules are as expected, for instance the $!^y$-rule requires the x-zone to be empty.

$$\frac{}{A_{\mathrm{p}}^{\perp} : \Theta : \cdot \Downarrow A_{\mathrm{p}}}\, I_x \qquad \frac{}{\cdot : \Theta, A_{\mathrm{p}}^{\perp} : \cdot \Downarrow A_{\mathrm{p}}}\, I_y \qquad \frac{}{\cdot : \Theta : A_{\mathrm{p}}^{\perp} \Downarrow A_{\mathrm{p}}}\, I_1$$

$$\frac{\Xi : \Theta : \Delta \Downarrow P}{\Xi, P : \Theta : \Delta \Uparrow \cdot}\, D_x \qquad \frac{\Xi : \Theta : \Delta \Downarrow P}{\Xi : \Theta, P : \Delta \Uparrow \cdot}\, D_x \qquad \frac{\Xi : \Theta : \Delta \Downarrow P}{\Xi : \Theta : \Delta, P \Uparrow \cdot}\, D_1$$

$$\frac{\Xi : \Theta : \Delta \Uparrow N}{\Xi : \Theta : \Delta \Downarrow N}\, R{\Downarrow} \qquad \frac{\Xi : \Theta : \Delta, S \Uparrow \Gamma}{\Xi : \Theta : \Delta \Uparrow \Gamma, S}\, R{\Uparrow}$$

$$\frac{}{\Xi : \Theta : \Delta \Uparrow \Gamma, \top}\, \top \qquad \frac{\Xi : \Theta : \Delta \Uparrow \Gamma}{\Xi : \Theta : \Delta \Uparrow \Gamma, \bot}\, \bot \qquad \frac{}{\cdot : \Theta : \cdot \Downarrow \mathbf{1}}\, \mathbf{1}$$

$$\frac{\Xi : \Theta : \Delta \Uparrow \Gamma, A \qquad \Xi : \Theta : \Delta \Uparrow \Gamma, B}{\Xi : \Theta : \Delta \Uparrow \Gamma, A \& B}\, \& \qquad \frac{\Xi : \Theta : \Delta \Uparrow \Gamma, A, B}{\Xi : \Theta : \Delta \Uparrow \Gamma, A \parr B}\, \parr$$

$$\frac{\Xi_1 : \Theta_1 : \Delta_1 \Downarrow A \qquad \Xi_2 : \Theta_2 : \Delta_2 \Downarrow B}{\Xi_1, \Xi_2 : \Theta_1, \Theta_2 : \Delta_1, \Delta_2 \Downarrow A \otimes B}\, \otimes \qquad \frac{\Xi : \Theta : \Delta \Downarrow A_i}{\Xi : \Theta : \Delta \Downarrow A_1 \oplus A_2}\, \oplus$$

$$\frac{\Xi, A : \Theta : \Delta \Uparrow \Gamma}{\Xi : \Theta : \Delta \Uparrow \Gamma, ?^x A}\, ?^x \qquad \frac{\Xi : \Theta, A : \Delta \Uparrow \Gamma}{\Xi : \Theta : \Delta \Uparrow \Gamma, ?^y A}\, ?^y$$

$$\frac{\Xi : \Theta : \cdot \Uparrow A}{\Xi : \Theta : \cdot \Downarrow !^x A}\, !^x \qquad \frac{\cdot : \Theta : \cdot \Uparrow A}{\cdot : \Theta : \cdot \Downarrow !^y A}\, !^y$$

Figure 7.3: A focused proof system for linear logic with the subexponentials x and y. $i \in \{1, 2\}$, A_{p} is a positive literal, P is not a negative literal, N is a negative formula and S is a literal or a positive formula.

7.3.3 Compiling $\mathrm{LKF}_{\mathrm{CF}}$

We now describe the compilation into a focused proof system for linear logic with subexponentials. We will use the y-zone to hold the context, as it admits weakening and not contraction, corresponding to the situation in $\mathrm{LKF}_{\mathrm{CF}}$. We will use the x-zone to hold the saved focus. The formulas in the context are translated with $!^y$ in front, thereby forcing the promotion rule to be applicable only when the x-zone is empty, meaning that there is no saved focus. The compilation is defined by three different translations on $\mathrm{LKF}_{\mathrm{CF}}$ formulas:

1. $N^{-1} = N$, if N is a literal.
2. $P^{-1} = !^y P^{+1}$.
3. $P^0 = ?^y !^y P^{+1}$.
4. $N^0 = ?^y N$, if N is a literal.
5. $(\mathcal{T}^-)^0 = \top$.
6. $(\mathcal{F}^-)^0 = \bot$.
7. $(A \wedge^- B)^0 = A^0 \& B^0$.
8. $(A \vee^- B)^0 = A^0 \parr B^0$.
9. $N^{+1} = N^0$.
10. $(\mathcal{F}^+)^{+1} = \mathbf{0}$.
11. $(\mathcal{T}^+)^{+1} = \mathbf{1}$.
12. $(A \wedge^+ B)^{+1} = A^{+1} \& B^{+1}$.
13. $(A \vee^+ B)^{+1} = (?^x !^x A^{+1} \parr B^0) \oplus (A^0 \parr ?^x !^x B^{+1})$.

In the translation N is negative, and P is positive . We note that these functions are well defined (if A is positive, then $\cdot^{+1}$ is only used on subformulas of A, and if A is negative, then $\cdot^0$ is only used on subformulas of A). Furthermore, we note that if N is negative, then N^0 is also negative.

The translation is mostly straightforward, except for a few places. We use a negative linear conjunction ($\&$) for the positive classical conjunction ($\wedge^+$), the reason being that

we do not want to split the context in the y-zone, which would happen, if we used the positive linear conjunction ($\otimes$). The reason why the tensor can be used for LJF is that the context of LJF is translated to the unrestricted context and therefore not split. This choice means that we briefly lose focus when encountering a translated positive conjunction. But this does not pose a problem, as the only way to continue a proof is to refocus on the negative conjunction, ensured by $!^y$ in the y-zone. One can prove this formally by looking on the polarity of the formula, as stated in the following lemma:

Lemma 7.3.1. *The focused proofs of* $\cdot : \Theta^{-1} : \cdot \Downarrow A^{+1}$ *and* $\cdot : \Theta^{-1} : \cdot \Uparrow A^{+1}$ *are in bijective correspondence.*

It is also relevant to note the translation of $\vee^+$. In order to get a correct encoding, we mix both the positive and the negative linear disjunction. The way to understand the translation is that we choose one of the disjuncts (with $\oplus$), but then we have to keep both formulas around (with $\parr$), because they are not present in the context anymore. Lastly, we use $?^x$ to make sure that the saved focus is put into the x-zone, and therefore will be picked for focus directly after the other part of the formula is decomposed. We can finally prove that our compilation successfully simulates $\mathrm{LKF}_{\mathrm{CF}}$.

Theorem 7.3.2 ($\mathrm{LKF}_{\mathrm{CF}}$ translation)**.** *There is a bijection between proofs of the following sequents:*

1. $\mapsto [\Theta], A$ *and* $\cdot : \Theta^{-1} : \cdot \Downarrow A^{+1}$.
2. $\vdash \xi, [\Theta], \Gamma$ *and* $!^x\xi^{+1} : \Theta^{-1} : \cdot \Uparrow \Gamma^0$.

Proof. By mutual induction on the derivations. We show two interesting cases:

1. Case:

$$\dfrac{\dfrac{\mapsto [\Theta], A}{\vdash A, [\Theta]}}{} \quad \longleftrightarrow \quad \dfrac{\dfrac{\cdot : \Theta^{-1} : \cdot \Uparrow A^{+1}}{\cdot : \Theta^{-1} : \cdot \Downarrow !^x A^{+1}}}{!^x A^{+1} : \Theta^{-1} : \cdot \Uparrow}$$

   The right derivation must look like that, because the $!^y$'s in Θ^{-1} prevent focus from picking them, when the second context is non-empty. The result follows from Lemma 7.3.1.

2. Case:

$$\dfrac{\dfrac{\vdash A, [\Theta], B}{\mapsto [\Theta], A \vee^+ B}}{} \quad \longleftrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{!^x A^{+1} : \Theta^{-1} : \cdot \Uparrow B^0}{\cdot : \Theta^{-1} : \cdot \Uparrow ?^x !^x A^{+1}, B^0}}{\cdot : \Theta^{-1} : \cdot \Uparrow ?^x !^x A^{+1} \parr B^0}}{\cdot : \Theta^{-1} : \cdot \Downarrow ?^x !^x A^{+1} \parr B^0}}{\cdot : \Theta^{-1} : \cdot \Downarrow (?^x !^x A^{+1} \parr B^0) \oplus (?^x !^x B^{+1} \parr A^0)}$$

□

Even though our theorem only relates (full) proofs, our compilation will, in principle, ensure a bijection on open derivations as well. The only places where the direct bijection fails, are when the linear logic proof system focuses on one of the formulas with a bang. But such a focus would fail immediately afterwards, thus ensuring the correspondence.

We conclude the section with a comment about the translation. We use a subexponential with weakening but not contraction, so one could consider, whether a dialect of affine logic could be used instead of full linear logic with subexponentials. The problem here is how to represent the saved focus of $\mathrm{LKF}_{\mathrm{CF}}$, for which we use the extra subexponential. But if one considered only the positive fragment, one would only need one subexponential, and we speculate that a dialect of affine logic might be enough for this fragment.

7.4 $\mathrm{LKF}_{\mathrm{CF}}$ and proof search

For a proof system like $\mathrm{LKF}_{\mathrm{CF}}$, it is relatively easy to do a naive implementation in a logical programming language like Prolog or λProlog [66]. We have therefore constructed several different provers and compared their relative performance on a set of benchmark problems. The set of benchmark problems is very important, as a biased set will generally benefit specific provers more. One method to get a non-biased set would be to look at some third-part library for test problems, like the TPTP problem library for automated theorem proving [93], and then choose a subset of the propositional problems. As the selection of problems could also be biased, and because we only build small naive theorem provers, we settle on a smaller problem set derived from the propositional part of the problems for theorem provers by Pelletier [78] instead.

The problems by Pelletier are given as general formulas using $\Rightarrow$ and $\Leftrightarrow$; we first convert these into negation normal form, and then, because the problems are too small to use as a proper benchmark, we generalize the problems by replacing the atoms in the formulas with composite formulas. In this study we consider four different composite formulas; two are based on a big disjunction of either different positive atoms or different negative atoms, and the other two are based on a big conjunction of different positive atoms or different negative atoms. We use a fixed size of the conjunction and the disjunction to prevent any prover from having an edge; in the experiment below, the size of those were 100 atoms. Some of the problems will be the same (e.g. Pelletier problem 6 and 7 have the same negation normal form), so we will only consider the different problems. In total we consider 52 different problems.

All the provers are implemented in λProlog version 2.0-b2; their code is available on the author's web page[2]. One is based on Fitting's implementation [29] of $\mathsf{lean}\, T^A P$ [9]; four other provers are based on versions of the contraction-free system G3c [94], depending on how early the initial rule is applied, or whether disjunctions or conjunctions are decomposed first; eight are based on LKF, with the variants based on whether disjunctions/conjunctions are polarized negatively or positively, and based on whether they apply a small optimization described below; finally, eight variants are based on $\mathrm{LKF}_{\mathrm{CF}}$, the variations being the same as for LKF.

Four of the LKF provers and four of the $\mathrm{LKF}_{\mathrm{CF}}$ provers apply an optimization, which allows the initial rule to be used in the negative phase, corresponding to adding the following rules to LKF and $\mathrm{LKF}_{\mathrm{CF}}$, respectively:

$$\frac{}{\vdash [\Theta, \neg L], \Gamma, L} \qquad \frac{}{\vdash \xi, [\Theta, \neg L], \Gamma, L}$$

where L is a literal.

All the provers have been run on a 2.13 MHz Intel dual core machine with 3 GB memory, and the results are summarized in Table 7.1. Solved is the number of problems

[2] http://www.diku.dk/hjemmesider/ansatte/starcke/

Table 7.1: Experimental study of different provers

Prover	Solved (10s)	Avg. (10s)	Solved (60s)	Avg. (60s)
$\mathrm{LKF}^{\mathrm{id*}}_{\mathrm{CF}}$ (+,–)	35 of 52	0.8s	41 of 52	3.2s
$\mathrm{LKF}^{\mathrm{id*}}$ (+,–)	33 of 52	1.3s	34 of 52	1.8s
$\mathrm{LKF}_{\mathrm{CF}}$ (+,–)	33 of 52	1.5s	34 of 52	2.2s
$\mathrm{LKF}_{\mathrm{CF}}$ (+,+)	27 of 52	1.6s	33 of 52	4.3s
$\mathrm{LKF}^{\mathrm{id*}}_{\mathrm{CF}}$ (+,+)	27 of 52	1.6s	33 of 52	4.4s
LK ($\vee$ first)	32 of 52	0.7s	32 of 52	0.7s
LK (eager initial)	28 of 52	0.9s	28 of 52	0.9s
$\mathsf{lean}\, T^A P$	24 of 52	2.8s	28 of 52	4.5s
LKF (+,–)	26 of 52	1.8s	27 of 52	2.4s
$\mathrm{LKF}^{\mathrm{id*}}$ (–,–)	22 of 52	2.6s	25 of 52	5.9s

solved with the given time limit for each problem. Avg. is the average running time for successfully proven problems. For the LKF and $\mathrm{LKF}_{\mathrm{CF}}$ provers, we assign polarity to the connectives uniformly, the first sign is the polarity for all conjunctions and the second sign is the polarity for all disjunctions; id* means that the optimization described above was used. Because of space limits, we only show the ten best provers ranked by the number of solved problems in 60 seconds.

When considering the results, we see that the implementations based on $\mathrm{LKF}_{\mathrm{CF}}$ perform well on our chosen set of benchmark problems. We will briefly mention a couple of details: we see that for both LKF and $\mathrm{LKF}_{\mathrm{CF}}$, a positive conjunction and a negative disjunction yield the best results; this might be because most of the benchmark problems contain disjunctions, where both branches are needed to get a proof, and therefore it is faster to just decompose the disjunction. For $\mathrm{LKF}_{\mathrm{CF}}$ it might be a bit surprising to learn that the negative disjunction is better than the (optimized) positive disjunction, but we think that this is due to the fact that the disjunctions in the benchmark problems are very wide, but not very deeply nested; meaning that it is more efficient to decompose the disjunction directly, although the $\mathrm{LKF}_{\mathrm{CF}}$ variant with positive disjunction still performs well. The formulas, where LKF and $\mathrm{LKF}_{\mathrm{CF}}$ have an advantage over the LK variants, seem to be the formulas, where big conjunctions can be put into the context and saved for later. Non-surprisingly, the advantage is smallest over the version where conjunctions are saved for last. When there are negative atoms, LKF and $\mathrm{LKF}_{\mathrm{CF}}$ have to refocus, so big conjunctions with negative atoms can give lesser performance for those focused systems, compared to the LK variants. Lastly, the $\mathsf{lean}\, T^A P$ implementation is very similar to our LK variant where the initial rule is applied eagerly, which can also be seen from the results; one difference is that the $\mathsf{lean}\, T^A P$ based approach can keep several copies of the same literal, whereas our LK variant only keeps new literals; a second difference is that $\mathsf{lean}\, T^A P$ searches on the negated formula instead of the original formula.

7.5 Future work

In this work we have only considered the propositional fragment. One might consider whether the work applies to quantifiers as well. It is straightforward to get a simple extension with quantifiers. One assigns $\forall$ to the negative connectives and $\exists$ to the positive connectives, and add the following rules to $\mathrm{LKF}_{\mathrm{CF}}$:

$$\frac{\vdash \xi, [\Theta], \Gamma, A}{\vdash \xi, [\Theta], \Gamma, \forall x A} \qquad \frac{\mapsto [\Theta, \exists y A], A[t/y]}{\mapsto [\Theta], \exists y A}$$

where x is not free in ξ, Θ or Γ. The proofs of soundness and completeness in Sect. 7.2 extend straightforwardly to this extension. One thing, which does not extend directly, is the linear logic compilation given in Sect. 7.3. The rule for the existential cannot be directly modelled in the translation; the problem is that the entire formula is added to the context and must be available for later refocus, which means that the translation of the existential must include itself as a subformula, which will not work. In LKF and LJF the situation is different, because the existential stays in the context, and contraction can be applied to it. A solution might be to add another subexponential, which allows both weakening and contraction, and use that one to hold the existentials. That subexponential would have to be larger than x and equal (with respect to $\preceq$) to y. Another idea could be to add fixpoints to the logic.

If the quantifiers range over a finite domain, S, then they are basically just big disjunctions. In that case, the existential can be given a rule similar to the positive disjunction:

$$\frac{\mapsto [\Theta, \exists y \in S \setminus \{t\}.\ A], A[t/y]}{\mapsto [\Theta], \exists y \in S.\ A}$$

Another topic of interest is how well suited the proof system is for reasoning; is it possible to prove, for instance, cut elimination directly in $\mathrm{LKF}_{\mathrm{CF}}$? We have tried a small example, proving that the generalized initial rule holds for the system. To prove such a theorem, we used the same generalization (using the $\uparrow$ relation) as is used to prove completeness. We speculate that it is therefore possible to prove cut elimination directly, although it might take some work.

A last line of possible future work is to look at how to extend the results to other systems, in particular an intuitionistic focused proof system like LJF. One could probably use the same idea for the focus rule, but one would have to take care with the rule for implication on the left. Possibly something similar to the system by Dyckhoff [27] could be used.

Part III

Conclusion and future work

Chapter 8

Conclusion and future work

Starting from a problem (Foundations for certified code for concurrent and distributed processes), which seemed mostly technical, we arrived at a conceptually different way of thinking about formal specifications. The main insight is that system analysis, design and modelling should not be done from an a priori cooperative viewpoint, where all actors are supposedly working together towards a common goal. Instead, one should rather take a fundamentally adversarial viewpoint, where the different actors behave in accordance which their own (to the other actors unknown) goals.

Returning to the foundations, we defined two concrete mathematical models for multiparty interactions, based on the inherently adversarial notion of games. These models allow certification, based on a paradigm which we call verification-time monitoring, allowing standard Floyd-Hoare logic to be used for certifying real-time communicating processes.

The work is mainly focused on the foundational part, and concrete real-world scenarios are not considered in detail in this work, nor are the substantial engineering challenges in making framework practically usable. Below, we summarise the results and future work.

8.1 Summary

In this work we have obtained the following results:

Adversarial composition. In Chapter 2 we argued that to model real-life situations faithfully, one has to account for adversarial compositions. In such compositions, the different actors may unknowingly or deliberately work against each other. We argued that a single goal or choreography in such a setting is inherently a compromise between different actors, and should therefore not be viewed as a specification, but rather as an implementation of a logically distributed system. The specifications should instead be expressed as legally binding contracts, stating exactly how the different actors are penalised or rewarded for different behaviour.

Adversarial modelling. In Chapter 3 we presented a mathematical framework for modelling real-time, adversarial compositions. Specifications were modelled as game-like contracts, assigning the participants a real-valued pay-off for each run. Implementations were modelled as strategies, where a strategy for a legal principal consists of a set of

tactics for the principal's agents, designed to collectively do well in all the principal's contracts.

The framework consisted of two concrete models. The first model formalised contracts and tactics as respectively real-valued predicates on and transformers of traces. We then showed how to compose the behaviour of two tactics, and proved that if each tactic conforms with a series of contracts, their composition conforms with the union of contracts, provided that their mutual interactions are described by a contract, in which the two tactics play dual roles. This allows reasoning about each tactic in isolation. To make contracts and tactics more intensional, the second model formalised them in terms of automata. We defined conformance for these automata and showed how to relate the automata-based model to the trace-based model. Concretely, we proved that if a tactic automaton conforms (as an automaton) with a set of contract automata, then the trace-based denotation of the tactic conforms (as a trace operator) with the trace-based denotation of the contracts.

Certification by verification-time monitoring. In Chapter 4 we introduced a new paradigm for certifying interacting real-time processes. We called this paradigm verification-time monitoring, based on the idea of a test harness, which supervises the tactic by feeding it input observations and monitoring its output actions, using both to advance the state of the contracts, and accumulating the pay-offs. The test harness returns an error, if at some point the tactic reaches a negative balance of pay-offs. This approach turns the question of conformance into a question of whether the test harness avoids an error state. We then defined a simple coroutine language, and showed how to implement the test harness as a sequential program in that language. That allowed us to define a Floyd-Hoare logic to certify safety of the harnessed tactic and therefore in turn conformance. Lastly, we showed a small proof-of-concept implementation of the verification-time monitoring paradigm, including a small example with one tactic and two contracts.

Towards resource-based interaction. In Chapter 5 we presented preliminary work on extending the framework to explicitly account for linear resources. Such resources have interesting applications in contracts about the transfer of physical goods. Furthermore, they can be used to model dynamically changing communication topologies. Concretely, we showed an extension to the trace-based model with support for attaching resources to communication events.

A proof hosting framework based on focusing. In Chapter 6 we showed how to use the focused proof system for intuitionistic logic, LJF, as a basis for hosting several different proof systems, with equivalence on the level of open derivations. The work was sparked by recent similar work, the basis of which was a focused proof system for classical linear logic. Our work shows that neither classical nor linear logic is needed to get full equivalence of open derivations – therefore focusing seems to be the main component in such an encoding, highlighting focusing as a strong proof-theoretic tool. Being an intuitionistic system, LJF also seemed to have nicer encodings of the intuitionistic systems, e.g. the only difference in the encoding of the intuitionistic sequent calculus, LJ, and the system of natural deduction, NJ, is a change in polarity.

Contraction in the presence of focusing. In Chapter 7 we showed how to remove contraction from a focused proof system for propositional classical logic. To our knowledge, all existing focused systems, for both classical and intuitionistic logic, use contraction exactly for the formula chosen for focus. With contraction, proofs risk redundancy, because the same formula could be chosen again and again. Removing contraction is not trivial, however, because contraction is needed to retain completeness. We showed that the main problem can be isolated to the treatment of the positive disjunction, because the proof of some formulas need to be able to choose the second disjunct. Our system solves this problem by adding only the non-taken disjunct to the context. We proved that this modification is both sound and complete with respect to the original system with ordinary contraction.

8.2 Future work

Each technical chapter already pointed out directions for future work. Here, we briefly summarise the most important directions:

Concrete scenarios. We have presented concrete models for performing verification and certification of concurrent and distributed systems. The practical applicability, however, needs to be thoroughly investigated by modelling and certifying concrete scenarios, preferably taken from the three application domains: programming-by-contract, communication protocols and organisational workflows.

Adversarial models. In our current models the set of negotiated contracts are static, i.e., contracts cannot be created or removed at runtime. The most interesting extension to the concrete models is accounting for dynamic creation of contracts. A promising idea for dynamic contracts is the idea of having meta-contracts that govern the creation and execution of certain object-contracts.

Verification-time monitoring. Our certification paradigm (verification-time monitoring) allows certification, using standard techniques based on Floyd-Hoare logic, to be used for communicating real-time processes. As future work it could be very interesting to investigate how well this paradigm works in practice. A concern is, how easily the full certification proofs can be constructed from minimal developer supplied annotations.

Focusing for hosting of proof systems. We showed how to use a focused proof system for intuitionistic logic to host several different proof systems. Future work should be concerned with how easy it is to work with the proofs in the framework, how easy can we e.g. prove cut elimination for the hosted systems in this framework.

Contraction elimination for focused proof systems. We showed how to remove contraction from the focused system for propositional classical logic, LKF. The most important direction for future work is whether the results can be used to remove contraction from a focused system for intuitionistic logic, e.g. LJF, and how to extend the result to the full fragment (with quantifiers).

Appendix A

Source code

A.1 Tactic, contracts and supervisor for the +2 case

```
----- GENERAL -----
-- Actions
type act = Emp | V : int

-- Input trace: lo1,li2
type train = NoIn | Actin : real, int, train

-- Output trace: li1, lo2
type traout = NoOut | Actout : real,int

-- Tactic state
type tState = S1 | S2 | S3

-- Contract states
type cState1 = C1start | C1run : int, real
type cState2 = C2start | C2run : int, real

-- Contract Running state
type run1 = Done1 : real | Run1 : cState1
type run2 = Done2 : real | Run2 : cState2

-- Supervisor states
type sTac1 = STac1 : real,real,real,real,run1,real,run2,
                     train,train,traout,traout,
                     act,act,act,act
type sTac2 = STac2 : real,real,real,real,run1,real,run2,
                     train,train,traout,traout,
                     act,act,act,act,act,act


type sCon1 = SCon1 : real,real,real,tState,real, real, run2,
                     train,train,traout,traout,
                     act,act,act,act

type sCon2 = SCon2 : real,real,real,tState,real, run1, real,
                     train,train,traout,traout,
                     act,act,act,act

----- TACTIC -----

-- Tactic io
```

```
tacI(ss : sTac2, ts : tState, lo1 : act, li2 : act) =
case ts of
  S1 -> if li2 = Emp then tacRetI(ss,S1,Emp,Emp)
                     else tacRetI(ss,S2,li2,Emp)
| S2 -> if lo1 = Emp then tacRetI(ss,S2,Emp,Emp)
                     else tacRetI(ss,S3,lo1,Emp)
| S3 -> if lo1 = Emp then tacRetI(ss,S3,Emp,Emp)
                     else tacRetI(ss,S1,Emp,lo1)
end

-- Tactic time
tacT(ss : sTac1, ts : tState) = tacRetT(ss,ts,Emp,Emp)

----- +1 CONTRACT -----

-- Contract io
con1I(ss : sCon1, cs1 : cState1, li1 : act, lo1 : act) =
case cs1 of
  C1start ->
    if lo1 /= Emp then con1RetI(ss,Done1(1.0))
    else case li1 of
           Emp -> con1RetI(ss,Run1(cs1))
         | V(n) -> con1RetI(ss,Run1(C1run(n,3.0)))
         end
| C1run (n,t) ->
  if li1 /= Emp then con1RetI(ss,Done1(0.0 - 1.0))
  else case lo1 of
         Emp -> con1RetI(ss,Run1(cs1))
       | V(n2) -> if n + 1 = n2 then con1RetI(ss,Run1(C1start))
                  else con1RetI(ss,Done1(1.0))
       end
end

-- Contract time
con1T(ss : sCon1,cs1 : cState1) =
case cs1 of
  C1start -> con1RetT(ss,Run1(cs1))
| C1run (n,t) ->
    if t <= 0.0
    then con1RetT(ss,Done1(1.0))
    else con1RetT(ss,Run1(C1run(n,t - 1.0)))
end

----- +2 CONTRACT -----

-- Contract io
con2I(ss : sCon2, cs2 : cState2, li2 : act, lo2 : act) =
case cs2 of
  C2start ->
    if lo2 /= Emp then con2RetI(ss,Done2(0.0 - 1.0))
    else case li2 of
           Emp -> con2RetI(ss,Run2(cs2))
         | V(n) -> con2RetI(ss,Run2(C2run(n,20.0)))
         end
| C2run (n,t) ->
  if li2 /= Emp then con2RetI(ss,Done2(1.0))
  else case lo2 of
         Emp -> con2RetI(ss,Run2(cs2))
       | V(n2) -> if n + 2 = n2 then con2RetI(ss,Run2(C2start))
                  else con2RetI(ss,Done2(0.0 - 1.0))
```

```
      end
end

-- Contract time
con2T(ss : sCon2,cs2 : cState2) =
case cs2 of
  C2start -> con2RetT(ss,Run2(cs2))
| C2run (n,t) ->
    if t <= 0.0
    then con2RetT(ss,Done2(0.0 - 1.0))
    else con2RetT(ss,Run2(C2run(n,t - 1.0)))
end

----- SUPERVISOR -----

start(tstart : real, tend : real, alo1 : train, ali2 : train) =
  loop(tstart,tend,
       tstart,S1,
       tstart,Run1(C1start),
       tstart,Run2(C2start),alo1,ali2,NoOut,NoOut)


loop(tnow : real, tend : real,
     ttac : real, ts : tState,
     tcon1 : real, rs1 : run1,
     tcon2 : real, rs2 : run2,
     alo1 : train, ali2 : train,
     ali1 : traout, alo2 : traout) =
  if tnow <= tend
  then splitLo1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                alo1,ali2,ali1,alo2)
  else done

-- Splitting -- {{{

splitLo1(tnow : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case alo1 of
    NoIn -> splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2,Emp)
  | Actin (tlo1,lo1,alo1_) ->
    if tlo1 <= tnow then splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1_,ali2,ali1,alo2,V(lo1))
                    else splitLi2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1,ali2,ali1,alo2,Emp)
  end

splitLi2(tnow : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout,
         lo1now : act) =
  case ali2 of
    NoIn -> splitLi1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
```

```
                    alo1,ali2,ali1,alo2,lo1now,Emp)
  | Actin (tli2,li2,ali2_) ->
    if tli2 <= tnow then splitLi1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1,ali2_,ali1,alo2,lo1now,V(li2))
                    else splitLi1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1,ali2,ali1,alo2,lo1now,Emp)
  end

splitLi1(tnow : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout,
         lo1now : act,li2now : act) =
  case ali1 of
    NoOut -> splitLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                      alo1,ali2,ali1,alo2,lo1now,li2now,Emp)
  | Actout (tli1,li1) ->
    if tli1 <= tnow then splitLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1,ali2,NoOut,alo2,
                                  lo1now,li2now,V(li1))
                    else splitLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                  alo1,ali2,ali1,alo2,
                                  lo1now,li2now,Emp)
  end

splitLo2(tnow : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout,
         lo1now : act,li2now : act,li1now : act) =
  case alo2 of
    NoOut -> invTac(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                    alo1,ali2,ali1,alo2,lo1now,li2now,li1now,Emp)
  | Actout (tlo2,lo2) ->
    if tlo2 <= tnow then invTac(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                alo1,ali2,ali1,NoOut,
                                lo1now,li2now,li1now,V(lo2))
                    else invTac(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                                alo1,ali2,ali1,alo2,
                                lo1now,li2now,li1now,Emp)
  end

-- Splitting -- }}}

-- Invoke and return from tactic {{{

invTac(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  if ttac <= tnow
  then tacT(STac1(tnow,tend,ttac + 1.0,tcon1,rs1,tcon2,rs2, -- Timeout
                  alo1,ali2,ali1,alo2,lo1now,li2now,li1now,lo2now),ts)
```

```
  else tacRetT(STac1(tnow,tend,ttac,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2,lo1now,li2now,li1now,lo2now),
               ts,Emp,Emp)

tacRetT(ss : sTac1, ts : tState, li1 : act, lo2 : act) =
  case ss of
    STac1(tnow,tend,ttac,tcon1,rs1,tcon2,rs2,
          alo1,ali2,ali1,alo2,lo1now,li2now,li1now,lo2now) ->
      if lo1now = Emp & li2now = Emp
      then updLi1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                  alo1,ali2,ali1,alo2,
                  lo1now,li2now,li1now,lo2now,li1,lo2)
      else tacI(STac2(tnow,tend,ttac,tcon1,rs1,tcon2,rs2,
                      alo1,ali2,ali1,alo2,
                      lo1now,li2now,li1now,lo2now,
                      li1,lo2),ts,lo1now,li2now)
  end


tacRetI(ss : sTac2, ts : tState, li1_ : act, lo2_ : act) =
  case ss of
    STac2(tnow,tend,ttac,tcon1,rs1,tcon2,rs2,
          alo1,ali2,ali1,alo2,
          lo1now,li2now,li1now,lo2now,li1,lo2) ->
    let li1__ = if li1 = Emp then li1_ else li1
    in let lo2__ = if lo2 = Emp then lo2_ else lo2
       in updLi1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                 alo1,ali2,ali1,alo2,
                 lo1now,li2now,li1now,lo2now,li1__,lo2__)
       end
    end
  end

-- Invoke and return from tactic }}}

-- Update output -- {{{

updLi1(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act,
       li1 : act,lo2 : act) =
  if ali1 = NoOut
  then case li1 of
         Emp -> updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                       NoOut,
                       alo2,lo1now,li2now,li1now,lo2now,lo2)
       | V(n) -> updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                        Actout(tnow + 1.0,n), -- latency
                        alo2,lo1now,li2now,li1now,lo2now,lo2)
       end
  else updLo2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
              ali1,
              alo2,lo1now,li2now,li1now,lo2now,lo2)

updLo2(tnow : real, tend : real,
       ttac : real, ts : tState,
```

```
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act,
       lo2 : act) =
  if alo2 = NoOut
  then case lo2 of
         Emp -> invCon1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                        ali1,NoOut,
                        lo1now,li2now,li1now,lo2now)
       | V(n) -> invCon1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
                         ali1,Actout(tnow + 1.0,n), -- latency
                         lo1now,li2now,li1now,lo2now)
       end
  else invCon1(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,
               ali1,alo2,lo1now,li2now,li1now,lo2now)

-- Update output -- }}}

-- Invoke and return from con1 {{{

invCon1(tnow : real, tend : real,
        ttac : real, ts : tState,
        tcon1 : real, rs1 : run1,
        tcon2 : real, rs2 : run2,
        alo1 : train, ali2 : train,
        ali1 : traout, alo2 : traout,
        lo1now : act,li2now : act,li1now : act,lo2now : act) =
  case rs1 of
    Done1(k) -> invCon2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                        alo1,ali2,ali1,alo2,
                        lo1now,li2now,li1now,lo2now)
  | Run1(cs) ->
    if tcon1 <= tnow
    then con1T(SCon1(tnow,tend,ttac,ts,tnow + 1.0, -- Timeout
                     tcon2,rs2,alo1,ali2,ali1,alo2,
                     lo1now,li2now,li1now,lo2now),cs)
    else con1RetT(SCon1(tnow,tend,ttac,ts,tcon1,
                        tcon2,rs2,alo1,ali2,ali1,alo2,
                        lo1now,li2now,li1now,lo2now),rs1)
  end

con1RetT(ss : sCon1, rs1 : run1) =
  case ss of
    SCon1(tnow,tend,ttac,ts,tcon1,
          tcon2,rs2,alo1,ali2,ali1,alo2,
          lo1now,li2now,li1now,lo2now) ->
      case rs1 of
        Done1(k) -> invCon2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                            alo1,ali2,ali1,alo2,
                            lo1now,li2now,li1now,lo2now)
      | Run1(cs) ->
        if lo1now = Emp & li1now = Emp
        then invCon2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2,
                     lo1now,li2now,li1now,lo2now)
        else con1I(ss, cs, li1now, lo1now)

    end
```

```
  end

con1RetI(ss : sCon1, rs1 : run1) =
  case ss of
    SCon1(tnow,tend,ttac,ts,tcon1,
          tcon2,rs2,alo1,ali2,ali1,alo2,
          lo1now,li2now,li1now,lo2now) ->
      invCon2(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
              alo1,ali2,ali1,alo2,
              lo1now,li2now,li1now,lo2now)
  end

-- Invoke and return from con1 }}}

-- Invoke and return from con2 {{{

invCon2(tnow : real, tend : real,
        ttac : real, ts : tState,
        tcon1 : real, rs1 : run1,
        tcon2 : real, rs2 : run2,
        alo1 : train, ali2 : train,
        ali1 : traout, alo2 : traout,
        lo1now : act,li2now : act,li1now : act,lo2now : act) =
  case rs2 of
    Done2(k) -> payoff(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                       alo1,ali2,ali1,alo2,
                       lo1now,li2now,li1now,lo2now)
  | Run2(cs) ->
    if tcon2 <= tnow
    then con2T(SCon2(tnow,tend,ttac,ts,tcon1,rs1,
                     tnow + 1.0,alo1,ali2,ali1,alo2, -- Timeout
                     lo1now,li2now,li1now,lo2now),cs)
    else con2RetT(SCon2(tnow,tend,ttac,ts,tcon1,rs1,
                        tcon2,alo1,ali2,ali1,alo2,
                        lo1now,li2now,li1now,lo2now),rs2)
  end

con2RetT(ss : sCon2, rs2 : run2) =
  case ss of
    SCon2(tnow,tend,ttac,ts,tcon1,rs1,
          tcon2,alo1,ali2,ali1,alo2,
          lo1now,li2now,li1now,lo2now) ->
      case rs2 of
        Done2(k) -> payoff(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                           alo1,ali2,ali1,alo2,
                           lo1now,li2now,li1now,lo2now)
      | Run2(cs) ->
        if lo2now = Emp & li2now = Emp
        then payoff(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                    alo1,ali2,ali1,alo2,
                    lo1now,li2now,li1now,lo2now)
        else con2I(ss, cs, li2now, lo2now)

    end
  end

con2RetI(ss : sCon2, rs2 : run2) =
  case ss of
    SCon2(tnow,tend,ttac,ts,tcon1,rs1,
          tcon2,alo1,ali2,ali1,alo2,
```

```
          lo1now,li2now,li1now,lo2now) ->
      payoff(tnow,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
             alo1,ali2,ali1,alo2,
             lo1now,li2now,li1now,lo2now)
  end

-- Invoke and return from con2 }}}

payoff(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  case rs1 of
    Done1(k1) -> case rs2 of
                   Done2(k2) -> if k1 + k2 < 0.0 then fail
                                else done
                 | Run2(cs2) -> if k1 < 0.0 then fail
                                else finish(tnow,tend,ttac,ts,
                                            tcon1,rs1,tcon2,rs2,
                                            alo1,ali2,ali1,alo2,
                                            lo1now,li2now,li1now,lo2now)
                 end
  | Run1(cs1) -> case rs2 of
                   Done2(k2) -> if k2 < 0.0 then fail
                                else finish(tnow,tend,ttac,ts,
                                            tcon1,rs1,tcon2,rs2,
                                            alo1,ali2,ali1,alo2,
                                            lo1now,li2now,li1now,lo2now)
                 | Run2(cs2) -> finish(tnow,tend,ttac,ts,
                                       tcon1,rs1,tcon2,rs2,
                                       alo1,ali2,ali1,alo2,
                                       lo1now,li2now,li1now,lo2now)
                 end
  end


finish(tnow : real, tend : real,
       ttac : real, ts : tState,
       tcon1 : real, rs1 : run1,
       tcon2 : real, rs2 : run2,
       alo1 : train, ali2 : train,
       ali1 : traout, alo2 : traout,
       lo1now : act,li2now : act,li1now : act,lo2now : act) =
  let tmin = if ttac < tcon1
             then if ttac < tcon2 then ttac else tcon2
             else if tcon1 < tcon2 then tcon1 else tcon2
  in firstLo1(tnow,tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
              alo1,ali2,ali1,alo2)
  end

-- Calculate next timepoint {{{

firstLo1(tnow : real, tmin : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
```

```
        ali1 : traout, alo2 : traout) =
  case alo1 of
    NoIn -> firstLi2(tnow,tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2)
  | Actin (tlo1,lo1,alo1_) ->
      firstLi2(tnow,if tlo1 < tmin then tlo1 else tmin,
               tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,ali1,alo2)
  end

firstLi2(tnow : real, tmin : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case ali2 of
    NoIn -> firstLi1(tnow,tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                     alo1,ali2,ali1,alo2)
  | Actin (tli2,li2,ali2_) ->
      firstLi1(tnow,if tli2 < tmin then tli2 else tmin,
               tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,ali1,alo2)
  end

firstLi1(tnow : real, tmin : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case ali1 of
    NoOut -> firstLo2(tnow,tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                      alo1,ali2,ali1,alo2)
  | Actout (tli1,li1) ->
      firstLo2(tnow,if tli1 < tmin then tli1 else tmin,
               tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,ali1,alo2)
  end

firstLo2(tnow : real, tmin : real, tend : real,
         ttac : real, ts : tState,
         tcon1 : real, rs1 : run1,
         tcon2 : real, rs2 : run2,
         alo1 : train, ali2 : train,
         ali1 : traout, alo2 : traout) =
  case alo2 of
    NoOut -> if tnow < tmin
             then loop(tmin,tend,ttac,ts,tcon1,rs1,tcon2,rs2,
                       alo1,ali2,ali1,alo2)
             else done
  | Actout (tlo2,lo2) ->
    if tnow < tmin
    then loop(if tlo2 < tmin then tlo2 else tmin,
              tend,ttac,ts,tcon1,rs1,tcon2,rs2,alo1,ali2,ali1,alo2)
    else done
  end

-- Calculate next timepoint }}}
```

Bibliography

[1] Samson Abramsky, Dan Ghica, Andrzej Murawski, and C. Ong. Applying game semantics to compositional software modeling and verification. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, pages 421–435. Springer Berlin / Heidelberg, 2004.

[2] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 291–301, 1992.

[3] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1998.

[4] Alexandre Alves, Assaf Arkin, Sid Askary abd Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Oasis web services business process execution language (WSBPEL) v2.0, 2007.

[5] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[6] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, 2001.

[7] David Baelde, Dale Miller, and Zachary Snow. Focused inductive theorem proving. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 278–292, 2010.

[8] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org, 2010.

[9] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15:339–358, 1995.

[10] Azer Bestavros. The input output timed automaton: A model for real-time parallel computation. In *International workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, 1990.

[11] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Towards a formal framework for choreography. In *14th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE '05)*, pages 107–112, 2005.

[12] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *16th European conference on Programming (ESOP 07)*, pages 2–17, 2007.

[13] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. WCD-Working Note, 2006. Available at http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf.

[14] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *First International Conference on Foundations of Software Science and Computation Structure (FoSSaCS '98)*, pages 140–155. Springer-Verlag, 1998.

[15] Kaustuv Chaudhuri and Frank Pfenning. A focusing inverse method theorem prover for first-order linear logic. In *20th Conference on Automated Deduction (CADE)*, pages 69–83, 2005.

[16] Rance Cleaveland and Scott A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28:607–625, 1996.

[17] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI' 00)*, pages 95–107. ACM, 2000.

[18] Karl Crary and Susmit Sarkar. Foundational certified code in the twelf metalogical framework. *ACM Transactions on Computational Logic (TOCL)*, 9:16:1–16:26, 2008.

[19] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Fifth international conference on Functional programming (ICFP)*, pages 233–243, 2000.

[20] Marcello D'Agostino and Marco Mondadori. The taming of the cut. Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.

[21] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 159–171. Springer, 1993.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[23] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *14th International Conference for Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, pages 337–340, 2008.

[25] Paul E. Dunne, Sarit Kraus, Efrat Manisterski, and Michael Wooldridge. Solving coalitional resource games. *Artificial Intelligence*, 174:20–50, 2010.

[26] R. Dyckhoff and S. Lengrand. LJQ: a strongly focused calculus for intuitionistic logic. In A. Beckmann et al, editor, *Computability in Europe 2006*, volume 3988 of *LNCS*, pages 173–185. Springer, 2006.

[27] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, September 1992.

[28] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *The 7th ACM SIGPLAN international conference on Functional programming (ICFP' 02)*, pages 48–59. ACM, 2002.

[29] Melvin Fitting. leanTAP revisited. *Journal of Logic and Computation*, 8(1):33–47, 1998.

[30] R. W. Floyd. Assigning meaning to programs. In *Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[31] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *23rd symposium on Principles of programming languages (POPL '96)*, pages 372–385, 1996.

[32] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR '96)*, pages 406–421. Springer-Verlag, 1996.

[33] Dov M. Gabbay and Uwe Reyle. N-prolog: An extension of prolog with hypothetical implications I. *Journal of Logical Programming*, 1(4):319–355, 1984.

[34] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[35] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *2nd Symposium on Logic in Computer Science (LICS'87)*, pages 194–204, June 1987.

[36] Anders Starcke Henriksen. Comparing metalogical code-certification approaches. Master's thesis, Department of Computer Science, University of Copenhagen, 2008.

[37] Anders Starcke Henriksen. Using LJF as a framework for proof systems. Technical Report, University of Copenhagen, 2009. Available at http://hal.inria.fr/inria-00442159/en/.

[38] Anders Starcke Henriksen, Tom Hvitved, and Andrzej Filinski. A game-theoretic model for distributed programming by contract. In *Workshop on Games, Business Processes, and Models of Interactions*, September 2009.

[39] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *3rd international joint conference on Artificial intelligence (IJCAI)*, pages 235–245, 1973.

[40] Thomas Hildebrandt. Trustworthy pervasive healthcare services (TrustCare). Webpage http://www.trustcare.eu, 2008.

[41] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Distributed dynamic condition response structures. In *International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2010)*, 2010.

[42] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Declarative modelling and safe distribution of healthcare workflows. Accepted for International Symposium on Foundations of Health Information Engineering and Systems (FHIES'11), 2011.

[43] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Designing a cross-organizational case management system using dynamic condition response graphs. Accepted for IEEE International EDOC Conference (EDOC' 11), 2011.

[44] André Hirschowitz, Michel Hirschowitz, and Tom Hirschowitz. Contraction-free proofs and finitary games for linear logic. In *25th Conference on Mathematical Foundations of Programming Semantics (MFPS)*, volume 249 of *LNCS*, pages 287–305, 2009.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[46] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., 1985.

[47] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, pages 122–138, 1998.

[48] Jozef Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6:801–825, 1994.

[49] Tom Hvitved and Anders Starcke Henriksen. Foundations for programming by contract in a concurrent and distributed environment. Internal document, 2009.

[50] Dilsun Kirli Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *24th International Real-Time Systems Symposium*, 2003.

[51] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78:293–303, 2009.

[52] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[53] Chuck Liang and Dale Miller. A unified sequent calculus for focused proofs. In *24th Symposium on Logic in Computer Science (LICS)*, pages 355–364, 2009.

[54] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[55] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *2nd International Workshop on Process-oriented information systems in healthcare (ProHealth 08)*, 2008.

[56] Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized focused inverse method for intuitionistic propositional logic. In *15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 174–181, 2008.

[57] Elliot Mendelson. *Introducing Game Theory and its Applications*. Chapman & Hall/CRC, 2004.

[58] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[59] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *9th Symposium on Logic in Computer Science (LICS)*, pages 272–281, Paris, July 1994. IEEE Computer Society Press.

[60] Dale Miller and Elaine Pimentel. Using linear logic to reason about sequent systems. In Uwe Egly and Christian G. Fermüller, editors, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *LNCS*, pages 2–23. Springer, 2002.

[61] Dale Miller and Elaine Pimentel. Linear logic as a framework for specifying sequent calculus. In *Logic Colloquium '99: Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic*, Lecture Notes in Logic, pages 111–135. A K Peters Ltd, 2004.

[62] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1980.

[63] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[64] Robin Milner. *Communicating and mobile systems: the π-Calculus*. Cambridge University Press, 1999.

[65] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Compututation*, 100(1):1–40, 1992.

[66] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *5th International Conference on Logic Programming (ICLP)*, pages 810–827, August 1988.

[67] George C. Necula. Proof-carrying code. In *24th Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, 1997.

[68] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI' 98)*, pages 333–344. ACM, 1998.

[69] George Ciprian Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, 1998.

[70] Sara Negri. Contraction-free sequent calculi for geometric theories with an application to barr's theorem. *Archive for Mathematical Logic*, 42:389–401, 2003.

[71] Vivek Nigam. *Exploiting non-canonicity in the sequent calculus*. PhD thesis, Ecole Polytechnique, September 2009.

[72] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *11th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 129–140, 2009.

[73] Vivek Nigam and Dale Miller. A framework for proof systems. *Journal of Automated Reasoning*, 45:157–188, 2010.

[74] Joëlle Noailly, Jeroen C.J.M. van den Bergh, and Cees A. Withagen. Local and global interactions in an evolutionary resource game. *Computational Economics*, 33(2):155–173, 2009.

[75] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[76] Michel Parigot. Free deduction: An analysis of "computations" in classical logic. In *Proceedings of the First Russian Conference on Logic Programming*, pages 361–380, London, UK, 1992. Springer-Verlag.

[77] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *International Conference on Typed Lambda Calculi and Applications (TLCA' 93)*, pages 328–345. Springer-Verlag, 1993.

[78] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[79] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[80] Frank Pfenning. Automated theorem proving. Lecture notes, March 2004.

[81] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction (CADE '99)*, pages 679–679, 1999.

[82] Elaine Pimentel and Dale Miller. On the specification of sequent systems. In *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, number 3835 in LNAI, pages 352–366, 2005.

[83] Elaine Gouvêa Pimentel. *Lógica linear e a especificação de sistemas computacionais*. PhD thesis, Universidade Federal de Minas Gerais, Belo Horizonte, M.G., Brasil, December 2001. Written in English.

[84] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS' 77)*, pages 46–57, 1977.

[85] Princeton University. zChaff. Available at http://www.princeton.edu/~chaff/zchaff.html.

[86] Microsoft Research. Z3 theorem prover. http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[87] Steve Ross-Talbot and Tony Fletcher. Web services choreography description language: Primer. W3C working draft, 2006. Available at http://www.w3.org/TR/ws-cdl-10-primer/.

[88] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2:277–288, 1984.

[89] Peter Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, 1984.

[90] Dana Scott. Identity and existence in intuitionistic logic. In *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer Berlin / Heidelberg, 1979.

[91] Wilfried Sieg and John Byrnes. Normal natural deduction proofs (in classical logic). *Studia Logica*, 60(1):67–106, 1998.

[92] Raymond M. Smullyan. Analytic cut. *Journal of Symbolic Logic*, 33(4):560–564, 1968.

[93] Geoff Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[94] Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.

[95] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and implementation of the YAWL system. In *16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, pages 142–159. Springer-Verlag, 2004.

[96] W.M.P. van der Aalst and M. Pesic. Decserflow: Towards a truly declarative service flow language. In *Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin / Heidelberg, 2006.

[97] W.M.P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23:99–113, 2009.

[98] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–344, 1986.

[99] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 2004.

[100] Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001.

[101] W3C. Web Services Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2005.

[102] Glynn Winskel. Event structures. In *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987.

[103] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *36th Symposium on Principles of programming languages*, pages 41–52, 2009.