CARNEGIE MELLON UNIVERSITY

Session-Typed Recursive Processes and Circular Proofs

Farzaneh Derakhshan

Thesis submitted in partial fulfillment
for the degree of Doctor of Philosophy

Thesis committee:
David Baelde (ENS Cachan)
Stephanie Balzer
Adam Bjorndahl
Frank Pfenning (Chair)
Wilfried Sieg

May 2021

Acknowledgements

First and foremost, I would like to extend my sincerest gratitude and respect to my advisor, Professor Frank Pfenning, for his kind encouragement, patience, and supervision. It has been a real privilege being his student, and I have learned a great deal from him. He has been an excellent role model for me as a researcher, teacher, and person.

My sincere thanks goes to Professor Wilfried Sieg for his constant encouragement, numerous stimulating discussions, and excellent advice. It has been an honor for me to work with him. I would like to also express my gratitude to Dr. Stephanie Balzer with whom I have had the pleasure of collaborating. Her ideas and her support have provided me with great opportunities. I would like to thank the other committee members, Dr. David Baelde, and Dr. Adam Bjorndahl, for serving on the committee and for their valuable feedback. In particular, I appreciate the detailed discussions with Dr. David Baelde about infinitary proof theory.

I wish to thank Dr. Majid Alizadeh, who first introduced me to proof theory and logic during my undergraduate studies.

I gratefully acknowledge my colleagues at CMU for their helpful comments and discussions on topics related to this work: Ankush Das, Henry Deyoung, Zhibo Chen, Jonas Frey, Klaas Pruiksma, and Siva Somayyajula.

Many thanks to my friends and colleagues, Kendra Chilson, Fernando Larrain Langlois, Zesen Qian, Egbert Rijke, Elizabeth Viera-Patron, Patrick Walsh, Colin Zwanziger, and many others for contributing to a friendly and enjoyable atmosphere at the Department of Philosophy. I express my deep appreciation to my close friends Mate Szabo and Jeff Adams, who kindly supported me in my path and helped me in any way they could. Without them, I would not have made it through the first couple of years of graduate school.

There are many others I would like to mention - in no particular order - my friends who supported me to finish the thesis: Ghazaleh Parvini, Nedah Nemati, Ardeshir Raihanian, Nooshin Shafaei, Fatemeh mohammadhashem, Marzieh sharifi and many more.

Most importantly, I must express my deepest appreciation to my lovely family. My mother, Touran Nikpour, and my father, Eskandar Derakhshan, have encouraged me and supported me with endless love since I can remember, and my dear sisters have always been there for me. They have been in my heart, even though I could not visit them for more than three years. Most of my gratitude goes to my spouse, Farshad Ghanei, for his patience and love throughout these years.

Abstract

Session types describe the communication behavior of interacting processes. Binary session types, in which each channel has two endpoints, corresponds to intuitionistic linear logic by a Curry-Howard interpretation of propositions as types, proofs as programs, and cut reduction as communication. This interpretation provides a solid foundation for reasoning about the behavior of session-typed processes. For example, termination of a process can be inferred from the cut elimination property of its underlying proof. However, as soon as we add recursive session types we abandon this correspondence and lose our solid ground. From the programming perspective it means that we can no longer exploit the cut elimination property to guarantee termination.

This document establishes a logical foundation for recursive binary session-typed processes using infinitary proof systems for linear logic. We show that if we refine recursive types as least and greatest fixed points and impose a guard condition on recursive processes, we can still guarantee meaningful communication, ensuring that a program always terminates either in an empty configuration or one attempting to communicate along external channels. To develop this logical foundation, we appeal to two well-known paradigms that relate programs to logical systems: *types-as-propositions* and *processes-as-formulas*.

Contents

Chapter 1

Introduction

Proof theory as we know it today is built upon Hilbert's efforts to prove the consistency of mathematics. Hilbert's radical foundational goal to prove consistency was abandoned after Gödel presented his incompleteness theorems; however, the logical structure further established based on Hilbert's efforts became a foundation for computer science and particularly programming languages. Proof theory has heavily influenced the development of programming languages and provided new insights into designing new computational systems. This thesis exploits this foundation to study termination of concurrent recursive programs.

1.1 Logic and programming languages

Two main paradigms have endorsed the foundational role of logical proofs in formalizing programming languages: *types-as-propositions* and *programs-as-formulas*.

- The **types-as-propositions** paradigm, also known as Curry-Howard correspondence, has its roots in Curry's [20] discovery that axiomatic schemas correspond to types of combinators and Howard's [56] interpretation of simply-typed λ-calculus in intuitionistic natural deduction. Under this interpretation, types correspond to propositions, programs to proofs, and computations to proof reductions.

  The (simply-typed) λ-calculus is the original (statically-typed) functional programming language introduced by Church [19]. Like all other functional programming languages, the (simply-typed) λ-calculus emphasis is on interpreting the computation as the *evaluation of a function*. Evaluating functions in nature gives rise to a model with a uni-directional flow from the inputs to the output. This uni-directional flow results from the asymmetric behavior of functions toward their inputs and output. The asymmetry manifests itself in the corresponding natural deduction system too, where all logically genuine rules apply to the succedents.

More recently, a correspondence between linear logic [41] and binary session types (either in its intuitionistic [15, 16] or classical [98] formulation) has been recognized. Session types provide a typed foundation for communication centered programming (CCP). As opposed to the uni-directional nature of functions, the structural units of CCP are bi-directional communicating sessions (or processes) connected by channels. The communications along channels are governed by a protocol associated with them. These protocols are expressed as session types. A collection of interrelated processes is called a configuration. *Binary session types*, is a particular form of session types in which each channel has two endpoints. A binary channel connects the provider of a resource to its client. When such a channel connects two processes within a configuration of processes it is considered *internal* and private; other *external* channels provide an interface to a configuration and communication along them may be observed.

Caires and Pfenning [15] interpret a binary session type system as a Gentzen-style sequent calculus for intuitionistic linear logic. Under this interpretation, propositions are associated with session types, proofs in the sequent calculus with concurrent processes, and cut reduction with communication. They show that the left and right rules for each connective in the linear logic capture a principal action in session types, e.g., sending or receiving a label/channel to or from the left and right.

The Curry-Howard correspondence provides a solid ground for studying and reasoning about programming languages. For example, proving deadlock freedom for interrelated binary session-typed processes follows from cut-reduction for each connective in the underlying sequent calculus. Moreover, a terminating cut-elimination algorithm for the underlying proofs ensures termination of concurrent processes

- Logic programming, in general, expresses programs as formulas and computation as proof construction. As a particular case, Miller [68] introduced the **processes-as-formulas** paradigm to express processes in the π-calculus as formulas in linear logic. His approach is based on the observation that linear logic is a suitable interface between computer science and logic; it can model the dynamics of processes by describing the state of a process and its resources as a linear formula. In this interpretation, a computation of a process corresponds to the construction of its corresponding formula proof, and logical implication gives rise to a process preorder. This approach has been used to prove properties about processes, e.g., proofs of deadlock-freedom for (recursive) session types [54], and bisimilarity for π-calculus processes [96].

1.2 Recursion and termination

In the context of programming, recursive session types and recursive processes have also been considered [60, 97]. Together they can capture unbounded interactions between processes. They seem to fit smoothly, just as recursive types fit well into functional programming languages. However, this comes at a price: we abandon the proposition-as-types correspondence.

From the programming perspective it means that we can no longer exploit the cut elimination property to guarantee termination of programs.

Even in the presence of the recursive session types and recursively defined processes, the process typing rules guarantee deadlock-freedom, also known as the progress property [15, 97]. The *progress property* for a configuration of processes ensures that during its computation, it either (i) takes an internal communication step, or (ii) is empty, or (iii) communicates along one of its external channels. However, we may spawn a vacuous process that will get stuck in an infinite number of internal communication steps without ever communicating along an external channel. The programmer will generally be interested in a stronger form of progress that restricts this non-communicating behavior. We introduce strong progress that requires any of conditions (ii) or (iii) to hold after a finite number of computation steps. This strong version of the progress property ensures that a configuration terminates either in an empty configuration or one attempting to communicate along an external channel.

This is similar to the functional programming setting with general recursive types. Without any restrictions on the programs or types, a well-typed program may diverge and never return a value.

1.3 Our work

The main thesis can be summarized as follows:

> *Even in the presence of recursion, we can retain the propositions-as-types correspondence between linear logic and session-typed concurrent programs if we (a) refine general recursive session types into least and greatest fixed points, and (b) impose conditions under which recursively defined processes correspond to valid circular proofs. With this approach we can retain the correspondence between cut elimination, and meaningful communication with type preservation and strong progress.*

General (nonlinear) type theory has followed a similar path, isolating inductive and coinductive types with a variety of conditions to ensure validity of proofs. Mendler [65, 66] first formalized inductive and coinductive types in simply-typed lambda calculus. He imposed a simple positivity condition on types and proved normalization of the calculus. The recursion schema introduced by Mendler resembles general recursion, but its power is limited to primitive recursion [65] through typing. Moreover, the use of Mendler's recursive types is restricted in the sense that it can reason about recursive types using the recursion principle but cannot unfold their definitions directly. Our system resembles Mendler-style languages as we also introduce inductive and coinductive types to our language. However, in contrast to Mendler's system, our language has the full power of general recursion schema, and we unfold the definition of the inductive and coinductive types directly both on the left and right of the judgment. Moreover, in the setting of linear logic, we find many more symmetries than typically present in traditional type theories that appear to be naturally biased towards least fixed points and inductive reasoning.

1.3.1 Design choice: rules for fixed points

There are two main approaches for incorporating fixed points in a linear logic proof system. One approach studied by Baelde and Miller [8] yields a finitary system that enjoys the cut elimination property but not the sub-formula property. The other approach results in an infinitary system that maintains the sub-formula property but not necessarily the cut elimination property [7, 33, 36]. The infinitary systems for linear logic are always equipped with additional conditions on derivations to ensure the cut elimination property. We can represent some infinitary derivations, called *circular derivations*, as finite trees with loops (or circular edges).

We choose to build the typing rules for least and greatest fixed points of session types upon an infinitary proof system since it requires a minimal change to the system of session types. This is in contrast to the design choice made by Lindley and Morris [60] to build their system for recursive session-typed processes based on the finitary system introduced by Baelde and Miller [8]. In our design, the recursive process calls are supported as they correspond to circular edges in the derivation. Moreover, the left and right unfolding rules for general recursive types remain intact when refined into least and greatest fixed points; the left and right rules for both fixed points unfold the definition of a recursive type. The rules of least and greatest fixed points are only differentiated by their different semantics. As a drawback, we need to provide an extra condition on the processes to ensure strong progress.

A similar design choice exists when considering inductive and coinductive types in the setting of functional programming. One may choose to work with an infinitary system that requires additional guard conditions but is closer to the recursive programming schema, or a finitary system with a simpler condition but not ergonomically suitable for implementing recursion.

1.3.2 Design choice: subsingleton fragment

Subsingleton logic is a fragment of intuitionistic linear logic [18, 41] in which the antecedent and succedent of each judgment consist of at most one proposition. This reduces consideration to the additive connectives and multiplicative units, because the left or right rules of other connectives would violate this restriction. The expressive power of pure subsingleton logic is rather limited, among other things due to the absence of the exponential $!A$. However, we can recover significant expressive power by adding least and greatest fixed points, which can be done without violating the subsingleton restriction.

In this thesis we focus mainly on binary session types defined over the subsingleton fragment of linear logic. In this fragment each process uses the service of at most one process on its left and provides its own resource to the right. Moreover, we allow least and greatest fixed points and recursively defined processes that correspond to circular derivations in the underlying logic. The computational power of the subsingleton fragment in the presence of recursion is surprisingly as good as Turing machines [31].

We think of the subsingleton fragment as a laboratory in which to study the properties and behaviors of least and greatest fixed points in their simplest nontrivial form, following the seminal work of Fortier and Santocanale [36].

1.3.3 Our approach

Our approach toward the thesis involves several layers:

1. The first step is to build prior work on infinitary linear logics with fixed points, and form a Curry-Howard correspondence between them and binary session types. We build the correspondence upon the infinitary system introduced by Fortier and Santoconale [36, 83] for singleton logic, a fragment of intuitionistic logic in which the antecedent consists of exactly one formula [27].

   Fortier and Santoconale [36, 83] extend the sequent calculus for singleton logic with rules for least and greatest fixed points. A naive extension, however, loses the cut elimination property (even when allowing infinite proofs) so they call derivations *pre-proofs*. *Circular pre-proofs* are distinguished as a subset of derivations which are *regular* in the sense that they can be represented as finite trees with loops. Fortier and Santocanale then impose a validity condition on pre-proofs to single out a class of pre-proofs that satisfy cut elimination. Moreover, they provide a cut elimination algorithm and show its productivity on valid derivations. In this thesis we call a pre-proof *FS-valid* if it satisfies Fortier and Santocanale's condition.

   We establish a correspondence between (mutually) recursive session-typed processes and circular pre-proofs in subsingleton logic with fixed points. We introduce a guard condition to check a stricter version of the FS-validity condition. Our condition is local in the sense that we check each process separately to be guarded, and it is stricter in the sense that it accepts a proper subset of the processes with an underlying valid proofs.

   We further introduce a synchronous computational semantics of cut reduction in subsingleton logic with fixed points in the context of session types, based on a key step in Fortier and Santocanale's cut elimination algorithm which is compatible with prior operational interpretation of session-typed programming languages [97]. We show that the *strong progress property* for session typed programs corresponds to productivity of the cut elimination property for their underlying derivations. Since the FS-validity condition satisfies productivity of cut elimination, our local guard condition ensures the strong progress property.

2. One shortcoming of building our local guard condition upon the FS-validity condition is that we cannot capture some interesting programs, although they enjoy the strong

progress property. Of course, as a corollary to the halting problem, no decidable guard condition can recognize all programs with the strong progress property. But, our goal is to capture more programs with this property as long as the guard condition is still effective, local, and elegant.

For generalizing the guard condition, we need to prove the strong progress property directly. The first step towards this goal is to formalize strong progress as a predicate independent from cut elimination of its underlying derivation, and prove it in a metalogic. Here is where our subgoal arises:

- **New metalogic.** To carry out our argument in a metalogic we need a calculus in which we can easily embed session-typed processes and define their operational behavior, which strongly suggests a linear metalogic. Moreover, the formalization of strong progress inherits the need for using nested least and greatest fixed points from the session types that it is defined upon. Furthermore, we must be able to prove properties formalized using nested fixed points. For these reasons, we decided to introduce a new metalogic: a calculus for first order intuitionistic multiplicative additive linear logic with fixed points and infinitary proofs [28]. Our metalogic also supports term equality. To avoid the general unification problem in the presence of higher-order terms with binding operators, which is undecidable and non-deterministic, we use higher-order patterns in the sense of Miller [67]. Miller's higher-order patterns inherit the good properties of first-order unification, e.g. a linear-time decision procedure and existence of most general unifiers, and are sufficient in many applications where representation requires binding operators.

  For data types mutually defined by induction and coinduction the separate principles of induction and coinduction are insufficient. One recent approach in type theory integrates induction and coinduction by pattern and copattern matching and explicit well-founded induction on ordinals [2], following a number of earlier representations of induction and coinduction in type theory [1]. Here, we pursue a different line of research in *linear logic* with fixed points. Our goal is to introduce a sequent calculus to reason about linear predicates defined as nested least and greatest fixed points. Instead of applying induction and coinduction principles directly, we follow the approach of Brotherston et al. [12] to allow circularity in derivations. Unlike Brotherston's calculus which only has least fixed points, we consider nested least and greatest fixed points. To ensure soundness of the proofs we impose a validity condition on our derivations. We generalize the cut elimination algorithm introduced by Fortier and Santocanale for infinitary singleton logic with fixed points to our setting. We prove that this algorithm is productive on first order valid derivations and produces a cut-free (possibly infinite) valid proof productively. An algorithm is productive if every piece of its output is generated in a finite number of steps.

  Our metalogic and the validity condition imposed upon its derivations are a generalization of Fortier and Santocanale's singleton logic and their validity condition, respectively. Baelde et al. [7, 33] introduced a validity condition on the pre-proofs in multiplicative-additive linear logic with fixed points and proved cut-elimination for valid derivations. Our results, when restricted to the propositional fragment, differ from Baelde et al.'s in the treatment of intuitionistic linear implication ($\multimap$) versus its classical counterpart ($\parr$). In Chapter 4, we will compare our work with Baelde et al.'s in more detail.

We follow the approach of processes-as-formulas to provide an asynchronous semantics for session-typed processes and a predicate for strong progress indexed by session types in our metalogic. We carry out the proof of strong progress in the metalogic [28]. In this logic, we can build an elegant derivation for the strong progress property of a process with clearly marked (simultaneous) inductive and coinductive steps. We form a bisimulation between the resulting metalogical derivation and the process typing derivation. This helps us to better understand the interplay between mutual inductive and coinductive steps in the proof of strong progress, and how they relate to the behavior of the program. Finally, we show that for a guarded process this derivations ensures strong progress of the process when executed with any synchronous scheduler.

We use this proof technique as a case study of the guard criterion established in prior step, but also of how to prove properties of programming languages in a metalanguage with *circular* proofs.

1.4 Synopsis

This thesis is split into two parts. The first part is dedicated to proof theory of infinitary systems for linear logic with fixed points. In the second part, we focus on the strong progress property of session-typed processes.

Part 1. Proof theory

- Chapter 2 provides a brief history of sequent calculus and a discussion on the importance of cut elimination. We review the proof system of a few linear logics that are of interest in this thesis: subsingleton logic, multiplicative additive linear logic, and first-order multiplicative additive linear logic.

- Chapter 3 provides a brief history of fixed-points in proof theory. We review the literature that generalizes linear logic to reason with least and greatest fixed points. We recall the system for infinitary subsingleton logic with fixed points [36] and infinitary multiplicative additive linear logic with fixed points [7, 33].

- Chapter 4 introduces our infinitary system for first order multiplicative additive linear logic ($FIMALL^{\infty}_{\mu,\nu}$). We describe our validity condition on derivations and provide a cut-elimination algorithm for valid derivations.

Part 2. Session typed processes

- Chapter 5 provides a background on session types. We recall the typing rules, operational semantics, and the type safety properties of session-typed processes. We also extend the Curry-Howard correspondence of derivations in infinitary subsingleton logic with fixed points as recursive communicating processes.
- Chapter 6 presents our local guard condition to check a stricter version of the Fortier and Santocanale's validity condition. Our condition is local in the sense that we check each process definition separately, and it is stricter in the sense that it accepts a proper subset of the processes that their underlying proofs are recognized by the FS validity condition. We provide a proof for strong progress of guarded subsingleton session-typed processes based on the Curry-Howard correspondence built in Chapter 5.
- Chapter 7 revisits strong progress for session-typed processes. Following the processes-as-formulas approach, we formalize strong progress as a predicate in $FIMALL^{\infty}_{\mu,\nu}$ defined using mutual least and greatest fixed points. We provide an infinitary proof for this predicate when defined over guarded processes in $FIMALL^{\infty}_{\mu,\nu}$.
- Chapter 8 briefly describes an implementation of our local guard condition for subsingleton session-typed processes in SML.

In Chapter 9 we conclude this thesis by discussing future lines of work.

Chapter 2

Preliminaries - Proof theory

This chapter reviews the foundations of proof systems of different logics used in this thesis. In the next chapter, we review the literature that generalizes these systems to reason with least and greatest fixed points.

2.1 A bit of history

The history of proof theory goes back to 1920, when Hilbert was pursuing formalizability of mathematics as part of his program. Hilbert and Bernays introduced the original natural deduction as an axiomatic proof system with axioms to eliminate and introduce each connective [49, 50]. Gentzen articulates these axioms as the introduction and elimination rules for each connective [39]. In Gentzen's Natural Deduction (ND), proofs are structured as trees. It has a novel feature that allows adding and discharging assumptions. The nodes in ND are sequents of the form $\Gamma \vdash A$, meaning that the proof of succedent formula A uses the assumptions in the set of formulas Γ. The genuinely logical actions in both elimination and introduction rules are taking place on the right (succedent A). Here we only formalize introduction and elimination rules for disjunction:

$$\oplus\mathtt{I} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash (A \oplus B)} \text{ and } \frac{\Gamma \vdash B}{\Gamma \vdash (A \oplus B)}$$

$$\oplus\mathtt{E} \qquad \frac{\Gamma \vdash (A \oplus B) \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

This calculus's essential property is *normalization* [77], which roughly is the ability to transform a proof into another that does not make any detours. In particular, a calculus's *consistency* is a corollary to its normalization. The proof of normalization is typically by induction on the structure of the derivation and the complexity of detour formulas.

Later, in 1934-35 Gentzen introduced another proof system known as Sequent Calculus (SC) [39, 39, 40]. Like ND, the proofs in an intuitionistic Sequent calculus are tree form structures with their nodes being sequents of the form $\Gamma \vdash A$. He assigned two rules to each connective; a left rule (L) to specify how to use an assumption (or antecedent) with that connective and a right rule (R) to prove a succedent with it. As a result, the logical actions in SC may occur in both left and right. The left and right rules for disjunction, for example, are:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\oplus R_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\oplus R_2 \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\oplus L$$

Other than the left and right rules for the connectives, SC has two rules to capture the meaning of logical consequence: Identity (Id) and Cut (Cut). Identity states that an assumption A is sufficient to achieve the succedent A. In contrast, Cut states that proving a formula A is enough to use it as an assumption.

$$\frac{}{A \vdash A}\textsc{Id} \qquad \frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\textsc{Cut}$$

Using these two judgmental rules, we can test that a connective's behavior specified by its left and right rules respects the meaning of pure logical consequence as understood by Gentzen [37, 73]. In the first test, we consider breaking down Identity for each connective compound formula into smaller subformulas. This test is called *Identity expansion*. For example, for disjunction, we have:

Identity expansion: $$\frac{}{A \oplus B \vdash A \oplus B}\textsc{Id} \quad \Rightarrow \quad \dfrac{\dfrac{\dfrac{}{A \vdash A}\textsc{Id}}{A \vdash A \oplus B}\oplus R_1 \quad \dfrac{\dfrac{}{B \vdash B}\textsc{Id}}{B \vdash A \oplus B}\oplus R_2}{A \oplus B \vdash A \oplus B}\oplus L$$

The second test is *Cut reduction*. It states that we can reduce a cut for each connective compound formula to its smaller subformulas. For disjunction, the test is as follows:

Cut reduction: $$\dfrac{\dfrac{\dfrac{\mathcal{T}_1}{\Gamma_1 \vdash A}}{\Gamma_1 \vdash A \oplus B}\oplus R_1 \quad \dfrac{\dfrac{\mathcal{T}_2}{\Gamma_2, A \vdash C} \quad \dfrac{\mathcal{T}_3}{\Gamma_2, B \vdash C}}{\Gamma_2, A \oplus B \vdash C}\oplus L}{\Gamma_1, \Gamma_2 \vdash C}\textsc{Cut} \quad \Rightarrow \quad \dfrac{\dfrac{\mathcal{T}_1}{\Gamma_1 \vdash A} \quad \dfrac{\mathcal{T}_2}{\Gamma_2, A \vdash C}}{\Gamma_1, \Gamma_2 \vdash C}\textsc{Cut}$$

where $\mathcal{T}_1$, for example, is the proof tree given for the sequent $\Gamma_1 \vdash A$.

Identity expansion and Cut reduction ensure that each connective's left and right rules are in harmony. They are both strong enough to match up with the other one [73, 85].

The analog of normalization in SC is the cut-elimination property originally called Hauptsatz ("main theorem") by Gentzen. It states that we can eliminate the use of Cut in any given proof.

Similar to normalization, cut elimination is of utmost importance. Cut elimination is necessary for the proof of the sub-formula property. This property bounds the logical complexity of formulas in the proof by the complexity of the formulas in the proved sequent (conclusion) and is crucial for proof search. Among other results, cut-elimination for a system ensures its consistency and completeness for the synthetic notion of logical consequence; it ensures that the calculus captures correct inferences.

Cut reductions for each connective are essential for eliminating Cuts in a proof. Usually, cut-elimination follows from cut reduction by a straightforward induction. However, in the next chapter, we will see that this is not always the case.

To fully describe Gentzen's sequent calculus, we need to mention three structural rules that he considered in his system:[1]

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \textsc{Weakening} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \textsc{Contraction} \qquad \frac{\Gamma_1, B, A, \Gamma_2 \vdash C}{\Gamma_1, A, B, \Gamma_2 \vdash C}\ \textsc{Exchange}$$

We are interested in a refinement of this calculus in which Weakening and Contraction rules are not necessarily applicable. This refinement is called *linear logic* and was first described by Girard [43]. Without weakening and contraction, formulas behave as resources that have to be used exactly once. Consuming (left rules) and producing (right rules) such resources can model changes in the states of computation.

In a linear proof system, additive ($\&$,$\oplus$) and multiplicative ($\otimes$, $\multimap$ in the intuitionistic setting, $⅋$ in the classical setting) connectives are distinguished and not equivalent. The contexts of premises remain the same in the additive rules, while in the multiplicative ones the contexts are divided.

2.2 Sequent calculi for linear logics

We focus on linear Sequent Calculus for two main reasons. First, it corresponds to the underlying structure of session-typed processes, the main subjects of this thesis. We will elaborate on this correspondence later in Chapter 5. Second, we need the ability of linear logic to represent state in order to model the behavior of session-typed processes (Chapters 4 and 7).

We first review a system in which sequents are restricted to those with at most one resource on the left, called subsingleton logic. Next, we lift this restriction and provide the calculus for intuitionistic Multiplicative Additive Linear Logic (MALL). Finally, we show the first-order generalization of it.

[1]The Exchange rule is *implicitly* admissible when Γ in sequent $\Gamma \vdash A$ is defined as a multiset of formulas.

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C}\ \textsc{Cut}$$

$$\frac{\omega \vdash A}{\omega \vdash A \oplus B}\ \oplus R_1 \quad \frac{\omega \vdash B}{\omega \vdash A \oplus B}\ \oplus R_2 \qquad \frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C}\ \oplus L$$

$$\frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \,\&\, B}\ \& R \qquad \frac{A \vdash C}{A \,\&\, B \vdash C}\ \& L_1 \quad \frac{B \vdash C}{A \,\&\, B \vdash C}\ \& L_2$$

$$\frac{}{\cdot \vdash 1}\ 1R \qquad \frac{\cdot \vdash C}{1 \vdash C}\ 1L$$

$$\frac{}{\omega \vdash T}\ \top R \qquad \frac{}{0 \vdash A}\ 0L$$

Figure 2.1: Sequent calculus for subsingleton logic.

2.2.1 Propositional subsingleton logic.

The basic judgment of the subsingleton sequent calculus has the form $\omega \vdash A$, where ω is either empty or a single proposition A. If we restrict ω to be exactly one formula, we get the singleton sequent calculus.

The syntax of propositions follows the grammar

$$A, B ::= A \oplus B \mid A \,\&\, B \mid \top \mid 1 \mid 0$$
$$\omega ::= \cdot \mid A$$

We summarize the subsingleton logic rules [73] in Figure 2.1. Our only connectives are disjunction ($\oplus$) and conjunction ($\&$). With the restriction of having at most one formula as an antecedent, the multiplicative connectives cannot be captured in the subsingleton setting. We will see the multiplicative connectives in the next section when this restriction is lifted.

We generalize $\oplus$ and $\&$ to be n-ary connectives $\oplus\{\ell : A_\ell\}_{\ell \in L}$ and $\&\{\ell : A_\ell\}_{\ell \in L}$, where L ranges over finite sets of labels denoted by ℓ and k.

$$A ::= \cdots \mid \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L}$$

The binary disjunction and conjunction are defined as $A \oplus B = \oplus\{\pi_1 : A, \pi_2 : B\}$ and $A\&B = \&\{\pi_1 : A, \pi_2 : B\}$. Constants 0 and $\top$ are defined as the nullary version of these connectives: $0 = \oplus\{\}$ and $\top = \&\{\}$.

$$\frac{\omega \vdash A_k \quad k \in L}{\omega \vdash \oplus\{\ell{:}A_\ell\}_{\ell \in L}}\ \oplus R \quad \frac{A_\ell \vdash C \quad \forall \ell \in L}{\oplus\{\ell{:}A_\ell\}_{\ell \in L} \vdash C}\ \oplus L \quad \frac{\omega \vdash A_\ell \quad \forall \ell \in L}{\omega \vdash \&\{\ell{:}A_\ell\}_{\ell \in L}}\ \& R \quad \frac{A_k \vdash C \quad k \in L}{\&\{\ell{:}A_\ell\}_{\ell \in L} \vdash C}\ \& L$$

To prove the cut-elimination property, we need to provide cut reductions for each connective. These reductions cover the cases where the cut-formula is a *principal* formula of a left rule in the first assumption of Cut rule and a right rule in the second one (we call them *internal*

$$\dfrac{\dfrac{}{\cdot \vdash 1}\,1R \qquad \dfrac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\,1L}{\Gamma \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \Gamma \vdash C$$

$$\dfrac{\dfrac{\omega \vdash A_k \quad k \in L}{\omega \vdash \oplus\{\ell{:}A_\ell\}_{\ell\in L}}\,\oplus R \qquad \dfrac{A_\ell \vdash C \quad \forall \ell \in L}{\oplus\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\oplus L}{\omega \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \dfrac{\omega \vdash A_k \quad A_k \vdash C}{\omega \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\omega \vdash A_\ell \quad \forall \ell \in L}{\omega \vdash \&\{\ell{:}A_\ell\}_{\ell\in L}}\,\& R \qquad \dfrac{A_k \vdash C \quad k \in L}{\&\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\& L}{\omega \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \dfrac{\omega \vdash A_k \quad A_k \vdash C}{\omega \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\cdot \vdash B}{1 \vdash B}\,1L \qquad B \vdash C}{1 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{\cdot \vdash B \quad B \vdash C}{\cdot \vdash C}\,\textsc{Cut}}{1 \vdash C}\,1L$$

$$\dfrac{\omega \vdash B \qquad \dfrac{B \vdash A_k \quad k \in L}{B \vdash \oplus\{\ell{:}A_\ell\}_{\ell\in L}}\,\oplus R}{\omega \vdash \oplus\{\ell{:}A_\ell\}_{\ell\in L}}\,\textsc{Cut} \quad \xRightarrow{\text{RFLip}} \quad \dfrac{\dfrac{\omega \vdash B \quad B \vdash A_k}{\omega \vdash A_k}\,\textsc{Cut} \qquad k \in L}{\omega \vdash \oplus\{\ell{:}A_\ell\}_{\ell\in L}}\,\oplus R$$

$$\dfrac{\dfrac{A_\ell \vdash B \quad \forall \ell \in L}{\oplus\{\ell{:}A_\ell\}_{\ell\in L} \vdash B}\,\oplus L \qquad B \vdash C}{\oplus\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{A_\ell \vdash B \quad B \vdash C}{A_\ell \vdash C}\,\textsc{Cut} \qquad \forall \ell \in L}{\oplus\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\oplus L$$

$$\dfrac{\omega \vdash B \qquad \dfrac{B \vdash A_\ell \quad \ell \in L}{B \vdash \&\{\ell{:}A_\ell\}_{\ell\in L}}\,\& R}{\omega \vdash \&\{\ell{:}A_\ell\}_{\ell\in L}}\,\textsc{Cut} \quad \xRightarrow{\text{RFLip}} \quad \dfrac{\dfrac{\omega \vdash B \quad B \vdash A_\ell}{\omega \vdash A_\ell}\,\textsc{Cut} \qquad \forall \ell \in L}{\omega \vdash \&\{\ell{:}A_\ell\}_{\ell\in L}}\,\& R$$

$$\dfrac{\dfrac{A_k \vdash B \quad k \in L}{\&\{\ell{:}A_\ell\}_{\ell\in L} \vdash B}\,\& L \qquad B \vdash C}{\&\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{A_k \vdash B \quad B \vdash C}{A_k \vdash C}\,\textsc{Cut} \qquad k \in L}{\&\{\ell{:}A_\ell\}_{\ell\in L} \vdash C}\,\& L$$

Figure 2.2: Internal and External reductions for subsingleton logic

reductions). We also need to consider *external reductions* or *Flip* rules to cover other cases in the proof of cut-elimination. We provide a full list of internal and external reductions for subsingleton logic in Figure 2.2.

Theorem 2.1 (Cut admissibility)**.** *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are cut-free proofs for $\omega \vdash A$ and $A \vdash C$, respectively, we can build a cut-free proof for $\omega \vdash C$.*

Proof. The proof is by a lexicographic induction on the structure of formula A and structure of $\mathcal{T}_1$ and $\mathcal{T}_2$. If the last step in $\mathcal{T}_1$ or $\mathcal{T}_2$ is Id, we can eliminate it outright:

$$\dfrac{\dfrac{}{A \vdash A}\,\textsc{Id} \qquad A \vdash C}{A \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{ID}-\text{Elim}} \quad A \vdash C$$

$$\dfrac{\omega \vdash A \qquad \dfrac{}{A \vdash A}\,\textsc{Id}}{\omega \vdash A}\,\textsc{Cut} \quad \xRightarrow{\text{ID}-\text{Elim}} \quad \omega \vdash A$$

If the last step in $\mathcal{T}_1$ is a right rule (applied on A) and the last step in $\mathcal{T}_2$ is a left rule (applied on A), we use the corresponding internal cut reduction and apply the inductive hypothesis. Otherwise, either the last step in $\mathcal{T}_1$ is a left rule or the last step in $\mathcal{T}_2$ is a right rule, in each of these cases we apply the corresponding external reduction. For a complete proof see [73]. □

Theorem 2.2 (Cut elimination). *Sequent $\omega \vdash C$ is provable in subsingleton logic if and only if it is provable without the cut rule.*

Proof. The proof is by induction on the structure of the proof given for $\omega \vdash C$ and is a straightforward corollary of cut-admissibility. □

The proof of the cut-elimination theorem provides a mechanical algorithm to remove cut from any given derivation. The cut-free derivation has a subformula property meaning that any formula appearing in a derivation for $\omega \vdash C$ is a subformula of either ω or C. Consistency is a straightforward corollary of this property: there is no proof in subsingleton logic for $\cdot \vdash 0$.

2.2.2 Propositional intutitionistic multiplicative additive linear logic.

Derivations in Intuitionistic Multiplicative Additive Linear Logic (IMALL) establish judgments of the form $\Gamma \vdash A$ where Γ is an unordered list of formulas. The syntax of formulas follows the grammar

$$A \quad ::= \quad 1 \mid A \otimes A \mid A \multimap A \mid \oplus\{\ell{:}A_\ell\}_{\ell \in L} \mid \&\{\ell{:}A_\ell\}_{\ell \in L}.$$

The sequent calculus for this logic, originally presented by Girard [41], is given in Figure 4.1. Intuitionistic sequents in the calculus of Figure 4.1 are restricted to have only one formula as the succedent. Classical Multiplicative Additive Linear Logic (MALL) is obtained by allowing a set of formulas for the succedent instead: $\Gamma \vdash \Delta$. In the classical setting, we have an alternative disjunction $⅋$, the counterpart of $\multimap$ in the intuitionistic framework:

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A ⅋ B}\, ⅋R \qquad\qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A ⅋ B \vdash \Delta, \Delta'}\, ⅋R$$

There is an equivalent one-sided representation for classical MALL, in which the sequents are of the form $\vdash \Delta$. A proof for $\vdash \Delta$ in the one-sided calculus implies $\cdot \vdash \Delta$ in a two-sided system of classical MALL, and $\Gamma \vdash \Delta$ in the two-sided calculus implies $\vdash \Gamma^\perp, \Delta$ in the one-sided system. Where the involution of formulas by negation satisfies:

$$\begin{array}{cc} \oplus\{\ell{:}A_\ell\}_{\ell \in L}^\perp = \&\{\ell{:}A_\ell^\perp\}_{\ell \in L} & \&\{\ell{:}A_\ell\}_{\ell \in L}^\perp = \oplus\{\ell{:}A_\ell^\perp\}_{\ell \in L} \\ (A \otimes B)^\perp = A^\perp ⅋ B^\perp & (A ⅋ B)^\perp = A^\perp \otimes B^\perp \end{array}$$

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{\Gamma \vdash A \quad \Gamma', A \vdash C}{\Gamma, \Gamma' \vdash C}\ \textsc{Cut}$$

$$\frac{}{\cdot \vdash 1}\ 1R \qquad \frac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\ 1L$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma' \vdash A_2}{\Gamma, \Gamma' \vdash A_1 \otimes A_2}\ \otimes R \qquad \frac{\Gamma, A_1, A_2 \vdash B}{\Gamma, A_1 \otimes A_2 \vdash B}\ \otimes L$$

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \multimap A_2}\ \multimap R \qquad \frac{\Gamma \vdash A_1 \quad \Gamma', A_2 \vdash B}{\Gamma, \Gamma', A_1 \multimap A_2 \vdash B}\ \multimap L$$

$$\frac{\Gamma \vdash A_k \quad k \in I}{\Gamma \vdash \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R \qquad \frac{\Gamma, A_i \vdash B \quad \forall i \in I}{\Gamma, \oplus\{l_i : A_i\}_{i \in I} \vdash B}\ \oplus L$$

$$\frac{\Gamma \vdash A_i \quad \forall i \in I}{\Gamma \vdash \&\{l_i : A_i\}_{i \in I}}\ \& R \qquad \frac{\Gamma, A_k \vdash B \quad k \in I}{\Gamma, \&\{l_i : A_i\}_{i \in I} \vdash B}\ \& L$$

FIGURE 2.3: Propositional intuitionistic MALL

The cut-elimination theorem can be proved for IMALL and MALL, similar to Theorem 2.2 for subsingleton logic. Internal and External cut reductions needed for the proof of IMALL are given in Figure 2.4.[2]

2.2.3 First-order intutitionistic multiplicative additive linear logic.

The grammar

$$A \quad ::= \quad \cdots \mid \exists x.\, A(x) \mid \forall x.\, A(x) \mid s = t \mid T(\bar{t}),$$

extends intuitionistic multiplicative additive linear logic to a first-order language. s, t stand for terms, $\bar{t}$ for a sequence of terms, and x, y for term variables. For our purposes, no grammar is specified for terms; all terms are of the only type U with binders. $T(\bar{t})$ is an instance of a predicate. The rules for the first order extensions are given in Figure 2.5.

Our first-order quantifiers and equality rules follow the standard representation of the rules in a linear logic with fixed points presented in [8]. In the $= L$ rule, mgu stands for the most general unifier. A substitution σ is a function that replaces variables by terms. σ unifies two terms t and s if $t[\sigma] = s[\sigma]$. A substitution, θ, is a most general unifier of t and s if it unifies t and s, and for any unifier σ for t and s, there is a unifier λ, such that $\sigma = \theta \circ \lambda$, where $\circ$ stands for the composition of θ and λ as functions. We restrict our terms to Miller's [67] higher-order patterns: an extension of first-order formulas including bound variable names and scopes. With this restriction, we ensure that we have the most general unifier when unifiers exist. In this setting, the set $\mathtt{mgu}(t, s)$ in $= L$ is either empty, or a singleton set containing a most general unifier.

[2]This thesis focuses on intuitionistic systems, so we only provide cut reductions for IMALL.

$$\dfrac{\dfrac{}{\cdot \vdash 1}\,1R \quad \dfrac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\,1L}{\Gamma \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{Reduce}} \quad \Gamma \vdash C$$

$$\dfrac{\dfrac{\Gamma_1' \vdash A_1 \quad \Gamma_2' \vdash A_2}{\Gamma_1', \Gamma_2' \vdash A_1 \otimes A_2}\,\otimes R \quad \dfrac{\Gamma'', A_1, A_2 \vdash B}{\Gamma'', A_1 \otimes A_2 \vdash B}\,\otimes L}{\Gamma_1', \Gamma_2', \Gamma'' \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{Reduce}} \quad \dfrac{\Gamma_1' \vdash A_1 \quad \Gamma_2' \vdash A_2 \quad \Gamma'', A_1, A_2 \vdash B}{\Gamma_1', \Gamma_2', \Gamma'' \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\Gamma', A_1 \vdash A_2}{\Gamma' \vdash A_1 \multimap A_2}\multimap R \quad \dfrac{\Gamma_1'' \vdash A_1 \quad \Gamma_2'', A_2 \vdash B}{\Gamma_1'', \Gamma_2'', A_1 \multimap A_2 \vdash B}\multimap L}{\Gamma', \Gamma_1'', \Gamma_2'' \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{Reduce}} \quad \dfrac{\Gamma_1'' \vdash A_1 \quad \Gamma', A_1 \vdash A_2 \quad \Gamma_2'', A_2 \vdash B}{\Gamma', \Gamma_1'', \Gamma_2'' \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\Gamma_1 \vdash A_k \quad k \in L}{\Gamma_1 \vdash \oplus\{\ell{:}A_\ell\}_{\ell \in L}}\,\oplus R \quad \dfrac{\Gamma_2, A_\ell \vdash C \quad \forall \ell \in L}{\Gamma_2, \oplus\{\ell{:}A_\ell\}_{\ell \in L} \vdash C}\,\oplus L}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{Reduce}} \quad \dfrac{\Gamma_1 \vdash A_k \quad \Gamma_2, A_k \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\Gamma_1 \vdash A_\ell \quad \forall \ell \in L}{\Gamma_1 \vdash \&\{\ell{:}A_\ell\}_{\ell \in L}}\,\& R \quad \dfrac{\Gamma_2, A_k \vdash C \quad k \in L}{\Gamma_2, \&\{\ell{:}A_\ell\}_{\ell \in L} \vdash C}\,\& L}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{Reduce}} \quad \dfrac{\Gamma_1 \vdash A_k \quad \Gamma_2, A_k \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut}$$

$$\dfrac{\Gamma' \vdash B \quad \dfrac{\Gamma_1'', B \vdash A_1 \quad \Gamma_2'' \vdash A_2}{\Gamma_1'', \Gamma_2'', B \vdash A_1 \otimes A_2}\,\otimes R}{\Gamma', \Gamma_1'', \Gamma_2'' \vdash A_1 \otimes A_2}\,\textsc{Cut} \quad \xRightarrow{\textsf{RFLip}} \quad \dfrac{\dfrac{\Gamma' \vdash B \quad \Gamma_1'', B \vdash A_1}{\Gamma', \Gamma_1'', B \vdash A_1}\,\textsc{Cut} \quad \Gamma_2'' \vdash A_2}{\Gamma', \Gamma_1'', \Gamma_2'' \vdash A_1 \otimes A_2}\,\otimes R$$

$$\dfrac{\dfrac{\Gamma', A_1, A_2 \vdash B}{\Gamma', A_1 \otimes A_2 \vdash B}\,\otimes L \quad \Gamma'', B \vdash C}{\Gamma', \Gamma'', A_1 \otimes A_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{LFLip}} \quad \dfrac{\dfrac{\Gamma', A_1, A_2 \vdash B \quad \Gamma'', B \vdash C}{\Gamma', A_1, A_2 \vdash C}\,\textsc{Cut}}{\Gamma', \Gamma'', A_1 \otimes A_2 \vdash C}\,\otimes L$$

$$\dfrac{\Gamma' \vdash B \quad \dfrac{\Gamma'', B, A_1 \vdash A_1}{\Gamma'', B \vdash A_1 \multimap A_2}\multimap R}{\Gamma', \Gamma'' \vdash A_1 \multimap A_2}\,\textsc{Cut} \quad \xRightarrow{\textsf{RFLip}} \quad \dfrac{\dfrac{\Gamma' \vdash B \quad \Gamma'', B, A_1 \vdash A_2}{\Gamma', \Gamma'', A_1 \vdash A_2}\,\textsc{Cut}}{\Gamma', \Gamma'' \vdash A_1 \multimap A_2}\multimap R$$

$$\dfrac{\dfrac{\Gamma_1' \vdash A_1 \quad \Gamma_2', A_2 \vdash B}{\Gamma_1', \Gamma_2', A_1 \multimap A_2 \vdash B}\multimap L \quad \Gamma'', B \vdash C}{\Gamma_1', \Gamma_2', \Gamma'', A_1 \multimap A_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\textsf{LFLip}} \quad \dfrac{\Gamma_1' \vdash A_1 \quad \dfrac{\Gamma_2', A_2 \vdash B \quad \Gamma'', B \vdash C}{\Gamma_2', \Gamma'', A_2 \vdash C}\,\textsc{Cut}}{\Gamma_1', \Gamma_2', \Gamma'', A_1 \multimap A_2}\multimap L$$

FIGURE 2.4: Internal cut reductions and selected external cut reductions for IMALL

The following derivation is an example of $= L$ application when the most general unifier of the terms is an empty set:

$$\dfrac{\dfrac{}{\mathbf{z} = \mathsf{s}x \vdash A}\,= L}{\forall x.(\mathsf{z} = \mathsf{s}x) \vdash A}\,\forall L$$

Cut reductions for the first-order components of first-order intutitionistic multiplicative additive linear logic (FIMALL) are given in Figure 2.6. The proof of cut elimination is similar to the previous fragments of linear logic by induction on the structure of derivations.

$$\dfrac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x.P(x)}\,\exists R \qquad \dfrac{\Gamma, P(x) \vdash B \quad x \text{ fresh}}{\Gamma, \exists x.P(x) \vdash B}\,\exists L_x$$

$$\dfrac{\Gamma \vdash P(x) \quad x \text{ fresh}}{\Gamma \vdash \forall x.P(x)}\,\forall R_x \qquad \dfrac{\Gamma, P(t) \vdash B}{\Gamma, \forall x.P(x) \vdash B}\,\forall L$$

$$\dfrac{}{\cdot \vdash s = s}\,{=}R \qquad \dfrac{\Gamma[\theta] \vdash B[\theta] \quad \forall \theta \in \mathtt{mgu}(t, s)}{\Gamma, s = t \vdash B}\,{=}L$$

FIGURE 2.5: First-order extension of IMALL.

$$\dfrac{\dfrac{\Gamma_1 \vdash P(t)}{\Gamma_1 \vdash \exists x.P(x)}\,\exists R \quad \dfrac{\overset{\mathcal{T}}{\Gamma_2, P(x) \vdash B}}{\Gamma_2, \exists x.P(x) \vdash B}\,\exists L}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \dfrac{\Gamma_1 \vdash P(t) \quad \overset{\mathcal{T}[t/x]}{\Gamma_2, P(t) \vdash B}}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{\overset{\mathcal{T}}{\Gamma_1 \vdash P(x)}}{\Gamma_1 \vdash \forall x.P(x)}\,\forall R \quad \dfrac{\Gamma_2, P(t) \vdash B}{\Gamma_2, \forall x.P(x) \vdash B}\,\forall L}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \dfrac{\overset{\mathcal{T}[t/x]}{\Gamma_1 \vdash P(t)} \quad \Gamma_2, P(t) \vdash B}{\Gamma_1, \Gamma_2 \vdash C}\,\textsc{Cut}$$

$$\dfrac{\dfrac{}{\cdot \vdash s = s}\,= R \quad \dfrac{\Gamma \vdash C}{\Gamma, s = s \vdash C}\,= L}{\Gamma \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{Reduce}} \quad \Gamma \vdash C$$

$$\dfrac{\Gamma' \vdash B \quad \dfrac{\Gamma''_1 \vdash P(t)}{\Gamma'', B \vdash \exists x.P(x)}\,\exists R}{\Gamma', \Gamma'' \vdash \exists x.P(x)}\,\textsc{Cut} \quad \xRightarrow{\text{RFLip}} \quad \dfrac{\dfrac{\Gamma' \vdash B \quad \Gamma'', B \vdash P(t)}{\Gamma', \Gamma'' \vdash P(t)}\,\textsc{Cut}}{\Gamma', \Gamma'' \vdash \exists x.P(x)}\,\exists R$$

$$\dfrac{\dfrac{\Gamma', P(x) \vdash B}{\Gamma', \exists x.P(x) \vdash B}\,\exists L \quad \Gamma'', B \vdash C}{\Gamma', \Gamma'', \exists x.P(x) \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{\Gamma', P(x) \vdash B \quad \Gamma'', B \vdash C}{\Gamma', \Gamma'', P(x) \vdash C}\,\textsc{Cut}}{\Gamma', \Gamma'', \exists x.P(x) \vdash C}\,\exists L$$

$$\dfrac{\Gamma' \vdash B \quad \dfrac{\Gamma'', B \vdash P(x)}{\Gamma'', B \vdash \forall x.P(x)}\,\forall R}{\Gamma', \Gamma'' \vdash \forall x.P(x)}\,\textsc{Cut} \quad \xRightarrow{\text{RFLip}} \quad \dfrac{\dfrac{\Gamma' \vdash B \quad \Gamma'', B \vdash P(x)}{\Gamma', \Gamma'' \vdash P(x)}\,\textsc{Cut}}{\Gamma', \Gamma'' \vdash \forall x.P(x)}\,\forall R$$

$$\dfrac{\dfrac{\Gamma', P(t) \vdash B}{\Gamma', \forall x.P(x) \vdash B}\,\forall L \quad \Gamma'', B \vdash C}{\Gamma', \Gamma'', \forall x.P(x) \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{\Gamma' \vdash B \quad \Gamma'', B \vdash P(t)}{\Gamma', \Gamma'' \vdash P(t)}\,\textsc{Cut}}{\Gamma', \Gamma'', \forall x.P(x)}\,\forall L$$

$$\dfrac{\dfrac{\Gamma'[\theta] \vdash B[\theta] \quad \forall \theta \in \mathtt{mgu}(t, s)}{\Gamma', s = t \vdash B}\,= L \quad \overset{\mathcal{T}}{\Gamma'', B \vdash C}}{\Gamma', \Gamma'', s = t \vdash C}\,\textsc{Cut} \quad \xRightarrow{\text{LFLip}} \quad \dfrac{\dfrac{\Gamma'[\theta] \vdash B[\theta] \quad \overset{\mathcal{T}[\theta]}{\Gamma''[\theta], B[\theta] \vdash C[\theta]}}{\Gamma'[\theta], \Gamma''[\theta] \vdash C[\theta]}\,\textsc{Cut} \quad \forall \theta \in \mathtt{mgu}(t, s)}{\Gamma', \Gamma'', s = t \vdash C}\,= L$$

FIGURE 2.6: Selected internal and external cut reductions for FIMALL

Chapter 3

Preliminaries - Fixed points in logic

3.1 A bit of history

Inductive reasoning is well known and presented in the literature in many different contexts. In general, it is used to prove properties about inductively defined structures such as natural numbers, finite lists, and finite trees. Computer scientists also use induction to specify and reason about recursive programs' behavior. The usual induction schema for induction over the natural numbers is the following:

$$\frac{P(z) \quad \forall x{\in}\mathbb{N}\,(P(x) \to P(s(x)))}{P(t)}$$

where z stands for $0 \in \mathbb{N}$ and $s(x)$ for successor of $x \in \mathbb{N}$.

Several approaches were introduced throughout the proof theory history that embraces inductive reasoning in a proof system. Hilbert in his 1930 Hamburg talk extends the elementary arithmetic by a finitist rule called Hilbert's rule (HR) to introduce universally quantified formulas. Essentially, it asserts that a universally quantified formula is justified when all of its instances are justified. Later in 1931, Bernays, inspired by the HR rule, proposed an infinitary rule (ω-rule) for universal quantifier that captures complete induction [87, 88]. Lorenzen, Schütte, and Tait further developed the schema of ω-rule [58, 61, 94]. In a sequent calculus system, this schema is as the following rules:

$$\frac{\Gamma \vdash A(n) \quad \textit{for all } n \in \mathbb{N}}{\Gamma \vdash \forall x.A(x)}\;\forall R \qquad\qquad \frac{\Gamma, A(n) \vdash C \quad \textit{for some } n \in \mathbb{N}}{\Gamma, \forall x.A(x) \vdash C}\;\forall L$$

The $\forall\, R$ rule is an infinitary rule; it requires infinitely many premises. Alternatively, the universally quantified formula $\forall x.A(x)$ can be translated as an infinitary conjunction $\Pi_n A(n)$.

Proofs in the calculus are *infinitely branching* but *still well-founded* trees. This extension is strong enough to capture inductive reasoning. As with other sequent calculi, cut-elimination is of utmost importance for the system's consistency and has been studied for this extension. Tait's proof for cut-elimination of infinitary classical propositional logic is one particular result [94].

A different approach entails adding rules for unfolding inductive definitions to the calculus. This approach has its roots in Martin Löfs' natural deduction system with iterated inductive definitions [62], Girard's description of linear logic with fixed points [44], and Eriksson [35] and Schroeder-Heister definitional reflection [84]. It is developed further by McDowell and Miller 2000 [64], and Momigliano and Tiu [71].

There are two main methods for adding unfolding rules of inductive definitions (least fixed points) to the proof system. The first method is closely associated with the induction principle and yields a finitary system. The other is related to infinite descent and produces a non-wellfounded infinitary system (but finitely branching).

A typical schema for the unfolding rules of least fixed points in a finitary system is as follows:

$$\frac{\Gamma \vdash A[\mu x.A/x]}{\Gamma \vdash \mu x.A}\ \mu R \qquad\qquad \frac{A[B/x] \vdash B \quad \Gamma, B \vdash C}{\Gamma, \mu x.A \vdash C}\ \mu L$$

where x is a propositional variable.

This set of rules is a variant of Park's rules; in contrast to original Park's rules, they do not break cut admissibility. Cut elimination for such finitary systems is proved in different contexts, e.g. [8]. However, the subformula property is necessarily lost because of the left fixed-point rule. As a result, consistency is not a straightforward corollary of cut-elimination in such systems anymore.

Unfolding rules for the least fixed points in an infinite system are usually more straightforward and respect the subformula property:

$$\frac{\Gamma \vdash A[\mu x.A/x]}{\Gamma \vdash \mu x.A}\ \mu R \qquad\qquad \frac{\Gamma, A[\mu x.A/x] \vdash C}{\Gamma, \mu x.A \vdash C}\ \mu L$$

This set of rules in a finitary system is not strong enough to prove all derivations in a calculus with the previous set of fixed-point rules [33]. To revive this strength, we allow the system to be infinitary.

Cut reductions for matching left and right rules hold in the infinitary calculus. However, the non-wellfoundedness of infinite derivations breaks the induction needed to prove cut-elimination. Without the cut-elimination property, derivations are not proper proofs.

To establish the cut-elimination property, an additional soundness condition is imposed on infinitary derivations. A typical soundness condition on an infinitary calculus ensures that *some inductive definition unfolds infinitely often along each infinite branch.* Sound derivations are called proofs. Cut-elimination is proved for *proofs* in infinitary calculi in different contexts [7, 12, 33].

Even though infinitary derivations can be "infinite," an interesting subset of them can be expressed as finite trees with loops. A *circular derivation* is the finite representation of an infinite one in which we can identify each open subgoal with an identical interior judgment.

This thesis focuses on **non-wellfounded infinitary sequent calculi with fixed points**. Our interest in such calculi is rooted in the close correspondence of circular derivations to recursive programs and cyclic reasoning employed in computer science. We elaborate more on this correspondence in Chapter 7.4.

In this historical section, we only considered inductive definitions. However, our general interest is more extensive: we want to include coinductive definitions (greatest fixed points) in our sequent calculus. To show properties of coinductive definitions, e.g. streams and infinite trees, a dual principle of coinduction is needed. Although this dual principle has been used in the literature before, David Park was the first one who explicitly used greatest fixed points in a non-trivial form. For an intensive historical review of coinduction and greatest fixed points, see the paper titled "*On the Origins of Bisimulation and Coinduction*" by Sangiorgi [80].

Similar to the least fixed points, we can introduce the rules for greatest fixed points in both finitary and infinitary systems. The following is a typical schema for the unfolding rules of greatest fixed points in a finitary system:

$$\frac{\Gamma \vdash B \quad B \vdash A[B/x]}{\Gamma \vdash \nu x.A}\ \nu R \qquad\qquad \frac{\Gamma, A[\nu x.A/x] \vdash C}{\Gamma, \nu x.A \vdash C}\ \nu L$$

where x is a propositional variable. By adding this set of rules, cut admissibility remains intact. The subformula property is lost also because of the right greatest-point rule.

The typical unfolding rules for the greatest fixed points in an infinite system is similar to the rules we provided for the least fixed points and respect the subformula property:

$$\frac{\Gamma \vdash A[\nu x.A/x]}{\Gamma \vdash \nu x.A}\ \nu R \qquad\qquad \frac{\Gamma, A[\nu x.A/x] \vdash C}{\Gamma, \nu x.A \vdash C}\ \nu L$$

To summarize, we are interested in non-wellfounded infinitary sequent calculi that integrate mutually defined least and greatest fixed points. The rest of this chapter is devoted to reviewing such calculi presented previously.

3.2 Mutual fixed points and priorities

The celebrated Knaster-Tarski theorem[95] guarantees existence of both least and greatest fixed points of a monotone operator in a complete lattice:

Definition 3.1. A lattice L with a partial order $\leq$ is called complete if every subset A of L has a least upper bound (join) and a greatest lower bound (meet).

Definition 3.2. A mapping $f : L \to L$ is monotonic if $\forall x, y \in L.\, x \leq y$ implies $f(x) \leq f(y)$.

Theorem 3.3 (Knaster-Tarski fixed point theorem)**.** *For a complete lattice L and a monotone operator $f : L \to L$, the least fixed point $\mu x.f(x) = \bigwedge\{u{\in}L : f(u) \leq u\}$ and the greatest fixed point $\nu x.f(x) = \bigvee\{u{\in}L : u \leq f(u)\}$ exist.*

This result can be generalized to n-ary operators $f{:}L^n \to L$ monotonic in all variables.

It is well-known that for a monotonic operator $f{:}L^{n+1} \to L$, the fixed-points $\mu x.f(x, \bar{y})$ and $\nu x.f(x, \bar{y})$ are also monotonic operators from L^n to L. As a result, we can consider nested fixed-points such as $\mu x.\, \nu x.\, (1 \oplus x \oplus y)$ [6].

Example 3.1. *1. $\mu x.1 \oplus x$ represents natural numbers.*

2. $\nu y.(\mu x.1 \oplus x) \otimes y$ represents a stream of natural numbers.

As an alternative to the vectorial presentation of fixed points as $\mu x.f(x)$ for least fixed-points and $\nu x.f(x)$ for greatest fixed points, we can use *systems of equations*. We use an example to explain this alternative representation's key ideas; for more information, see [86].

The formula $\nu y.(\mu x.x \oplus y) \otimes y$ can be represented using equations $y =_\nu x \otimes y$ and $x =_\mu x \oplus y$. To ensure that this signature truly represents the formula $\nu y.(\mu x.x \oplus y) \otimes y$ (and not $\mu x.x \oplus (\nu y.x \otimes y)$), we need to impose a linear ordering on the equations. The importance of this ordering is apparent from the following inequality:

$$\nu y.(\mu x.x \oplus y) \otimes y \neq \mu x.x \oplus (\mu y.x \otimes y).$$

We call the place of an equation in this ordering its priority and present it as a superscript on the equation: $y =^1_\nu x \otimes y$ and $x =^2_\mu x \oplus y$. We will see that priorities provide central information to determine validity of circular proofs.

Priorities are not significant when comparing two least fixed points and similarly two greatest fixed points:

$$\mu x.\mu y.f(x, y) = \mu y.\mu x.f(x, y).$$

As a result, a signature of fixed-point equations is divided into "layers" within which only fixed-points of the same kind occur. In other words, relational priorities form a heirarchy analogous to quantifiers in logic [86].

Using systems of equations to represent fixed points is more in line with recursive definitions of data types in programming languages. Moreover, it is more flexible in the sense that we can reformulate a signature only by changing the priorities, without altering the fixed-point formulas.

We conclude this section by providing an example of a system of equations, also called signature.

Example 3.2. *The signature*

$$\Sigma = \{\mathsf{stream} =^1_\nu \mathsf{nat} \otimes \mathsf{stream}, \quad \mathsf{nat} =^2_\mu 1 \oplus \mathsf{nat}\}$$

describes natural numbers nat *and stream of natural numbers* stream.

3.3 Subsingleton logic with fixed points

The expressive power of pure subsingleton logic described in Section 2.2.1 is rather limited, among other things due to the absence of the exponential $!A$. However, we can recover significant expressive power by adding least and greatest fixed points, which can be done without violating the subsingleton restriction.

Fortier and Santocanale [36] introduce an extension of singleton logic with the least and greatest fixed points. This section summarizes the fundamental ideas of Fortier and Santocanale's seminal work. However, we allow some deviations from their original formulation. For instance, we generalize this result for the subsingleton logic.

The syntax of propositions follows the grammar

$$A ::= \oplus\{\ell{:}A_\ell\}_{\ell \in L} \mid \&\{\ell{:}A_\ell\}_{\ell \in L} \mid 1 \mid t$$

where t ranges over a set of propositional variables denoting least or greatest fixed points. We define them in a *signature* Σ which records some important additional information, namely their relative *priority*.

$$\Sigma ::= \cdot \mid \Sigma, t =^i_\mu A \mid \Sigma, t =^i_\nu A,$$

with the conditions that

- if $t =^i_a A \in \Sigma$ and $t' =^i_b B \in \Sigma$, then $a = b$, and
- if $t =^i_a A \in \Sigma$ and $t =^j_b B \in \Sigma$, then $i = j$ and $A = B$.

For a fixed point t defined as $t =^i_a A$ in Σ the subscript a is the *polarity* of t: if $a = \mu$, then t is a fixed point with *positive* polarity and if $a = \nu$, then it is of *negative* polarity. Finitely representable least fixed points (e.g., natural numbers and lists) can be represented in this system as defined propositional variables with positive polarity, while the potentially infinite greatest fixed points (e.g., streams and infinite depth trees) are represented as those with negative polarity.

The superscript i is the *priority* of t. Fortier and Santocanale interpreted the priority of fixed points in their system as the order in which the least and greatest fixed point equations are solved in the categorical semantics [36, 81]. They also used them syntactically as central information to determine validity of circular proofs.

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C}\ \textsc{Cut}$$

$$\frac{\omega \vdash A_k \quad k \in L}{\omega \vdash \oplus\{A_\ell\}_{\ell \in L}}\ \oplus R \qquad \frac{A_\ell \vdash C \quad \forall \ell \in L}{\oplus\{A_\ell\}_{\ell \in L} \vdash C}\ \oplus L$$

$$\frac{\omega \vdash A_\ell \quad \forall \ell \in L}{\omega \vdash \&\{A_\ell\}_{\ell \in L}}\ \& R \qquad \frac{A_k \vdash C \quad k \in L}{\oplus\{A_\ell\}_{\ell \in L} \vdash C}\ \& L_1$$

$$\frac{}{\cdot \vdash 1}\ 1R \qquad \frac{\cdot \vdash C}{1 \vdash C}\ 1L$$

$$\frac{\omega \vdash A \quad t =^i_\mu A \in \Sigma}{\omega \vdash t}\ \mu R \qquad \frac{A \vdash C \quad t =^i_\mu A \in \Sigma}{t \vdash C}\ \mu L$$

$$\frac{\omega \vdash A \quad t =^i_\nu A \in \Sigma}{\omega \vdash t}\ \nu R \qquad \frac{A \vdash C \quad t =^i_\nu A \in \Sigma}{t \vdash C}\ \nu L$$

FIGURE 3.1: Infinitary sequent calculus for subsingleton logic with fixed points.

We write $p(t) = i$ for the priority of t, and $\epsilon(i) = a$ for the polarity of propositional variable t with priority i. The condition on Σ ensures that ϵ is a well-defined function.

The basic judgment of the subsingleton sequent calculus has the form $\omega \vdash_\Sigma A$, where ω is either empty or a single proposition A and Σ is a signature. Since the signature never changes in the rules, we omit it from the turnstile symbol.

The rules of subsingleton logic with fixed points are summarized in Figure 3.1. We added the fixed points in the two last rows to Figure 2.1. This set of rules must be interpreted as *infinitary*, meaning that a judgment may have an infinite derivation in this system.

Even a cut-free derivation may be of infinite length since each defined propositional variable may be unfolded infinitely many times. Also, cut elimination no longer holds for the derivations after adding fixed point rules. What the rules define then are the so-called *pre-proofs*. In particular, we are interested in *circular pre-proofs*, which are the pre-proofs that can be illustrated as finite trees with loops [33].

Fortier and Santocanale [36] introduced a *validity condition* for identifying *proofs* among all infinite pre-proofs in singleton logic with fixed points. They used the notion of transition systems to define their validity condition formally. Here we only provide a high-level description of the condition. It states that every cycle should be supported by the unfolding of a **least fixed point** on the **antecedent** or a **greatest fixed point** on the **succedent**. Since they allow mutual dependency of least and greatest fixed points, they need to consider the priority of each fixed point as well. Each cycle's supporting fixed point has to be of the highest priority among all fixed points that are unfolded in the cycle.

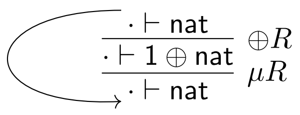
Fortier and Santocanale proved that the valid subset of derivations enjoys the cut-elimination property; in particular, a cut composing any two valid derivations can be eliminated effectively. They introduced a cut-elimination algorithm that applies internal cut reductions in a sub-algorithm $Treat$ until an external cut reduction is available. They proved that the algorithm is productive for infinite proofs satisfying their validity condition, which means that it outputs a cut-free step after every finite number of steps.

Their proof of cut elimination is based on a critical lemma which states termination of the internal reductions in the sub-algorithm $Treat$. The sub-algorithm creates a trace that is a complete lattice. The lemma is proved using the observation that a valid derivation tree has only a limited number of branches on its right or left side created by the cut rule. By this observation, they deduce that the sub-algorithm does not have an infinite computation tree.

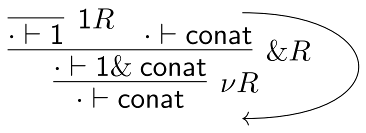
As a corollary to this lemma, the cut-elimination algorithm produces a potentially infinite cut-free proof for the annotated derivation. They further prove that the output of the cut-elimination algorithm is valid.

In Chapter 4 we use a variant of this technique to prove cut-elimination for a first-order intuitionistic linear logic with fixed points.

We conclude this section with two examples of circular derivations. The following circular pre-proof defined on the signature $\mathsf{nat} =^1_\mu 1 \oplus \mathsf{nat}$ depicts an infinite pre-proof that consists of repetitive application of μR followed by $\oplus R$:


$\cdot \vdash \mathsf{nat}$
$\oplus R$
$\cdot \vdash 1 \oplus \mathsf{nat}$
μR
$\cdot \vdash \mathsf{nat}$

This derivation is not valid. On the other hand, on the signature $\mathsf{conat} =^1_\nu 1 \,\&\, \mathsf{conat}$, we can define a circular pre-proof using greatest fixed points that is valid:


$1R$
$\cdot \vdash 1$
$\cdot \vdash \mathsf{conat}$
$\& R$
$\cdot \vdash 1 \& \mathsf{conat}$
νR
$\cdot \vdash \mathsf{conat}$

3.4 Classical multiplicative additive linear logic with fixed points ($\mu MALL^\infty$)

This section reviews the infinitary sequent calculus for multiplicative additive linear logic ($\mu MALL^\infty$) introduced by Baelde, Doumane, and Saurin and reviews some of their main results [7, 33].

$$\dfrac{}{\vdash A, A^{\perp}}\ \textsc{Id} \qquad \dfrac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, A^{\perp}}{\vdash \Gamma_1, \Gamma_2}\ \textsc{Cut}$$

$$\dfrac{\vdash \Gamma_1, A_1 \quad \vdash \Gamma_2, A_2}{\vdash \Gamma_1, \Gamma_2, A_1 \otimes A_2}\ \otimes R \qquad \dfrac{\vdash \Gamma, A_1, A_2}{\vdash \Gamma, A_1 \parr A_2}\ \parr R$$

$$\dfrac{\vdash \Gamma, A_k \quad k \in I}{\vdash \Gamma, \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R \qquad \dfrac{\vdash \Gamma, A_i \quad \forall i \in I}{\vdash \Gamma, \&\{l_i : A_i\}_{i \in I}}\ \& R$$

$$\dfrac{\vdash \Gamma, A[\mu x.A/x]}{\vdash \Gamma, \mu x.A}\ \mu R \qquad \dfrac{\vdash \Gamma, A[\nu x.A/x]}{\vdash \Gamma, \nu x.A}\ \nu R$$

FIGURE 3.2: Infinitary multiplicative additive linear logic with fixed points

The grammar for building formulas in $\mu MALL^{\infty}$ is as follows:

$$A \quad ::= \quad A \otimes A \mid A \parr A \mid \oplus\{\ell{:}A_\ell\}_{\ell \in L} \mid \&\{\ell{:}A_\ell\}_{\ell \in L} \mid \mu t.A \mid \nu t.A \mid t.$$

Fixed points are presented in the vectorial form, and their rules follow from the following classical negation involution:

$$(\mu t.A)^{\perp} = \nu t.A^{\perp} \qquad (\nu t.A)^{\perp} = \mu t.A^{\perp} \qquad t^{\perp} = t$$

They represent their proof system as the one-sided sequent calculus of Figure 3.2.

Like other infinitary calculi, they enforced a validity condition on infinite derivation to ensure cut-elimination. Their validity condition is similar to Fortier and Santocanale's validity condition: they both require an infinite branch to have infinitely many unfoldings of a proper fixed-point. However, there are a few differences in presenting these two validity conditions worth mentioning:

- Since multiple formulas are allowed to be in the succedents, we need to consider the thread formed from connecting each formula's derivatives in the infinitary branch separately. These threads are defined carefully by annotating each formula's occurrence by an address. In essence, a thread starting from a formula on a derivation is a list of its sub-occurrences. The sub-occurrences of a formula in the conclusion when a logical rule is applied on it are depicted in Figure 3.2 as formulas in the premises with the same color. When no logical rule is applied on a formula in the conclusion its sub-occurrence is the identical formula in a premise. (For the exact definition see [33].)

- Moreover, with a one-sided calculus, the proper fixed-point for supporting an infinite branch is always in succedents and thus is a greatest fixed-point.

- Finally, instead of using the notion of priority which only makes sense for a language presented using systems of equations, they use subformula ordering on the fixed-points.

In summary, a thread is valid if supported with a greatest fixed point which is minimal (wrt. subformula ordering) among those occurring infinitely often. A derivation is valid if every infinite branch has a valid thread.

Their cut-elimination algorithm is similar to Fortier and Santocanale's: to apply internal reductions and, when available, external reductions. However, the proof of the algorithm's productivity follows a different technique based on the semantical soundness of a truncation of the infinitary calculus. They form a contradiction with semantical soundness from assuming a non-productive sequence of reductions on a valid proof.

3.5 Other related work

3.5.1 Parity games and circular proofs.

Circular proofs in singleton logic are interpreted as the winning strategy in parity games [81]. A winning strategy corresponds to an asynchronous protocol for a deadlock-free communication of the players [57]. The cut-elimination result for circular proofs is a ground for reasoning about these communication protocols, and the related categorical concept of μ-bicomplete categories [82, 83].

3.5.2 Other approaches.

In the literature, bisimulation has been used effectively to prove the equality of structures defined as greatest fixed points. To prove properties other than equality for coinductive data types, one needs to use the somewhat less familiar coinduction principle [11, 26, 48, 70, 80]. Kozen and Silva established a practical proof principle to produce sound proofs by coinduction [59]. However, these separate principles are insufficient for data types mutually defined by induction and coinduction.

One recent approach in type theory integrates induction and coinduction by pattern and copattern matching and explicit well-founded induction on ordinals [2], following some earlier representations of induction and coinduction in type theory [1]. We will discuss this line of work further in Chapter 9.

Chapter 4

First order linear logic with least and greatest fixed points

In this chapter we introduce first order intuitionistic multiplicative additive linear logic with fixed points ($FIMALL^{\infty}_{\mu,\nu}$). In our first order calculus, we allow circularity in derivations generalizing the approach of Brotherston et al. [12, 13] by allowing both least and greatest fixed points. To ensure soundness of the proofs we impose a validity condition on our derivations. We introduce a cut elimination algorithm and prove that it produces a cut-free proof when applied on valid derivations. This algorithm is productive for valid derivations: it receives a potentially infinite valid proof as an input and outputs a cut-free infinite valid proof productively. (An algorithm is productive if every piece of its output is generated in a finite number of steps.) Our results, when restricted to the propositional singleton fragment are the same as Fortier and Santocanale's (Section 3.3).

We restrict the linear implication to allow only an atomic formula as its assumption, i.e. A_1 in $A_1 \multimap A_2$, is an atom. Our validity condition for the linear implication is also more restrictive than Baelde et al.'s treatment of its classical counterpart ($\parr$). Recall form Section 3.4 that in Baelde et al.'s notion of thread both A and B are considered to be sub-occurrences of $A \parr B$ in the $\parr$ rule. We, however, only consider B (and not A) as a continuation of the formula $A \multimap B$ in the rules for $\multimap$. We will comment more on this difference in Section 4.3.

The restrictions on the linear implication are motivated by two considerations. First, we can obtain a clearer cut elimination proof even in the presence of the multiplicative connectives; it allows us to adapt Fortier and Santocanale's cut elimination proof. Our proof is essentially different from Baelde et al.'s proof of cut elimination [7, 33] for infinitary multiplicative additive linear logic; in particular, we do not need to interpret the logical formulas in a classical truth semantics. Second, the resulting metalogic is strong enough for our primary application, i.e., encoding session-typed processes as formulas in linear logic. Furthermore, our restriction seems to be in line with the restrictions imposed on implication in multiset rewriting, e.g.

the work by Cervesato and Scedrov [17] in which they restrict an implication to be between tensors of atomic formulas.

McDowell and Miller [63] presented what can be considered a precursor of the metalogic we introduce in this chapter. They designed a metalogic with a similar goal: to prove properties about the specified programming systems in a formal framework. Their metalogic is defined based on (non-linear) intuitionistic sequent calculus that allows higher-order quantification over simply typed terms. Similar to our use of fixed points in $FIMALL^{\infty}_{\mu,\nu}$, they use definitional reflection [84] via left- and right- rules for unfolding definitions in the sequent calculus. Their logic also admits inductive reasoning using an explicit rule for induction on natural numbers. However, it does not support coinduction and is not based on an infinitary logic.

4.1 Language and calculus

The syntax and calculus of $FIMALL^{\infty}_{\mu,\nu}$ is similar to the first order linear logic (Section 2.2.3), but it is extended to handle predicates that are defined as mutual least and greatest fixed points. The syntax of formulas follows the grammar

$$A \quad ::= \quad 1 \mid A \otimes A \mid A \multimap A \mid \oplus\{l_i : A_i\}_{i\in I} \mid \&\{l_i : A_i\}_{i\in I} \mid \exists x.\, A(x) \mid \forall x.\, A(x) \mid s = t \mid T(\bar{t})$$

where s, t stand for terms and x, y for term variables. We do not specify a grammar for terms; all terms are of the only type U with binders. Similar to Section 2.2.3 we restrict our terms to Miller's [67] higher-order patterns which is an extension of first-order terms that include bound variable names and scopes.

$T(\bar{t})$ is an instance of a predicate. A predicate can be defined using least and greatest fixed points in a *signature* Σ

$$\Sigma ::= \cdot \mid \Sigma, T(\overline{x}) =^i_\mu A \mid \Sigma, T(\overline{x}) =^i_\nu A.$$

An atom is an instance of a predicate $T(\bar{t})$ that is *not defined in the signature* as a least or greatest fixed point. We restrict the formulas in our metalogic to those in which the left-hand side of a linear implication is an atom, i.e. formula A in $A \multimap B$ is atomic.

The subscript a of a fixed point $T(\overline{x}) =^i_a A$ determines whether it is a least or greatest fixed point. If $a = \mu$, then predicate $T(\overline{x})$ is a least fixed point and inductively defined (e.g., the property of being a natural number) and if $a = \nu$ it is a greatest fixed point and coinductively defined (e.g., the lexicographic order on streams).

The superscript $i \in \mathbb{N}$ is the relative priority of $T(\overline{x})$ in the signature Σ with the condition that if $T_1(\overline{x}) =^i_a A, T_2(\overline{x}) =^i_b B \in \Sigma$, then $a = b$. We say $T_1(\overline{x}) =^i_a A$ has higher priority than $T_2(\overline{x}) =^j_b B$ if $i < j$. The priorities determine the order by which the fixed-point equations in Σ are solved [81], and we use them to define the validity condition on infinite derivations.

Example 4.1. *Let signature Σ_1 be*

$$\begin{array}{lll} \mathtt{Stream}(x) & =^1_\nu & (\exists y.\exists z.(x = y \cdot z) \otimes \mathtt{Nat}(y) \otimes \mathtt{Stream}\,(z)) \\ \mathtt{Nat}(x) & =^2_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Nat}(y)) \oplus (x = \mathsf{z}) \end{array}$$

where predicate $\mathtt{Nat}$ *refers to the property of being a natural number, and predicate* $\mathtt{Stream}$ *refers to the property of being a stream of natural numbers. A stream of natural numbers is defined as a concatenation of natural numbers* $y.z$ *where* y *is a natural number and* z *is the rest of the stream.* z *corresponds to* $0 \in \mathtt{N}$ *and* $\mathsf{s}y$ *corresponds to the successor of* y.

We define $\mathtt{Stream}$ *to have a higher priority relative to* $\mathtt{Nat}$ *since the definition of a natural number is nested in the definition of a stream.*

Example 4.2. *Let signature Σ_2 be*

$$\begin{array}{lll} \mathtt{Nat}(x) & =^1_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Nat}(y)) \oplus ((x = \mathsf{z})) \\ \mathtt{Even}(x) & =^2_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Odd}(y)) \oplus ((x = \mathsf{z})) \\ \mathtt{Odd}(x) & =^2_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Even}(y)) \end{array}$$

where positive predicates $\mathtt{Nat}$, $\mathtt{Even}$, *and* $\mathtt{Odd}$ *refer to the properties of being natural, even, and odd numbers respectively. All predicates in this signature are inductive, and their relative priority is not important. We assign a higher priority to* $\mathtt{Nat}$ *so that we can use this signature for illustrating some notations in the future examples.*

Derivations in $\mathit{FIMALL}^\infty_{\mu,\nu}$ establish judgments of the form $\Gamma \vdash_\Sigma A$ where Γ is a multiset of formulas and Σ is the signature. We omit Σ from the judgments, since it never changes throughout a proof. The infinitary sequent calculus for this logic is given in Figure 4.1.

Example 4.3. *Consider signature* Σ_2 *and predicates* $\mathtt{Even}$ *and* $\mathtt{Odd}$ *defined in Example 4.2. The following derivation is a finite proof of one* (sz) *being an odd number.*

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash \mathsf{sz} = \mathsf{sz}} = R \qquad \dfrac{\dfrac{\dfrac{}{\cdot \vdash (\mathsf{z} = \mathsf{z})} = R}{\cdot \vdash (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Odd}(y)) \oplus (\mathsf{z} = \mathsf{z})} \oplus R}{\cdot \vdash \mathtt{Even}(\mathsf{z})} \mu R}{\dfrac{\cdot \vdash (\mathsf{sz} = \mathsf{sz}) \otimes \mathtt{Even}(\mathsf{z})}{\cdot \vdash \exists y.(\mathsf{sz} = \mathsf{s}y) \otimes \mathtt{Even}(y)} \exists R} \otimes R}{\cdot \vdash \mathtt{Odd}(\mathsf{sz})} \mu R$$

The calculus of Figure 4.1 is infinitary, meaning that we allow finitely branching but non-wellfounded derivations. Like other infinitary calculi described in this thesis, derivations in $\mathit{FIMALL}^\infty_{\mu,\nu}$ do not necessarily enjoy the cut-elimination property and are called *pre-proofs.*

We call the open leaves in a partial derivation *open subgoals* of the derivation. The judgments in the derivation that are the conclusion of a rule are called *interior judgments.* A *circular derivation* is the finite representation of an infinite one in which we can identify each open

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{\Gamma \vdash A \quad \Gamma', A \vdash C}{\Gamma, \Gamma' \vdash C}\ \textsc{Cut}$$

$$\frac{}{\cdot \vdash 1}\ 1R \qquad \frac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\ 1L$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma' \vdash A_2}{\Gamma, \Gamma' \vdash A_1 \otimes A_2}\ \otimes R \qquad \frac{\Gamma, A_1, A_2 \vdash B}{\Gamma, A_1 \otimes A_2 \vdash B}\ \otimes L$$

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \multimap A_2}\ \multimap R \qquad \frac{\Gamma \vdash A_1 \quad \Gamma', A_2 \vdash B}{\Gamma, \Gamma', A_1 \multimap A_2 \vdash B}\ \multimap L$$

$$\frac{\Gamma \vdash A_k \quad k \in I}{\Gamma \vdash \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R \qquad \frac{\Gamma, A_i \vdash B \quad \forall i \in I}{\Gamma, \oplus\{l_i : A_i\}_{i \in I} \vdash B}\ \oplus L$$

$$\frac{\Gamma \vdash A_i \quad \forall i \in I}{\Gamma \vdash \&\{l_i : A_i\}_{i \in I}}\ \& R \qquad \frac{\Gamma, A_k \vdash B \quad k \in I}{\Gamma, \&\{l_i : A_i\}_{i \in I} \vdash B}\ \& L$$

$$\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x.P(x)}\ \exists R \qquad \frac{\Gamma, P(x) \vdash B \quad x \text{ fresh}}{\Gamma, \exists x.P(x) \vdash B}\ \exists L_x$$

$$\frac{\Gamma \vdash P(x) \quad x \text{ fresh}}{\Gamma \vdash \forall x.P(x)}\ \forall R_x \qquad \frac{\Gamma, P(t) \vdash B}{\Gamma, \forall x.P(x) \vdash B}\ \forall L$$

$$\frac{T(\overline{x}) =_\mu A \quad \Gamma \vdash [\overline{t}/\overline{x}]A}{\Gamma \vdash T(\overline{t})}\ \mu_T R \qquad \frac{T(\overline{x}) =_\mu A \quad \Gamma, [\overline{t}/\overline{x}]A \vdash B}{\Gamma, T(\overline{t}) \vdash B}\ \mu_T L$$

$$\frac{T(\overline{x}) =_\nu A \quad \Gamma \vdash [\overline{t}/\overline{x}]A}{\Gamma \vdash T(\overline{t})}\ \nu_T R \qquad \frac{T(\overline{x}) =_\nu A \quad \Gamma, [\overline{t}/\overline{x}]A \vdash B}{\Gamma, T(\overline{t}) \vdash B}\ \nu_T L$$

$$\frac{}{\cdot \vdash s = s}\ {=}R \qquad \frac{\forall \theta \in \mathtt{mgu}(t, s) \quad \Gamma[\theta] \vdash B[\theta]}{\Gamma, s = t \vdash B}\ {=}L$$

FIGURE 4.1: Infinitary calculus for first order linear logic with fixed points. (In the =L rule, the set $\mathtt{mgu}(t, s)$ is either empty, or a singleton set containing a most general unifier.)

subgoal with an identical interior judgment[1]. In the first order context we may need to use a substitution rule right before a circular edge to make the subgoal and interior judgment exactly identical [12]:

$$\frac{\Gamma \vdash B}{\Gamma[\theta] \vdash B[\theta]}\ \mathtt{subst}_\theta$$

We can transform a circular derivation to its underlying infinite derivation in a productive way, i.e. at each step we can produce one rule of the infinite derivation. Consider a $\mathtt{subst}_\theta$ rule and a circular edge in the circular derivation. We (1) instantiate the (possibly circular) derivation to which the circular edge pointed with substitution θ, (2) replace the $\mathtt{subst}_\theta$ rule with the instantiated derivation, and (3) remove the circular edge. Lemma 4.1 proves that instantiation of a derivation used in step (1) exists and does not change the structure of derivation.

[1]By this definition vacuous circular derivation that identifies an open sub-goal with itself is not allowed.

Lemma 4.1 (Substitution). *For a derivation*

$$\begin{array}{c}\Pi\\ \Gamma \vdash A\end{array}$$

in the infinite system and substitution θ, there is a derivation for

$$\begin{array}{c}\Pi[\theta]\\ \Gamma[\theta] \vdash A[\theta]\end{array}$$

where $\Pi[\theta]$ is the whole derivation Π or a prefix of it instantiated by θ.

Proof. The proof is by coinduction on the structure of

$$\begin{array}{c}\Pi\\ \Gamma \vdash A\end{array}$$

The only interesting case is where we get to the $= L$ rule.

$$\dfrac{\begin{array}{c}\Pi'\\ \Gamma[\theta'] \vdash B[\theta']\end{array} \quad \forall \theta' \in \mathsf{mgu}(s, t)}{\Gamma, s = t \vdash B} = L$$

If the set $\mathsf{mgu}(t[\theta], s[\theta])$ is empty then so is $\Pi'[\theta]$. Otherwise if η is the single element of $\mathsf{mgu}(t[\theta], s[\theta])$, then for some substitution λ we have $\theta\eta = \theta'\lambda$, and we can form the rest of derivation for substitution λ as $\Pi'[\lambda]$ coinductively. □

The definitions and proofs in this chapter are based on the infinite system of Figure 4.1. When possible, we present the derivations in a circular form.

Example 4.4. *Consider signature Σ_2 and predicates* Nat*,* Even*, and* Odd *defined in Example 4.2. Figure 4.2 represents a circular derivation for* $\mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}\, x)$*. Π is the finite derivation given in Example 4.3.*

In Figure 4.2, the judgment $\star\ \mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}x)$ is an open subgoal that can be identified with the interior judgment $\star\ \mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}x)$. We marked both judgments with the same symbol $\star$ to represent this identification.

We can interpret the proof in Example 4.4 as an inductive proof where its circular edge corresponds to applying the induction hypothesis. In the next two examples we represent two coinductive proofs in our circular calculus. Both examples are adapted from [59].

Example 4.5. *Define Σ_3 to consist of a single predicate with negative polarity $(x \sim y) =^1_\nu (\mathsf{hd}\, x = \mathsf{hd}\, y) \,\&(\mathsf{tl}\, x \sim \mathsf{tl}\, y)$.*
Predicate $(x \sim y)$ can be read as a bisimulation between streams x and y, where the term $\mathsf{hd}\, x$ refers to the head of the stream x and $\mathsf{tl}\, x$ refers to its tail. For simplicity, we use the hd *and* tl *notation as an alternative to the concatenation in Example 4.1. We present a circular derivation for $\sim$ being symmetric in Figure 4.3.*

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\cdot \vdash (\mathsf{ss}y = \mathsf{ss}y)}\;{=}R
        \qquad
        \dfrac{
          \dfrac{
            \dfrac{
              \dfrac{
                \dfrac{
                  \dfrac{
                    \dfrac{
                      \dfrac{
                        \dfrac{}{\cdot \vdash \mathsf{ss}z = \mathsf{ss}z}\;{=}R
                        \quad
                        \dfrac{\star\ \mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}x)}{\mathtt{Even}(z) \vdash \mathtt{Odd}(\mathsf{s}z)}\;\mathtt{Subst}_{[z/x]}
                      }{\mathtt{Even}(z) \vdash (\mathsf{ss}z = \mathsf{ss}z) \otimes \mathtt{Odd}(\mathsf{s}z)}\;\otimes R
                    }{\mathtt{Even}(z) \vdash (\exists y.(\mathsf{ss}z = \mathsf{s}y) \otimes \mathtt{Odd}(y))}\;\exists R
                  }{\mathtt{Even}(z) \vdash (\exists y.(\mathsf{ss}z = \mathsf{s}y) \otimes \mathtt{Odd}(y)) \oplus (\mathsf{ss}z = \mathsf{z})}\;\oplus R
                }{\mathtt{Even}(z) \vdash \mathtt{Even}(\mathsf{ss}z)}\;\mu R
              }{(y = \mathsf{s}z), \mathtt{Even}(z) \vdash \mathtt{Even}(\mathsf{s}y)}\;{=}L
            }{(y = \mathsf{s}z) \otimes \mathtt{Even}(z) \vdash \mathtt{Even}(\mathsf{s}y)}\;\otimes L
          }{\exists z.(y = \mathsf{s}z) \otimes \mathtt{Even}(z) \vdash \mathtt{Even}(\mathsf{s}y)}\;\exists L
        }{\mathtt{Odd}(y) \vdash \mathtt{Even}(\mathsf{s}y)}\;\mu L
      }{\mathtt{Odd}(y) \vdash (\mathsf{ss}y = \mathsf{ss}y) \otimes \mathtt{Even}(\mathsf{s}y)}\;\otimes R
    }{\mathtt{Odd}(y) \vdash \exists z.(\mathsf{ss}y = \mathsf{s}z) \otimes \mathtt{Even}(z)}\;\exists R
  }{\dfrac{\dfrac{\dfrac{\mathtt{Odd}(y) \vdash \mathtt{Odd}(\mathsf{ss}y)}{(x = \mathsf{s}y), \mathtt{Odd}(y) \vdash \mathtt{Odd}(\mathsf{s}x)}\;{=}L}{(x = \mathsf{s}y) \otimes \mathtt{Odd}(y) \vdash \mathtt{Odd}(\mathsf{s}x)}\;\otimes L}{\exists y.(x = \mathsf{s}y) \otimes \mathtt{Odd}(y) \vdash \mathtt{Odd}(\mathsf{s}x)}\;\exists L}\;\mu R
  \qquad
  \dfrac{\begin{matrix}\Pi \\ \cdot \vdash \mathtt{Odd}(\mathsf{sz})\end{matrix}}{(x = \mathsf{z}) \vdash \mathtt{Odd}(\mathsf{s}x)}\;{=}L
}{
  \dfrac{(\exists y.(x = \mathsf{s}y) \otimes \mathtt{Odd}(y)) \oplus (x = \mathsf{z}) \vdash \mathtt{Odd}(\mathsf{s}x)}{\star\ \mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}x)}\;\mu L
}\;\oplus L
$$

FIGURE 4.2: Successor of every even number is odd.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\cdot \vdash (\mathsf{hd}\,x = \mathsf{hd}\,x)}\;{=}R
      }{(\mathsf{hd}\,x = \mathsf{hd}\,y) \vdash (\mathsf{hd}\,y = \mathsf{hd}\,x)}\;{=}L
    }{(\mathsf{hd}\,x = \mathsf{hd}\,y)\,\&(\mathsf{tl}\,x \sim \mathsf{tl}\,y) \vdash (\mathsf{hd}\,y = \mathsf{hd}\,x)}\;\&L
    \qquad
    \dfrac{
      \dfrac{\star\ (x \sim y) \vdash (y \sim x)}{(\mathsf{tl}\,x \sim \mathsf{tl}\,y) \vdash (\mathsf{tl}\,y \sim \mathsf{tl}\,x)}\;\mathsf{Subst}_{[\mathsf{tl}x/x, \mathsf{tl}y/y]}
    }{(\mathsf{hd}\,x = \mathsf{hd}\,y)\,\&(\mathsf{tl}\,x \sim \mathsf{tl}\,y) \vdash (\mathsf{tl}\,y \sim \mathsf{tl}\,x)}\;\&L
  }{
    \dfrac{(\mathsf{hd}\,x = \mathsf{hd}\,y)\,\&(\mathsf{tl}\,x \sim \mathsf{tl}\,y) \vdash (\mathsf{hd}\,y = \mathsf{hd}\,x)\,\&(\mathsf{tl}\,y \sim \mathsf{tl}\,x)}{(x \sim y) \vdash (\mathsf{hd}\,y = \mathsf{hd}\,x)\,\&(\mathsf{tl}\,y \sim \mathsf{tl}\,x)}\;\nu L
  }\;\&R
}{\star\ (x \sim y) \vdash (y \sim x)}\;\nu R
$$

FIGURE 4.3: Relation $\sim$ defined on streams is symmetric.

Example 4.6. *We can reason about the properties of stream operations in our calculus as well. Consider three operations* merge, split_1 *and* split_2*. Operation* merge *receives two streams and merge them into a single stream by alternatively outputting an element of each. Operations* split_1 *and* split_2 *receive a stream x as an input and return the odd and even elements of it, respectively. We define these operations as negative predicates in our language. Define signature Σ_4 as*

$$
\begin{aligned}
\mathtt{Merge}(x, y, z) \quad &=^1_\nu \quad (\mathsf{hd}\,z = \mathsf{hd}\,x \otimes \mathtt{Merge}\,(y, \mathsf{tl}\,x, \mathsf{tl}\,z)) \\
\mathtt{Split}_1(x, y) \quad &=^1_\nu \quad (\mathsf{hd}\,y = \mathsf{hd}\,x \otimes \mathtt{Split}_2(\mathsf{tl}\,x, \mathsf{tl}\,y)) \\
\mathtt{Split}_2(x, y) \quad &=^1_\nu \quad (\mathtt{Split}_1(\mathsf{tl}\,x, y))
\end{aligned}
$$

The derivation given in Figure 4.4 shows that operations merge *and* split_i *are inverses: Split a stream x into two streams y_1 and y_2 using* split_1 *and* split_2*, respectively, then merge y_1 and y_2. The result is x.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash \mathsf{hd}\, y_1 = \mathsf{hd}\, y_1}=R}{\mathsf{hd}\, y_1 = \mathsf{hd}\, x \vdash \mathsf{hd}\, x = \mathsf{hd}\, y_1}=L \qquad \dfrac{\star\ \mathtt{S}_2(x, y_2), \mathtt{S}_1(x, y_1) \vdash \mathtt{M}(y_1, y_2, x)}{\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), \mathtt{S}_1(\mathsf{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}\mathsf{Sub}_{[\mathsf{tl}x, \mathsf{tl}y_1, y_2/x, y_2, y_1]}}{\mathsf{hd}\, y_1 = \mathsf{hd}\, x\, , \mathtt{S}_2(\mathsf{tl}\, x, \mathsf{tl}\, y_1), \mathtt{S}_1(\mathsf{tl}\, x, y_2) \vdash \mathsf{hd}\, x = \mathsf{hd}\, y_1 \otimes \mathtt{M}(y_2, \mathsf{tl}\, y_1, \mathsf{tl}\, x)}\otimes R}{\mathsf{hd}\, y_1 = \mathsf{hd}\, x \otimes \mathtt{S}_2(\mathsf{tl}\, x, \mathsf{tl}\, y_1), \mathtt{S}_1(\mathsf{tl}\, x, y_2) \vdash \mathsf{hd}\, x = \mathsf{hd}\, y_1 \otimes \mathtt{M}(y_2, \mathsf{tl}\, y_1, \mathsf{tl}\, x)}\otimes L}{\mathsf{hd}\, y_1 = \mathsf{hd}\, x \otimes \mathtt{S}_2(\mathsf{tl}\, x, \mathsf{tl}\, y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\, x = \mathsf{hd}\, y_1 \otimes \mathtt{M}(y_2, \mathsf{tl}\, y_1, \mathsf{tl}\, x)}\nu L}{\mathtt{S}_1(x, y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\, x = \mathsf{hd}\, y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\, y_1, \mathsf{tl}\, x)}\nu L}{\star\ \mathtt{S}_1(x, y_1), \mathtt{S}_2(x, y_2) \vdash \mathtt{M}(y_1, y_2, x)}\nu R$$

FIGURE 4.4: Operations Merge(M) and Split$_\mathtt{i}$(S$_\mathtt{i}$) are inverses.

4.2 Pattern Matching

It may not be feasible to present a large piece of derivation fully in the calculus of Figure 4.1. For the sake of brevity, we may represent predicates of positive polarity in the signature using pattern matching and build equivalent derivations based on that signature [12, 79]. In all the examples in this thesis, if we use pattern matching, it should be clear how to transform the signature and derivations into the main logical system.

Example 4.7. *Recall signature Σ_2 in Example 4.2*

$$\begin{array}{lll}
\mathtt{Nat}(x) & =^1_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Nat}(y)) \oplus ((x = \mathsf{z})) \\
\mathtt{Even}(x) & =^2_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Odd}(y)) \oplus ((x = \mathsf{z})) \\
\mathtt{Odd}(x) & =^2_\mu & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Even}(y))
\end{array}$$

where positive predicates Nat, Even, *and* Odd *refer to the properties of being natural, even, and odd numbers respectively.*

Redefine predicates Even, Odd, *and* Nat *by pattern matching in signature Σ_2' as:*

$$\begin{array}{lllllll}
\mathtt{Nat}(\mathsf{z}) & =^1_\mu & 1 & \qquad & \mathtt{Nat}(\mathsf{s}y) & =^1_\mu & \mathtt{Nat}(y) \\
\mathtt{Odd}(\mathsf{z}) & =^1_\mu & 0 & & \mathtt{Odd}(\mathsf{s}y) & =^1_\mu & \mathtt{Even}(y) \\
\mathtt{Even}(\mathsf{z}) & =^1_\mu & 1 & & \mathtt{Even}(\mathsf{s}y) & =^1_\mu & \mathtt{Odd}(y)
\end{array}$$

The circular derivation in Example 4.4 can be simplified in the following way:

$$[1]\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}1R}{\cdot \vdash \mathtt{Even}(\mathsf{z})}\mu R}{\cdot \vdash \mathtt{Odd}(\mathsf{s}\,\mathsf{z})}\mu R}{1 \vdash \mathtt{Odd}(\mathsf{s}\,\mathsf{z})}1L}{\dagger\,\mathtt{Even}(\mathsf{z}) \vdash \mathtt{Odd}(\mathsf{s}\,\mathsf{z})}\mu L \qquad\qquad [2]\quad \dfrac{\dfrac{\star\,\mathtt{Odd}(x) \vdash \mathtt{Even}(\mathsf{s}\,x)}{\mathtt{Odd}(x) \vdash \mathtt{Odd}(\mathsf{s}\,\mathsf{s}\,x)}\mu R}{\dagger\,\mathtt{Even}(\mathsf{s}\,x) \vdash \mathtt{Odd}(\mathsf{s}\,\mathsf{s}\,x)}\mu L$$

$$[3]\quad \dfrac{\dfrac{}{0 \vdash \mathtt{Odd}(\mathsf{z})}0L}{\star\,\mathtt{Odd}(\mathsf{z}) \vdash \mathtt{Even}(\mathsf{s}\,z)}\mu L \qquad\qquad [4]\quad \dfrac{\dfrac{\dagger\,\mathtt{Even}(x) \vdash \mathtt{Odd}(\mathsf{s}\,x)}{\mathtt{Even}(x) \vdash \mathtt{Even}(\mathsf{s}\,\mathsf{s}\,x)}\mu R}{\star\,\mathtt{Odd}(\mathsf{s}\,x) \vdash \mathtt{Even}(\mathsf{s}\,\mathsf{s}\,x)}\mu L$$

By the definition of signature Σ_2', the pattern of x in $\texttt{Odd}(x)$ is either of the form $\texttt{s}\,y$ or $\texttt{z}$. At the subgoal marked with $\star$ in subderivation 2, we form a branch similar to the $\oplus L$ rule to cover all possible patterns of x; we continue with subderivations 3 and 4. With the same reasoning at the subgoal marked with $\dagger$ in the subderivation 4 we form a branch with subderivations 1 and 2.

A major contribution of this chapter is to give a criterion for validity of theorems proved by simultaneous induction and coinduction. In the next example we see an interplay between positive and negative fixed points in the derivation. This example is adapted from [2]. Define predicate $\texttt{run}(x, t)$ to represent computation of a stream processor, where x is the list of operations we want to compute.

Example 4.8. *Define the signature Σ_5 to be*

$$\begin{array}{lll}
\texttt{run}(\mathit{end}\,, t) & =^1_\mu & 1 \\
\texttt{run}(\mathit{seq}(\mathit{skip}, x), t) & =^1_\mu & \texttt{run}(x, t) \\
\texttt{run}(\mathit{seq}(\mathit{put}(x), y), t) & =^1_\mu & \texttt{nrun}\,(x, y, t) \\
\texttt{nrun}(x, y, t) & =^2_\nu & \texttt{hd}\,t = \texttt{z} \,\&\, \texttt{run}(\mathit{seq}(x, y), \texttt{tl}\,t)
\end{array}$$

Operations can be either a skip or a $\mathit{put}(x)$. They are composed to each other with $\mathit{seq}(x, y)$. Operation skip simply skips one step and does not contribute to the output stream t. Operation $\mathit{put}(x)$ puts element $\texttt{z}$ as the head of the output stream t and appends a new list of operations x to the original list of operations. After computing skip the length of remaining operations in x goes down by one. So we can define $\texttt{run}(\mathit{seq}(\mathit{skip}, x), t)$ inductively. $\mathit{put}(x)$ increases the length of the operations, but produces an element of the output stream. So $\texttt{run}(\mathit{seq}(\mathit{put}(x), y), t)$ needs to be defined coinductively. We assigned a higher priority to the inductive predicates since the overall program is terminating given that the length of the list of operators x is finite.

The equivalent signature without pattern matching is

$$\begin{array}{lll}
\texttt{run}(x, t) & =^1_\mu & \oplus\{\mathsf{end} : x = \mathit{end} \otimes 1, \\
& & \quad \mathsf{skip} : \exists x'.\mathit{seq}(\mathit{skip}, x') \otimes \texttt{run}(x', t), \\
& & \quad \mathsf{put} : \exists x'.\exists y.x = \mathit{seq}(\mathit{put}(x'), y) \otimes \texttt{nrun}(x', y, t)\} \\
\texttt{nrun}(x, y, t) & =^2_\nu & \texttt{hd}\,t = \texttt{z} \,\&\, \texttt{run}(\mathit{seq}(x, y), \texttt{tl}\,t)
\end{array}$$

Here we define $\texttt{run}(\mathit{seq}(\mathit{put}(x), y), t)$ in two steps to follow the rules of definition by pattern matching: the pattern is broken down inductively and is defined as a positive fixed point. In the case where the pattern matches $\mathit{seq}(\mathit{put}(x), y)$, the predicate $\texttt{run}$ is defined using an intermediate negative predicate $\texttt{nrun}$ coinductively. We may abbreviate this definition to one step as:

$$\texttt{run}(\mathit{seq}(\mathit{put}(x), y), t) =^2_\nu \texttt{hd}\,t = \texttt{z} \,\&\, \texttt{run}(\mathit{seq}(x, y), \texttt{tl}\,t)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\,1R}{\cdot \vdash 1 \oplus \texttt{ztream}(t)}\,\oplus R}{\cdot \vdash \texttt{zlist}(t)}\,\mu R}{1 \vdash \texttt{zlist}(t)}\,1L}{\dagger\,\texttt{run}(end\,, t) \vdash \texttt{zlist}(t)}\,\mu L \qquad \dfrac{\dagger\,\texttt{run}(x,t) \vdash \texttt{zlist}(t)}{\dagger\,\texttt{run}(seq(skip, x), t) \vdash \texttt{zlist}(t)}\,\mu L$$

$$\dfrac{\dfrac{\dfrac{\star\,\texttt{nrun}\,(x,y,t) \vdash \texttt{ztream}(t)}{\texttt{nrun}\,(x,y,t) \vdash 1 \oplus \texttt{ztream}(t)}\,\oplus R}{\texttt{nrun}\,(x,y,t) \vdash \texttt{zlist}(t)}\,\mu R}{\dagger\,\texttt{run}(seq(put(x), y), t) \vdash \texttt{zlist}(t)}\,\mu L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{hd}\,t = \mathsf{z} \vdash \mathsf{hd}\,t = \mathsf{z}}\,\texttt{ID}}{\mathsf{hd}\,t = \mathsf{z} \mathbin{\&} \texttt{run}(seq(x,y), \texttt{tl}\,t) \vdash \mathsf{hd}\,t = \mathsf{z}}\,\& L}{\texttt{nrun}\,(x,y,t) \vdash \mathsf{hd}\,t = \mathsf{z}}\,\nu L \quad \dfrac{\dfrac{\overset{\dagger}{\texttt{run}(x; y, \texttt{tl}\,t) \vdash \texttt{zlist}(\mathsf{tl}\,t)}}{\mathsf{hd}\,t = \mathsf{z} \mathbin{\&} \texttt{run}(seq(x,y), \texttt{tl}\,t) \vdash \texttt{zlist}(\mathsf{tl}\,t)}\,\& L}{\texttt{nrun}\,(x,y,t) \vdash \texttt{zlist}(\mathsf{tl}\,t)}\,\nu L}{\texttt{nrun}\,(x,y,t) \vdash \mathsf{hd}\,t = \mathsf{z} \mathbin{\&} \texttt{zlist}(\mathsf{tl}\,t)}\,\& R}{\star\,\texttt{nrun}\,(x,y,t) \vdash \texttt{ztream}(t)}\,\nu R$$

FIGURE 4.5: run produces a possibly infinite list of elements z

In Figure 4.5, we prove that a run of any list of operations x produces a (possibly infinite) list of elements z*:*

$$\begin{array}{lll} \texttt{zlist}(t) & =^1_\mu & 1 \oplus \texttt{ztream}(t) \\ \texttt{ztream}(t) & =^2_\nu & \mathsf{hd}\,t = \mathsf{z} \mathbin{\&} \texttt{zlist}\,(\texttt{tl}\,t) \end{array}$$

We give circular derivations for both $\dagger\,\texttt{run}(x,t) \vdash \texttt{zlist}(t)$ *and* $\star\,\texttt{nrun}(x,y,t) \vdash \texttt{ztream}(t)$ *to show the interplay between coinductive and inductive predicates.*

4.3 A validity condition on first order derivations

In Section 4.1, we introduced an infinitary calculus for first order linear logic with fixed points. This section establishes a concept of validity with respect to cut elimination. As usual, cut elimination for valid derivations ensures consistency: it implies that there is no proof for $\cdot \vdash 0$ in our calculus. All derivations presented in this chapter are valid by the definition we present in this section. We leave it to the reader to check their validity.

To establish a validity condition we need to keep track of the behavior of any particular formula throughout the whole derivation. We uniquely annotate the formulas in a judgment with variables $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The variables used for labeling formulas are disjoint from the set of term variables. Similar to proof terms, the annotations can differentiate between two alternative proofs of a judgment. For example, by annotating the formulas, the proof

$$\dfrac{\dfrac{}{A \vdash A}\,\textsc{Id} \quad \dfrac{}{A \vdash A}\,\textsc{Id}}{A, A \vdash A \otimes A}\,\otimes R$$

can be differentiated into the following two proofs based on the roles of the distinct but equivalent formulas in the antecedent of the conclusion:

$$\dfrac{\dfrac{}{\mathbf{x}:A \vdash \mathbf{w}:A}\textsc{Id} \quad \dfrac{}{\mathbf{y}:A \vdash \mathbf{z}:A}\textsc{Id}}{\mathbf{x}:A, \mathbf{y}:A \vdash \mathbf{z}:A \otimes A}\otimes R \qquad\qquad \dfrac{\dfrac{}{\mathbf{y}:A \vdash \mathbf{w}:A}\textsc{Id} \quad \dfrac{}{\mathbf{x}:A \vdash \mathbf{z}:A}\textsc{Id}}{\mathbf{x}:A, \mathbf{y}:A \vdash \mathbf{z}:A \otimes A}\otimes R$$

where $\mathbf{w}$ is a fresh variable that we introduce as the (left) continuation of $\mathbf{z}$, when a $\otimes R$ rule applies on it. The freshness of variable $\mathbf{w}$ ensures the uniqueness invariant of each formula in the sequent. Similarly, in the cut rule we annotate the cut formula with a fresh variable $\mathbf{w}$ to distinguish it from the other formulas in the derivation:

$$\dfrac{\dfrac{}{\mathbf{x}:A \vdash \mathbf{w}:A}\textsc{Id} \quad \dfrac{}{\mathbf{w}:A \vdash \mathbf{y}:A}\textsc{Id}}{\mathbf{x}:A \vdash \mathbf{y}:A}\textsc{Cut}$$

We track the generation of variables to capture evolution of a formula in a derivation. A generation α is defined over natural numbers. We call a variable used for labelling formulas in a sequent and its generation ($\mathbf{x}^\alpha$) a generational variable. Different generations of the same variable allow us to capture the progress of a given formula which is made when an unfolding fixed point rule is applied on it. With this annotation we can keep track of the behaviour of any particular formula throughout the whole derivation. Our validity condition requires that at least one formula in every infinite branch behaves in a way that justifies validity of that branch.

Figure 4.6 shows the calculus annotated with variable generations and their relations. A basic judgment in the annotated calculus is of the form $\Delta \vdash_\Omega \mathbf{z}^\beta{:}C$ where Δ is a multiset of formulas annotated with (unique) generational variables, i.e. its elements are of the form $\mathbf{x}^\alpha{:}A$. The set Ω keeps the relation between different generations of variables in a derivation for each priority. The relations in Ω are of the form $\mathbf{x}_i^\alpha < \mathbf{y}_i^\beta$ or $\mathbf{x}_i^\alpha = \mathbf{y}_i^\beta$, where i is a priority of a predicate in the signature.

We picked generational variables to track formulas over alternatives since they resemble channels in session-typed processes [27]. We will use this analogy in the proof of strong progress property (Section 7.4). Generational variables are related to other alternatives for annotating a formula in an infinite derivation to track its behaviour [7, 92]. For example, Baelde et al.'s notion of (pre)formula occurrences is similar to annotated formulas with generational variables. However, some subtle differences make them distinct: 1) Generational variables are unique in each sequent, but they are not necessarily assigned uniquely to sub-formulas, e.g. $\&R/L$ rules. 2) The relation that we form between generational variables is not based on a subformula relation, e.g. μR, νL, and $\multimap R/L$ rules.

The relation of a new generation $\mathbf{y}^{\alpha+1}$ to its prior $\mathbf{y}^\alpha$ is determined by the role of the rule that introduces it in (co)induction. The μL rule breaks down an inductive antecedent and νR produces a coinductive succedent. They both take a step toward termination/productivity of the proof: we put the new generation of the variable they introduce to be less than the prior ones in the given priority. Their counterpart rules νL and μR, however, do not contribute to termination/productivity. They break the relation between the new generation and its prior ones for the given priority. In the μL rule, for example, we add the relation $\mathbf{y}_i^{\alpha+1} < \mathbf{y}_i^{\alpha}$ to Ω'. It is interpreted as the new generation $\mathbf{y}^{\alpha+1}$ is less than its prior generation on priority i. For the other priorities $j \neq i$ we keep $\mathbf{y}_j^{\alpha+1} = \mathbf{y}_j^{\alpha}$.

As explained above, in the Cut rule we introduce a fresh variable annotated with a generation: $\mathbf{w}^{\eta}$ where $\mathbf{w}$ is a fresh variable and η is a generational variable. Since it refers to a new formula, we designate it to be incomparable to other variables. We consider $\mathbf{w}^{\eta}$ as a continuation of $\mathbf{z}^{\beta}$ in the rule $\otimes R$ and add $\mathbf{w}_i^{\eta} = \mathbf{z}_i^{\beta}$ to Ω for each priority i. Similarly, we keep the relation of $\mathbf{y}^{\alpha}$ with its continuation $\mathbf{w}^{\eta}$ for each priority in Ω for the $\otimes L$ rule. This is similar to the validity condition for propositional μ-$MALL^{\infty}$ [7] where both A_1 and A_2 are sub-occurrences of $A_1 \otimes A_2$ and are considered to be on the same thread as $A_1 \otimes A_2$ (see Section 3.4).

The fresh variable $\mathbf{w}^{\eta}$ introduced in the $\multimap R$ (resp. $\multimap L$) rule switches its place in the sequent from right to left (resp. left to right) and thus it has a dual role in (co)induction compared to $\mathbf{z}^{\beta}$ (resp. $\mathbf{y}^{\alpha}$). We do not consider a relation between $\mathbf{w}^{\eta}$ and $\mathbf{z}^{\beta}$ in the $\multimap R$ (resp. $\mathbf{y}^{\alpha}$ in $\multimap L$). As a result, our condition on $\multimap$ is more restrictive than its classical counterpart $⅋$ in [7] where both A_1 and A_2 are sub-occurrences of $A_1 ⅋ A_2$ (see Section 3.4). Recall that A_1 is already restricted to be an atomic formula in our setting, and no logical rule will apply to it further in the derivation. The restrictions we put on the linear implication allow us to obtain a more straightforward cut-elimination proof than Baelde et al.'s proof. To accept more proofs, one may lift the restriction on linear implication and maintain a relation between A_1 and $A_1 \multimap A_2$ in the intuitionistic setting despite the switch of A_1 between left and right by polarizing all the connectives as described in [74]. This is a generalization we plan to pursue in future work since it is not necessary here in our principal application.

Unlike the existing validity conditions for infinitary calculi defined only over least fixed points [12, 13, 25, 90], priorities are essential in a setting with nested least and greatest fixed points. Here both inductive and coinductive predicates may be unfolded infinitely often along the left and right sides of a branch, but *only the one with the highest priority shall be used to ensure validity of it.*

To sort fixed point unfolding rules applied on previous generations of variable $\mathbf{x}^{\alpha}$ by their priorities we use a snapshot of $\mathbf{x}^{\alpha}$. For a given signature Σ, the *snapshot* of an generational variable $\mathbf{x}^{\alpha}$ is a list $\mathsf{snap}(\mathbf{x}^{\alpha}) = [\mathbf{x}_i^{\alpha}]_{i \leq n}$, where n is the maximum priority in Σ. Having the relation between generational variables in Ω, we can define a partial order on snapshots of generational variables. We write $\mathsf{snap}(\mathbf{x}^{\alpha}) = [\mathbf{x}_1^{\alpha} \cdots \mathbf{x}_n^{\alpha}] <_{\Omega} [\mathbf{z}_1^{\beta} \cdots \mathbf{z}_n^{\beta}] = \mathsf{snap}(\mathbf{z}^{\beta})$if the list $[\mathbf{x}_1^{\alpha} \cdots \mathbf{x}_n^{\alpha}]$ is less than $[\mathbf{z}_1^{\beta} \cdots \mathbf{z}_n^{\beta}]$ by the lexicographic order defined by the transitive closure of the relations in Ω.

Example 4.9. *Consider signature* Σ_1

$$\begin{array}{lll} \mathtt{Stream}(x) & =^1_{\nu} & (\exists y.\exists z.(x = y \cdot z) \otimes \mathtt{Nat}(y) \otimes \mathtt{Stream}\,(z)) \\ \mathtt{Nat}(x) & =^2_{\mu} & (\exists y.(x = \mathsf{s}y) \otimes \mathtt{Nat}(y)) \oplus ((x = \mathsf{z}) \otimes 1) \end{array}$$

$$\dfrac{}{\mathbf{x}^\alpha : A \vdash_\Omega \mathbf{z}^\beta : A}\ \textsc{Id} \qquad \dfrac{\Delta \vdash_\Omega \mathbf{w}^\eta : A \quad \Delta', \mathbf{w}^\eta : A \vdash_\Omega \mathbf{z}^\beta : C}{\Delta, \Delta' \vdash_\Omega \mathbf{z}^\beta : C}\ \textsc{Cut}$$

$$\dfrac{}{\cdot \vdash_\Omega \mathbf{z}^\beta : 1}\ 1R \qquad \dfrac{\Delta \vdash_\Omega \mathbf{z}^\beta : C}{\Delta, \mathbf{y}^\alpha : 1 \vdash_\Omega \mathbf{z}^\beta : C}\ 1L$$

$$\dfrac{\Delta \vdash_{\Omega\cup\{\mathbf{w}_i^\eta = \mathbf{z}_i^\beta | i\leq n\}} \mathbf{w}^\eta : A_1 \quad \Delta' \vdash_\Omega \mathbf{z}^\beta : A_2}{\Delta, \Delta' \vdash_\Omega \mathbf{z}^\beta : A_1 \otimes A_2}\ \otimes R \qquad \dfrac{\Delta, \mathbf{w}^\eta : A_1, \mathbf{y}^\alpha : A_2 \vdash_{\Omega\cup\{\mathbf{w}_i^\eta = \mathbf{y}_i^\alpha | i\leq n\}} \mathbf{z}^\beta : B}{\Delta, \mathbf{y}^\alpha : A_1 \otimes A_2 \vdash_\Omega \mathbf{z}^\beta : B}\ \otimes L$$

$$\dfrac{\Delta, \mathbf{w}^\eta : A_1 \vdash_\Omega \mathbf{z}^\beta : A_2}{\Delta \vdash_\Omega \mathbf{z}^\beta : A_1 \multimap A_2}\ \multimap R \qquad \dfrac{\Delta \vdash_\Omega \mathbf{w}^\eta : A_1 \quad \Delta', \mathbf{y}^\alpha : A_2 \vdash_\Omega \mathbf{z}^\beta : B}{\Delta, \Delta', \mathbf{y}^\alpha : A_1 \multimap A_2 \vdash_\Omega \mathbf{z}^\beta : B}\ \multimap L$$

$$\dfrac{\Delta \vdash_\Omega \mathbf{z}^\beta : A_k \quad k \in I}{\Delta \vdash_\Omega \mathbf{z}^\beta : \oplus\{l_i : A_i\}_{i\in I}}\ \oplus R \qquad \dfrac{\Delta, \mathbf{y}^\alpha : A_i \vdash_\Omega \mathbf{z}^\beta : B \quad \forall i \in I}{\Delta, \mathbf{y}^\alpha : \oplus\{l_i : A_i\}_{i\in I} \vdash_\Omega \mathbf{z}^\beta {:} B}\ \oplus L$$

$$\dfrac{\Delta \vdash_\Omega \mathbf{z}^\beta {:} A_i \quad \forall i \in I}{\Delta \vdash_\Omega \mathbf{z}^\beta : \&\{l_i : A_i\}_{i\in I}}\ \& R \qquad \dfrac{\Delta, \mathbf{y}^\alpha : A_k \vdash_\Omega \mathbf{z}^\beta : B \quad k \in I}{\Delta, \mathbf{y}^\alpha : \&\{l_i : A_i\}_{i\in I} \vdash_\Omega \mathbf{z}^\beta : B}\ \& L$$

$$\dfrac{\Delta \vdash_\Omega \mathbf{z}^\beta : P(t)}{\Delta \vdash_\Omega \mathbf{z}^\beta : \exists x.P(x)}\ \exists R \qquad \dfrac{\Delta, \mathbf{y}^\alpha : P(x) \vdash_\Omega \mathbf{z}^\beta : B \quad x \text{ fresh}}{\Delta, \mathbf{y}^\alpha : \exists x.P(x) \vdash_\Omega \mathbf{z}^\beta : B}\ \exists L_x$$

$$\dfrac{\Delta \vdash_\Omega P :: \mathbf{z}^\beta : P(x) \quad x \text{ fresh}}{\Delta \vdash_\Omega \mathbf{z}^\beta : \forall x.P(x)}\ \forall R_x \qquad \dfrac{\Delta, \mathbf{y}^\alpha : P(t) \vdash_\Omega \mathbf{z}^\beta : B}{\Delta, \mathbf{y}^\alpha : \forall x.P(x) \vdash_\Omega \mathbf{z}^\beta : B}\ \forall L$$

$$\dfrac{\begin{array}{c}\Omega' = \Omega \cup \{\mathbf{z}_i^{\beta+1} = \mathbf{z}_i^\beta \mid i \neq j\} \\ \Delta \vdash_{\Omega'} \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \quad T(\overline{x}) =^j_\mu A\end{array}}{\Delta \vdash_\Omega \mathbf{z}^\beta : T(\overline{t})}\ \mu R \qquad \dfrac{\begin{array}{c}\Omega' = \Omega \cup \{\mathbf{y}_i^{\alpha+1} = \mathbf{y}_i^\alpha \mid i \neq j\} \cup \{\mathbf{y}_j^{\alpha+1} < \mathbf{y}_j^\alpha\} \\ \Delta, \mathbf{y}^{\alpha+1} : [\overline{t}/\overline{x}]A \vdash_{\Omega'} \mathbf{z}^\beta : B \quad T(\overline{x}) =^j_\mu A\end{array}}{\Delta, \mathbf{y}^\alpha : T(\overline{t}) \vdash_\Omega \mathbf{z}^\beta : B}\ \mu L$$

$$\dfrac{\begin{array}{c}\Omega' = \Omega \cup \{\mathbf{z}_i^{\beta+1} = \mathbf{z}_i^\beta \mid i \neq j\} \cup \{\mathbf{z}_j^{\beta+1} < \mathbf{z}_j^\beta\} \\ \Delta \vdash_{\Omega'} \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \quad T(\overline{x}) =^j_\nu A\end{array}}{\Delta \vdash_\Omega \mathbf{z}^\beta : T(\overline{t})}\ \nu R \qquad \dfrac{\begin{array}{c}\Omega' = \Omega \cup \{\mathbf{y}_i^{\alpha+1} = \mathbf{y}_i^\alpha \mid i \neq j\} \\ \Delta, \mathbf{y}^{\alpha+1} : [\overline{t}/\overline{x}]A \vdash_{\Omega'} \mathbf{z}^\beta : B \quad T(\overline{x}) =^j_\nu A\end{array}}{\Delta, \mathbf{y}^\alpha : T(\overline{t}) \vdash_\Omega \mathbf{z}^\beta : B}\ \nu L$$

$$\dfrac{}{\cdot \vdash_\Omega \mathbf{z}^\beta : (s = s)}\ {=}R \qquad \dfrac{\Delta[\theta] \vdash_\Omega \mathbf{z}^\beta : B[\theta] \quad \forall \theta \in \mathtt{mgu}(t, s)}{\Delta, \mathbf{y}^\alpha : (s = t) \vdash_\Omega \mathbf{z}^\beta : B}\ {=}L$$

FIGURE 4.6: Infinitary calculus annotating formulas with labelling variables and their generations.

and variables $\mathbf{x}^\alpha$ *and* $\mathbf{z}^\beta$ *in the judgment* $\mathbf{x}^\alpha{:}\mathtt{Nat}(x), \mathbf{y}^\delta : \mathtt{Stream}(y) \vdash \mathbf{z}^\beta{:}\mathtt{Stream}(x \cdot z)$*. We have* $\mathsf{snap}(\mathbf{x}^\alpha) = [\mathbf{x}_i^\alpha]_{i\leq 2} = [\mathbf{x}_1^\alpha, \mathbf{x}_2^\alpha]$ *and* $\mathsf{snap}(\mathbf{z}^\beta) = [\mathbf{z}_i^\beta]_{i\leq 2} = [\mathbf{z}_1^\beta, \mathbf{z}_2^\beta]$.

Example 4.10. *Let* $\Omega = \{\mathbf{x}_1^\alpha = \mathbf{z}_1^\beta, \mathbf{x}_2^\alpha < \mathbf{z}_2^\gamma, \mathbf{z}_2^\gamma < \mathbf{z}_2^\beta\}$*. For* $\mathsf{snap}(\mathbf{x}^\alpha)$ *and* $\mathsf{snap}(\mathbf{z}^\beta)$ *defined over signature* Σ_1 *in Example 4.9, we have* $\mathsf{snap}(\mathbf{x}^\alpha) = [\mathbf{x}_1^\alpha, \mathbf{x}_2^\alpha] <_\Omega [\mathbf{z}_1^\beta, \mathbf{z}_2^\beta] = \mathsf{snap}(\mathbf{z}^\beta)$.

When comparing the snapshot of a generational variable with the snapshot of a previous generation of it, we can recognize the history of fixed point unfolding rules that has been applied on the formula between the two generations. For example, in the path

$$\dfrac{\begin{array}{c} x^4{:}A_2 \vdash_{\Omega'} \mathbf{y}^0 : B \\ \vdots \end{array}}{x^0{:}A_1 \vdash_{\Omega} \mathbf{y}^0 : B}$$

if $[\mathbf{x}_1^4 \cdots \mathbf{x}_n^4]$ is less than $[\mathbf{x}_1^0 \cdots \mathbf{x}_n^0]$, then we know that a least fixed point unfolding rule with priority i has been applied on a prior generation of x^4 in the path but no greatest fixed point rule with a higher priority than i has been applied on the prior generations of x^4.

We generalize the definitions of left μ-trace and right ν-trace from Fortier and Santocanale and adapt it to our setting.

Definition 4.2. An infinite branch of a derivation is a *left μ-trace* if for infinitely many generational variables $\mathbf{x1}^{\alpha_1}, \mathbf{x2}^{\alpha_2}, \cdots$ appearing as antecedents of judgments $\mathbf{xi}^{\alpha_i} : A_i, \Delta_i \vdash_{\Omega_i} \mathbf{w}^{\beta} : C_i$, in the branch as

$$\begin{array}{c} \vdots \\ \mathbf{x3}^{\alpha_3} : A_3, \Delta_3 \vdash_{\Omega_3} \mathbf{z}^{\eta} : C_3 \\ \hline \vdots \\ \mathbf{x2}^{\alpha_2} : A_2, \Delta_2 \vdash_{\Omega_2} \mathbf{y}^{\delta} : C_2 \\ \hline \vdots \\ \hline \mathbf{x1}^{\alpha_1} : A_1, \Delta_1 \vdash_{\Omega_1} \mathbf{w}^{\beta} : C_1 \\ \vdots \end{array}$$

we can form an infinite chain of inequalities $\mathsf{snap}(\mathbf{x1}^{\alpha_1}) >_{\Omega_2} \mathsf{snap}(\mathbf{x2}^{\alpha_2}) >_{\Omega_3} \cdots$.

Dually, an infinite branch of a derivation is a *right ν-trace* if for infinitely many generational variables $\mathbf{y1}^{\beta_1}, \mathbf{y2}^{\beta_2}, \cdots$ appearing as the succedents of judgments $\Delta_i \vdash_{\Omega_i} \mathbf{yi}^{\beta_i} : C_i$ in the branch as

$$\begin{array}{c} \vdots \\ \Delta_3 \vdash_{\Omega_3} \mathbf{y3}^{\beta_3} : C_3 \\ \hline \vdots \\ \Delta_2 \vdash_{\Omega_2} \mathbf{y2}^{\beta_2} : C_2 \\ \hline \vdots \\ \hline \Delta_1 \vdash_{\Omega_1} \mathbf{y1}^{\beta_1} : C_1 \\ \vdots \end{array}$$

we can form an infinite chain of inequalities $\mathsf{snap}(\mathbf{y1}^{\beta_1}) >_{\Omega_2} \mathsf{snap}(\mathbf{y2}^{\beta_2}) >_{\Omega_3} \cdots$.

Consider a derivation given in the system of Figure 4.1. We can annotate the derivation to get one in the system of Figure 4.6. This can be done productively: we start by annotating the root with arbitrary generational position variables, and continue by replacing the last rule with its annotated version.

Definition 4.3 (Validity condition for infinite derivations)**.** An infinite derivation in the calculus of Figure 4.1 is a *valid proof* if each of the infinite branches in its annotated derivation is either a left μ-trace or a right ν-trace.

We do not represent circular derivations in the annotated calculus directly; instead we unfold them to their underlying infinitary derivations. A circular derivation in the calculus of Figure 4.1 is a *proof* if it has a valid underlying infinite derivation.

We can prove that our validity condition is preserved by substitution.

Lemma 4.4 (Substitution preserves validity)**.** *For a valid derivation*

$$\frac{\Pi}{\Delta \vdash \mathbf{w}^{\alpha} : A}$$

in the infinite system and substitution θ, there is a valid derivation for

$$\frac{\Pi[\theta]}{\Delta[\theta] \vdash \mathbf{w}^{\alpha} : A[\theta]}$$

where $\Pi[\theta]$ is the whole derivation Π or a prefix of it instantiated by θ.

Proof. Similar to the proof of Lemma 4.4. □

Our validity condition, when restricted to the propositional singleton fragment considered by Fortier and Santocanale, is the same as their validity condition.

Example 4.11. *Figure 4.7 presents the first several steps of the derivation from Example 4.4 in the annotated calculus. To check the validity of this derivation, it is enough to observe that*

$$\mathsf{snap}(\mathbf{x}^{\alpha+2}) = [\mathbf{x}_1^{\alpha+2}, \mathbf{x}_2^{\alpha+2}] <_{\Omega_6} [\mathbf{x}_1^{\alpha}, \mathbf{x}_2^{\alpha}] = \mathsf{snap}(\mathbf{x}^{\alpha}).$$

Since the annotation of generational variables is straightforward, for the sake of conciseness, we present future examples as circular derivations in the calculus of Figure 4.1. We also use pattern matching whenever possible.

4.4 A productive cut elimination algorithm

We introduce a cut elimination algorithm for infinite pre-proofs in $FIMALL^{\infty}_{\mu,\nu}$. We prove that this algorithm is productive for valid derivations: the algorithm receives a potentially infinite valid proof as an input and outputs a cut-free (possibly infinite) valid proof productively. An algorithm is productive if every piece of its output is generated in a finite number of steps.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{$=R$}
\UnaryInfC{$\cdot \vdash_{\Omega_7} \mathbf{z}^{\eta} : (\mathtt{ss}y = \mathtt{ss}y)$}
\AxiomC{}
\RightLabel{$=R$}
\UnaryInfC{$\cdot \vdash_{\Omega_6} \mathbf{u}^{\gamma} : \mathtt{ss}z = \mathtt{ss}z$}
\AxiomC{$\vdots$}
\noLine
\UnaryInfC{$\mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_6} \mathbf{y}^{\beta+2} : \mathtt{Odd}(\mathtt{s}z)$}
\RightLabel{$\otimes R$}
\BinaryInfC{$\mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_6} \mathbf{y}^{\beta+2} : (\mathtt{ss}z = \mathtt{ss}z) \otimes \mathtt{Odd}(\mathtt{s}z)$}
\RightLabel{$\exists R$}
\UnaryInfC{$\mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_6} \mathbf{y}^{\beta+2} : (\exists y.(\mathtt{ss}z = \mathtt{s}y) \otimes \mathtt{Odd}(y))$}
\RightLabel{$\oplus R$}
\UnaryInfC{$\mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_6} \mathbf{y}^{\beta+2} : (\exists y.(\mathtt{ss}z = \mathtt{s}y) \otimes \mathtt{Odd}(y)) \oplus (\mathtt{ss}z = 0)$}
\RightLabel{μR}
\UnaryInfC{$\mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_5} \mathbf{y}^{\beta+1} : \mathtt{Even}(\mathtt{ss}z)$}
\RightLabel{$=L$}
\UnaryInfC{$\mathbf{v}^{\zeta} : (y = \mathtt{s}z), \mathbf{x}^{\alpha+2} : \mathtt{Even}(z) \vdash_{\Omega_5} \mathbf{y}^{\beta+1} : \mathtt{Even}(\mathtt{s}y)$}
\RightLabel{$\otimes L$}
\UnaryInfC{$\mathbf{x}^{\alpha+2} : (y = \mathtt{s}z) \otimes \mathtt{Even}(z) \vdash_{\Omega_4} \mathbf{y}^{\beta+1} : \mathtt{Even}(\mathtt{s}y)$}
\RightLabel{$\exists L$}
\UnaryInfC{$\mathbf{x}^{\alpha+2} : \exists z.(y = \mathtt{s}z) \otimes \mathtt{Even}(z) \vdash_{\Omega_4} \mathbf{y}^{\beta+1} : \mathtt{Even}(\mathtt{s}y)$}
\RightLabel{μL}
\UnaryInfC{$\mathbf{x}^{\alpha+1} : \mathtt{Odd}(y) \vdash_{\Omega_3} \mathbf{y}^{\beta+1} : \mathtt{Even}(\mathtt{s}y)$}
\RightLabel{$\otimes R$}
\BinaryInfC{$\mathbf{x}^{\alpha+1} : \mathtt{Odd}(y) \vdash_{\Omega_3} \mathbf{y}^{\beta+1} : (\mathtt{ss}y = \mathtt{ss}y) \otimes \mathtt{Even}(\mathtt{s}y)$}
\RightLabel{$\exists R$}
\UnaryInfC{$\mathbf{x}^{\alpha+1} : \mathtt{Odd}(y) \vdash_{\Omega_3} \mathbf{y}^{\beta+1} : \exists z.(\mathtt{ss}y = \mathtt{s}z) \otimes \mathtt{Even}(z)$}
\RightLabel{μR}
\UnaryInfC{$\mathbf{x}^{\alpha+1} : \mathtt{Odd}(y) \vdash_{\Omega_2} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{ss}y)$}
\RightLabel{$=L$}
\UnaryInfC{$\mathbf{w}^{\delta} : (x = \mathtt{s}y), \mathbf{x}^{\alpha+1} : \mathtt{Odd}(y) \vdash_{\Omega_2} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{s}x)$}
\RightLabel{$\otimes L$}
\UnaryInfC{$\mathbf{x}^{\alpha+1} : (x = \mathtt{s}y) \otimes \mathtt{Odd}(y) \vdash_{\Omega_1} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{s}x)$}
\RightLabel{$\exists L$}
\UnaryInfC{$\mathbf{x}^{\alpha+1} : \exists y.(x = \mathtt{s}y) \otimes \mathtt{Odd}(y) \vdash_{\Omega_1} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{s}x)$}
\AxiomC{$\cdots$}
\RightLabel{$\oplus L$}
\BinaryInfC{$\mathbf{x}^{\alpha+1} : (\exists y.(x = \mathtt{s}y) \otimes \mathtt{Odd}(y)) \oplus ((x = \mathtt{z})) \vdash_{\Omega_1} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{s}x)$}
\RightLabel{μL}
\UnaryInfC{$\mathbf{x}^{\alpha} : \mathtt{Even}(x) \vdash_{\emptyset} \mathbf{y}^{\beta} : \mathtt{Odd}(\mathtt{s}x)$}
\end{prooftree}
$$

$$
\begin{gathered}
\Omega_1 = \{\mathbf{x}_2^{\alpha+1} < \mathbf{x}_2^{\alpha}, \mathbf{x}_1^{\alpha+1} = \mathbf{x}_1^{\alpha}\}, \Omega_2 = \Omega_1 \cup \{\mathbf{w}_2^{\delta} = \mathbf{x}_2^{\alpha+1}, \mathbf{w}_1^{\delta} = \mathbf{x}_1^{\alpha+1}\}, \\
\Omega_3 = \Omega_2 \cup \{\mathbf{y}_1^{\beta+1} = \mathbf{y}_1^{\beta}\}, \Omega_4 = \Omega_3 \cup \{\mathbf{x}_2^{\alpha+2} < \mathbf{x}_2^{\alpha+1}, \mathbf{x}_1^{\alpha+2} = \mathbf{x}_1^{\alpha+1}\}, \\
\Omega_5 = \Omega_4 \cup \{\mathbf{v}_2^{\zeta} = \mathbf{x}_2^{\alpha+2}, \mathbf{v}_1^{\zeta} = \mathbf{x}_1^{\alpha+2}\}, \Omega_6 = \Omega_5 \cup \{\mathbf{y}_1^{\beta+2} = \mathbf{y}_1^{\beta+1}\}, \text{ and} \\
\Omega_7 = \Omega_3 \cup \{\mathbf{z}_2^{\eta} = \mathbf{y}_2^{\beta+1}, \mathbf{z}_1^{\eta} = \mathbf{y}_1^{\beta+1}\}
\end{gathered}
$$

FIGURE 4.7: Successor of an even number is odd in the annotated calculus.

Consider a derivation given in the system of Figure 4.1. We annotate it to get a proof in the system of Figure 4.6. We prove a lemma which states termination of the principal reductions (or internal reductions) of the algorithm (Lemma 4.7) on the annotated derivation. Fortier and Santocanale's proof of a similar lemma for singleton logic is based on an observation that after the principal reductions are applied on a ν-trace in a valid derivation tree, the resulting path only has a finite number of branches on its right side. These branches are created by the cut rule. The branches in the derivation that are created by additive connectives in $\& R$ and $\oplus L$ are not significant in this setting: in the principal cut reduction steps these branches are resolved and exactly one of them remains.

In our calculus, multiplicative connectives can also create a branch by the $\otimes R$ and $\multimap L$ rules. These branches continue to exist even after an internal cut reduction step. We introduce a fresh variable $\mathbf{w}^{\eta}$ as a succedent in the branching rules. In the branches created by $\otimes R$ we keep the relation between the fresh variable $\mathbf{w}^{\eta}$ with its parent $\mathbf{z}^{\beta}$. As a result there may be an infinite ν-trace with infinitely many branches on its right created by the $\otimes R$ rule. To take advantage of a similar observation to Fortier and Santocanale's, we distinguish between branching on fresh succedent generational variables created by a cut or a $\multimap L$ rule, and a $\otimes R$ rule. After this distinction, our cut elimination algorithm creates a trace which is a chain complete partially ordered set, rather than a complete lattice in Fortier and Santocanale's proof. We show that having a chain complete partially ordered set is enough for proving the lemma both in our setting and the subsingleton setting by slightly modifying Fortier and Santocanale's argument. As a corollary to this lemma our cut elimination algorithm produces a potentially infinite cut-free proof for the annotated derivation. We further prove that the output of the cut elimination algorithm is valid. By simply ignoring the annotations of the output, we get a cut free valid derivation in the calculus of Figure 4.1.

Since we are dealing with infinite derivations, to make the algorithm productive we need to push every cut away from the root with a lazy strategy. With this strategy we may need to permute two consecutive cuts which results into a loop. To overcome this problem, similarly to Fortier and Santocanale and also Baelde et al. [7] we generalize binary cuts to n-ary cuts using the notion of a *branching tape.*

Definition 4.5. A *branching tape* $\mathcal{C}$ is a finite list of sequents[2] $\Delta \vdash \mathbf{w}^{\beta} : A$, such that

- Every two judgments $\Delta \vdash \mathbf{w}^{\beta} : A$ and $\Delta' \vdash \mathbf{w}'^{\beta'} : A'$ on the tape share at most one generational variable $\mathbf{z}^{\alpha} : B$. If they share such a generational variable, we call them connected. Moreover, assuming that $\Delta \vdash \mathbf{w}^{\beta} : A$ appears before $\Delta' \vdash \mathbf{w}'^{\beta'} : A'$ on the list, we have $\mathbf{z}^{\alpha} : B \in \Delta'$ and $\mathbf{z}^{\alpha} : B = \mathbf{w}^{\beta} : A$.
- Each generational variable $\mathbf{z}^{\beta}$ appears at most twice in a tape, and if it appears more than once it connects two judgments.
- $\mathcal{C}$ is connected.

The *conclusion* $\mathsf{conc}_{\mathcal{M}}$ of a branching tape $\mathcal{M}$ is a sequent $\Delta \vdash \mathbf{x}^{\alpha} : A$ such that

- For some Δ', there is a sequent $\Delta' \vdash \mathbf{x}^{\alpha} : A$ in the tape that $\mathbf{x}^{\alpha} : A$ does not connect it to any other sequent in the tape.
- For every $\mathbf{y}^{\beta} : B \in \Delta$ there is a sequent $\Delta', \mathbf{y}^{\beta} : B \vdash \mathbf{z}^{\gamma} : C$ on the tape (for some Δ' and $\mathbf{z}^{\gamma} : C$) such that $\mathbf{y}^{\beta} : B$ does not connect it to any other sequent in the tape.

We call Δ the set of *leftmost formulas* of $\mathcal{M}$: $\mathsf{lft}(\mathcal{M})$. And $\mathbf{x}^{\alpha} : A$ is the *rightmost formula* of tape $\mathcal{M}$: $\mathsf{rgt}(\mathcal{M})$.

Lemma 4.6. *Every branching tape has a unique conclusion.*

Proof. By definition a branching tape is connected and acyclic. Therefore its conclusion always exists and is unique. □

[2]For brevity we elide the set Ω in the judgments.

An n-ary cut is a rule formed from a tape $\mathcal{M}$ and its conclusion $\mathsf{conc}_{\mathcal{M}}$: $\dfrac{\mathcal{M}}{\mathsf{conc}_{\mathcal{M}}}\; nCut$

We generalize Fortier and Santocanale's set of primitive operations to account for $FIMALL^{\infty}_{\mu,\nu}$. They closely resemble the reduction rules given by Doumane [33]. Figure 4.8 depicts a few interesting internal (PRd) and external (Flip) reductions, identity elimination, and merging a cut. It is straightforward to adapt the remainder of the reduction steps from the previous work [33, 36].

Our cut elimination algorithm is given as Algorithms 1 and 2. We define a function *Treat* that reduces the sequence in a branching tape with principal reductions (PRd) until either a left rule is applied on one of its leftmost formulas or a right rule is applied on its rightmost formula. While this condition holds, the algorithm applies a *flip* rule on a leftmost/rightmost formula of the tape (LFlip or RFlip). The flipping step is always productive since it pushes a cut one step up. It suffices to show that the principal reductions are terminating to prove productivity of the algorithm. We prove termination of the principal reductions in Lemma 4.7.

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta' \vdash \mathbf{z}^\beta : P(t)}{\Delta' \vdash \mathbf{z}^\beta : \exists x.P(x)}\,\exists R \quad \mathcal{C}_2 \quad \dfrac{\overset{\Pi'}{\Delta'', \mathbf{z}^\beta : P(x) \vdash \mathbf{w}^\alpha : B}}{\Delta'', \mathbf{z}^\beta : \exists x.P(x) \vdash \mathbf{w}^\alpha : B}\,\exists L \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut \;\xRightarrow{\mathsf{PRd}}$$

$$\dfrac{\mathcal{C}_1 \quad \Delta' \vdash \mathbf{z}^\beta : P(t) \quad \mathcal{C}_2 \quad \overset{\Pi'[t/x]}{\Delta'', \mathbf{z}^\beta : P(t) \vdash \mathbf{w}^\alpha : B} \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{}{\cdot \vdash \mathbf{z}^\beta : s = s}\,{=}R \quad \mathcal{C}_2 \quad \dfrac{\Delta'' \vdash \mathbf{w}^\alpha : B}{\Delta'', \mathbf{z}^\beta : s = s \vdash \mathbf{w}^\alpha : B}\,{=}L \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut \;\xRightarrow{\mathsf{PRd}}\; \dfrac{\mathcal{C}_1 \quad \mathcal{C}_2 \quad \Delta'' \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta'_1 \vdash \mathbf{u}^\eta : A_1 \quad \Delta'_2 \vdash \mathbf{z}^\beta : A_2}{\Delta' \vdash \mathbf{z}^\beta : A_1 \otimes A_2}\,\otimes R \quad \mathcal{C}_2 \quad \dfrac{\Delta'', \mathbf{u}^\eta : A_1, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B}{\Delta'', \mathbf{z}^\beta : A_1 \otimes A_2 \vdash \mathbf{w}^\alpha : B}\,\otimes L \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut \;\xRightarrow{\mathsf{PRd}}$$

$$\dfrac{\mathcal{C}_1 \quad \Delta'_1 \vdash \mathbf{u}^\eta : A_1 \quad \Delta'_2 \vdash \mathbf{z}^\beta : A_2 \quad \mathcal{C}_2 \quad \Delta'', \mathbf{u}^\eta : A_1, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta', \mathbf{u}^\eta : A_1 \vdash \mathbf{z}^\beta : A_2}{\Delta' \vdash \mathbf{z}^\beta : A_1 \multimap A_2}\,{\multimap}R \quad \mathcal{C}_2 \quad \dfrac{\Delta''_1 \vdash \mathbf{u}^\eta : A_1 \quad \Delta''_2, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B}{\Delta'', \mathbf{z}^\beta : A_1 \multimap A_2 \vdash \mathbf{w}^\alpha : B}\,{\multimap}L \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut \;\xRightarrow{\mathsf{PRd}}$$

$$\dfrac{\mathcal{C}_1 \quad \mathcal{C}_2 \quad \Delta''_1 \vdash \mathbf{u}^\eta : A_1 \quad \Delta', \mathbf{u}^\eta : A_1 \vdash \mathbf{z}^\beta : A_2 \quad \Delta''_2, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta' \vdash \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \quad T(\overline{x}) =_\mu A}{\Delta' \vdash \mathbf{z}^\beta : T(\overline{t})}\,\mu R \quad \mathcal{C}_2 \quad \dfrac{\Delta'', \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \vdash \mathbf{w}^\alpha : B \quad T(\overline{x}) =_\mu A}{\Delta'', \mathbf{z}^\beta : T(\overline{t}) \vdash \mathbf{w}^\alpha : B}\,\mu L \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut \;\xRightarrow{\mathsf{PRd}}$$

$$\dfrac{\mathcal{C}_1 \quad \Delta' \vdash \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \quad \mathcal{C}_2 \quad \Delta'', \mathbf{z}^{\beta+1} : [\overline{t}/\overline{x}]A \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_3}{\Delta \vdash \mathbf{v} : C}\; nCut$$

FIGURE 4.8: Primitive operations.

$$\dfrac{\mathcal{C} \quad \dfrac{\Delta'_1 \vdash \mathbf{u}^\eta : A_1 \quad \Delta'_2 \vdash \mathbf{z}^\beta : A_2}{\Delta'_1, \Delta'_2 \vdash \mathbf{z}^\beta : A_1 \otimes A_2}\otimes R}{\Delta_1, \Delta_2 \vdash \mathbf{z}^\beta : A_1 \otimes A_2}\, nCut \quad \xRightarrow{\mathsf{RFlip}} \quad \dfrac{\dfrac{\mathcal{C}_{\Delta'_1} \quad \Delta'_1 \vdash \mathbf{u}^\eta : A_1}{\Delta_1 \vdash \mathbf{u}^\eta : A_1}\, nCut \quad \dfrac{\mathcal{C}_{\Delta'_2} \quad \Delta'_2 \vdash \mathbf{z}^\beta : A_2}{\Delta_2 \vdash \mathbf{z}^\beta : A_2}\, nCut}{\Delta_1, \Delta_2 \vdash \mathbf{z}^\beta : A_1 \otimes A_2}\otimes R$$

$\mathcal{C}_{\Delta'_1}$ in the above reduction is a subset of the tape $\mathcal{C}$ connected to Δ'_1. By definition of tape, two sets $\mathcal{C}_{\Delta'_1}$ and $\mathcal{C}_{\Delta'_2}$ partition $\mathcal{C}$.

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta', \mathbf{u}^\eta : A_1, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B}{\Delta', \mathbf{z}^\beta : A_1 \otimes A_2 \vdash \mathbf{w}^\alpha : B}\otimes L \quad \mathcal{C}_2}{\Delta, \mathbf{z}^\beta : A_1 \otimes A_2 \vdash \mathbf{v} : C}\, nCut \quad \xRightarrow{\mathsf{LFlip}} \quad \dfrac{\dfrac{\mathcal{C}_1 \quad \Delta', \mathbf{u}^\eta : A_1, \mathbf{z}^\beta : A_2 \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_2}{\Delta, \mathbf{u}^\eta : A_1, \mathbf{z}^\beta : A_2 \vdash \mathbf{v} : C}\, nCut}{\Delta, \mathbf{z}^\beta : A_1 \otimes A_2 \vdash \mathbf{v} : C}\otimes L$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta'[\theta] \vdash \mathbf{w}^\alpha : B'[\theta] \quad \forall \theta \in \mathtt{mgu}(t,s)}{\Delta', \mathbf{z}^\beta : s = t \vdash \mathbf{w}^\alpha : B} = L \quad \mathcal{C}_2}{\Delta, \mathbf{z}^\beta : s = t \vdash \mathbf{w}^\alpha : B}\, nCut \quad \xRightarrow{\mathsf{LFlip}}$$

$$\dfrac{\dfrac{\mathcal{C}_1[\theta] \quad \Delta'[\theta] \vdash \mathbf{w}^\alpha : B'[\theta] \quad \mathcal{C}_2[\theta]}{\Delta[\theta] \vdash \mathbf{w}^\alpha : B[\theta]}\, nCut \quad \forall \theta \in \mathtt{mgu}(t,s)}{\Delta, \mathbf{z}^\beta : s = t \vdash \mathbf{w}^\alpha : B} = L$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{}{\mathbf{x}^\alpha : A \vdash \mathbf{w}^\gamma : A}\, ID \quad \mathcal{C}_2}{\Delta \vdash \mathbf{z}^\beta : C}\, nCut \quad \xRightarrow{\mathsf{ID-Elim}} \quad \dfrac{\mathcal{C}_1 \quad \mathcal{C}_2[\mathbf{x}^\alpha / \mathbf{w}^\gamma]}{\Delta \vdash \mathbf{z}^\beta : C}\, nCut$$

$$\dfrac{\mathcal{C}_1 \quad \dfrac{\Delta' \vdash \mathbf{z}^\beta : A \quad \Delta'', \mathbf{z}^\beta : A \vdash \mathbf{w}^\alpha : B}{\Delta', \Delta'' \vdash \mathbf{w}^\alpha : B}\, \textsc{Cut} \quad \mathcal{C}_2}{\Delta \vdash \mathbf{v} : C}\, nCut \quad \xRightarrow{\mathsf{Merge}} \quad \dfrac{\mathcal{C}_1 \quad \Delta' \vdash \mathbf{z}^\beta : A \quad \Delta'', \mathbf{z}^\beta : A \vdash \mathbf{w}^\alpha : B \quad \mathcal{C}_2}{\Delta \vdash \mathbf{v} : C}\, nCut$$

FIGURE 4.8: Primitive operations.

Lemma 4.7. *For every input tape M, computation of $Treat(M)$ halts.*

Proof. We show that $Treat(M)$ does not have an infinite computation. Assume for the sake of contradiction that $Treat(M)$ has an infinite computation and iterates indefinitely. Put M_i for $i \geq 1$ to be the branching tape before the i-th iteration of the loop, with $M_1 = M$.

We build the full trace T of the algorithm. T is a tree with nodes of the form (n, k) and a designated root $(0, 0)$. A node (n, k) corresponds to the k-th element of the branching tape M_n. We produce T coinductively with a level order traversal: when the n-th iteration of the loop in the Treat function creates a new tape M_{n+1} from M_n we add the nodes $(n + 1, k)$ corresponding to the sequents in M_{n+1} to the $n+1$-th level of the tree and connect them with labeled edges to the nodes in the n-th level of the tree. We provide the rules for building the edges from level n to $n + 1$ based on the step that is used to create the tape M_{n+1} from M_n, i.e. internal cut reductions, merging a cut, and identity elimination. For merging a cut, identity elimination, and internal cut reductions for additive connectives, we use essentially the same rules as in Fortier and Santocanale[36]. The rules for multiplicative internal cut reductions are different but based on a similar idea.

Algorithm 1: Cut elimination algorithm

Description: Q is a queue with the elements of the form (w, M) where M is a tape, and w is the node that was previously computed. The output of the algorithm is a tree labelled by $\{0,1\}$. Each node of the tree is identified with a sequent of 0 and 1s: $w \in \{0,1\}^*$. For each node in the tree, we also compute the corresponding sequent, $s(w)$, and the rule applied on it, $r(w)$. $\rho(s)$ is the rule applied on formula labelled with variable s, it can either be an Id, Cut, a L rule, or a R rule. $\mathsf{lft}(M)$ and $\mathsf{rgt}(M)$ are defined in Definition 4.5. The FLip rules return the rule that is permuted down after the external reduction step to prove w, the sequent corresponding to w, and a list $List$ of one or two tapes to continue with.

```
Initialization: $\Lambda \leftarrow \emptyset$; $Q \leftarrow [(\epsilon, [v])]$; $v$ is the root sequent.
while $Q \neq \emptyset$ do
    $(w, M) \leftarrow pull(Q)$;
    $\Lambda \leftarrow \Lambda \cup \{w\}$;
    $M \leftarrow Treat(M)$;
    if $\exists s \in \mathsf{lft}(M).\rho(s) \in L$ then
        $(r(w), s(w), List) \leftarrow \mathsf{LFlip}(M)$;
    else
        if $\exists s \in \mathsf{rgt}(M).\rho(s) \in R$ then
            $(r(w), s(w), List) \leftarrow \mathsf{RFlip}(M)$;
        end
    end
    if $List = [M']$ then
        $push((w0, M'), Q)$;
    else
        if $List = [M'_0, M'_1]$ then
            $push((w0, M'_0), Q)$;
            $push((w1, M'_1), Q$;
        end
    end
end
```

The initial step is to add an edge labeled by i from the root to each node corresponding to the i-th sequent in the initial tape M_1, i.e. $(1, i)$.

$$\text{For } 1 \leq i \leq |M_1|, (0,0) \rightarrow^i (1, i).$$

Next, we provide the rules for producing edges of T when the Treat function applies an identity elimination or merges a cut on the tape M_n

- If $M_{n+1} = \mathsf{ID} - \mathsf{Elim}(M_n, i)$ then
  - $(n, k) \rightarrow^\perp (n+1, k)$ for $k < i$,
  - $(n, k) \rightarrow^\perp (n+1, k-1)$ for $k > i$.

Algorithm 2: Treat Function

Description: M is a branching tape. i and j in $\mathsf{PRd}(M, i, j)$ are the index of the two sequents in tape on which the reduction rules are applied. Similarly i in $\mathsf{Merge}(M, i)$ and $\mathsf{ID} - \mathsf{Elim}$ is the index of the sequent in the tape on which the corresponding rule is applied. $\rho'(i)$ is the rule applied on the i-th sequent of the tape, it can either be an Id, Cut, a L rule, or a R rule.

```
while $\rho(\mathsf{lft}(M)) \notin L$ and $\rho(rgt(M)) \notin R$ and $|M| > 0$ do
    if $\exists i \in M : \rho'(i) = \textsc{Id}$ then
        $M \leftarrow \mathsf{ID} - \mathsf{Elim}(M, i)$;
    else
        if $\exists i \in M : \rho'(i) = \textsc{Cut}$ then
            $M \leftarrow \mathsf{Merge}(M, i)$;
        else
            if $\exists i.\exists j.\exists \circ \in \{1, \oplus, \&, \otimes, \multimap\}.\rho'(i) = \circ R$ and $\rho'(i) = \circ L$ then
                $M \leftarrow \mathsf{PRd}(M, i, j)$;
            end
        end
    end
end
```

- If $M_{n+1} = \mathsf{Merge}(M_n, i)$ then
  - $(n, k) \rightarrow^{\perp} (n+1, k)$ for $k < i$,
  - $(n, i) \rightarrow^{1} (n+1, i)$,
  - $(n, i) \rightarrow^{2} (n+1, i+1)$,
  - $(n, k) \rightarrow^{\perp} (n+1, k+1)$ for $k > i$.

Edges labeled by $\perp$ mean that the sequent has not evolved by the operation. We use labels 1 and 2 in the step corresponding to $\mathsf{Merge}(M_n, i)$. Edges labeled by 1 and 2 connecting (n, i) with $(n+1, i)$ and $(n+1, i+1)$, respectively, show that the sequent i-th of the tape M_n evolves to two new sequents: the i-th and $i+1$-th sequents of the tape M_{n+1}. Naturally, we have the relation $1 < 2$ between the labels. Observe that the sequents corresponding to the nodes $(n+1, i)$ and $(n+1, i+1)$ are connected via the fresh generational variable created by the cut rule.

Recall from Algorithm 2 that function $\mathsf{PRd}(M_n, i, j)$ receives a tape M and two indices i, j corresponding to the position of two sequents in the tape, applies an internal cut reduction on the sequents at positions i and j and outputs the new tape M_{n+1}. One difference between our algorithm, and Fortier and Santocanale's is that in our algorithm the sequents subject to reduction may not be next to each other. Thus, in our case the PRd function needs to receive the index of both sequents. Moreover, having the multiplicative connectives we need to deal with branching internal reductions too. All reductions except those corresponding to $\otimes$ and $\multimap$ are non-branching(nb). For the non-branching reductions the rules for creating the edges of T are quite similar to the ones introduced by Fortier and Santocanale[36]. For brevity, we put function $\mathsf{PRd}_{nb}(M_n, i, j)$ to stand for all internal reductions except $\otimes$ and $\multimap$. The rules for building the next level of T for this collection of reductions is as follows:

- If $M_{n+1} = \mathsf{PRd}_{nb}(M_n, i, j)$ then
  - $(n, k) \to^{\perp} (n+1, k)$ for $k \notin \{i, j\}$,
  - $(n, i) \to^{0} (n+1, i)$,
  - $(n, j) \to^{0} (n+1, j)$.

The label 0 connecting (n, i) and $(n+1, i)$ for example indicates that the sequent corresponding to the node (n, i) evolves to a new sequent corresponding to $(n+1, i)$ but it does not spawn any new sequent and thus does not create a branch.

The reductions corresponding to $\otimes$ and $\multimap$, however, produce a branch. We define the rules for building T when the algorithm applies such branching steps separately:

- If $M_{n+1} = \mathsf{PRd}_{\otimes}(M_n, i, j)$ then
  - $(n, k) \to^{\perp} (n+1, k)$ for $k < i$,
  - $(n, i) \to^{1_a} (n+1, i)$ and $(n, i) \to^{1_b} (n+1, i+1)$,
  - $(n, j) \to^{0} (n+1, j+1)$,
  - $(n, k) \to^{\perp} (n+1, k+1)$ for $i < k < j$ or $k > j$.
- If $M_{n+1} = \mathsf{PRd}_{\multimap}(M_n, i, j)$ then
  - $(n, k) \to^{\perp} (n+1, k)$ for $k < i$,
  - $(n, i) \to^{0} (n+1, j)$,
  - $(n, k) \to^{\perp} (n+1, k-1)$ for $i < k < j$,
  - $(n, j) \to^{1} (n+1, j-1)$ and $(n, j) \to^{2} (n+1, j+1)$,
  - $(n, k) \to^{\perp} (n+1, k+1)$ for $k > j$.

Labels $\perp$ and 0 are used with a similar meaning as before. In the internal reduction for the multiplicative conjunction ($\otimes$), the i-th sequent of the tape M_n is replaced by two sequents, the i-th and $i+1$-th sequents of the tape M_{n+1}. We connect the nodes corresponding to both these new sequents to (n, i) using two distinct labels 1_a and 1_b. We extend the order $<$ on natural numbers $\mathbb{N}$ to an order on $\mathbb{N} \cup \{1_a, 1_b\}$ such that 1_a and 1_b are incomparable to each other. (We can also extend the order to include $1 < 1_a, 1_b < 2$, but it is not significant in our proof.)

The internal reduction of the linear implication ($\multimap$) creates a branch too: the j-th sequent of the tape M_n is replaced by two sequents, the $j-1$-th and $j+1$-th sequents of the tape M_{n+1}. We connect the nodes corresponding to these new sequents to (n, j) using labels 1 and

2. More importantly, the reduction rule shifts the i-th element of M_n to position j at the tape M_{n+1}: in the new tape the sequent corresponding to $(n+1, j)$ is at the right of the sequent corresponding to $(n+1, j-1)$.

Labels $1_a, 1_b, 1, 2$ are to distinguish between two types of branching in Ψ: (i) the branching that occurs in Merge and $\mathsf{PRd}_{\multimap}$ rules are labeled with 1 and 2 , and (ii) the branching in the $\mathsf{PRd}_{\otimes}$ is labeled with 1_a and 1_b. In the first case the branch labeled with 1 is lexicographically less than the branch labeled with 2 since $1 < 2$, while in the second case the branches are incomparable (1_a is incomparable to 1_b).

Recall that the edges labelled by $\perp$ connect two identical sequents. We get the real trace Ψ by collapsing these $\perp$-edges. Ψ is an infinite, finitely branching labelled tree with prefix order $\sqsubseteq$ and lexicographical order $<$ (based on the order on natural numbers extended with labels 1_a and 1_b). A branch in Ψ is a maximal path with respect to $\sqsubseteq$. The set of all branches of Ψ ordered lexicographically forms a chain complete partially ordered set, meaning that a set of branches that form a $<$-chain has a least upper bound and a greatest lower bound. We provide a simple productive procedure of computing the greatest lower bound β for a chain of branches $\{\gamma_i\}_{i \in I}$ as it will be used later in the proof. The procedure assumes that (a) the greatest lower bound β is constructed up to (not including) its i-th element, and (b) it receives a chain of branches as an input such that they all have the same prefix up to (not including) the i-th element. With these assumptions the following provides an algorithm to find the i-th element of the greatest lower bound β. The assumptions clearly hold when we call the procedure for the first time to construct the first element of β, and it is preserved by each recursive call:

Compare the i-th elements of the given branches and choose the least one (the number of labels is finite and all of them are comparable). Put the i-th element of β to be the chosen label. Next, discard all branches that their i-th element is any other label, and repeat the procedure on the remaining branches to find the $i+1$-th element.

Before proceeding with the proof, we state and prove the main observations that we use in the rest of the proof.

Observation 1. Consider two sequents $\Gamma' \vdash \mathbf{x}^\alpha : A$ and $\Gamma, \mathbf{x}^\alpha : A \vdash \mathbf{y}^\beta : B$ on a branching tape M_n where A is not an atomic formula. The path in Ψ starting from the root and ending in the node corresponding to $\Gamma' \vdash \mathbf{x}^\alpha : A$ is lexicographically less than the path starting from the root and ending in the node $\Gamma, \mathbf{x}^\alpha : A \vdash \mathbf{y}^\beta : B$.

Proof. We prove that this property holds as an invariant of each tape. By Definition 4.5, the invariant holds for the starting tape M_1. We assume that the invariant holds for tape M_n and prove that it holds for tape M_{n+1} created by each possible step of the Treat function, i.e. a principal reduction, Merge, or Identity elimination. The proof is straightforward for all cases except the principal reduction for $\multimap$ and identity elimination. Consider the case of reduction for $\multimap$, in which the principal reduction is applied on the sequents at positions i and j of tape M_n (with $i < j$) as shown in Figure 4.8. Two new connections are formed in the new tape M_{n+1}: $\Delta''_1 \vdash \mathbf{u}^\eta : A_1$ is connected to $\Delta', \mathbf{u}^\eta : A_1 \vdash \mathbf{z}^\beta : A_2$ and $\Delta', \mathbf{u}^\eta : A_1 \vdash$

$\mathbf{z}^{\beta} : A_2$ is connected to $\Delta_2'', \mathbf{z}^{\beta} : A_2 \vdash \mathbf{w}^{\alpha} : B$. By the restriction on the assumption of a linear implication we know that A_1 is an atomic formula and thus the first connection can be dismissed. It is enough to prove that the path from the root to node $(n+1, j)$ is less than the path from the root to the node $(n+1, j+1)$. To get this we can simply use the assumption that the Tape M_n satisfies the invariant.

Next, consider identity elimination applied on the i-th sequent of the tape M_n. The interesting case is when the i-th sequent is of the form $\mathbf{x}^{\alpha}{:}A \vdash \mathbf{w}^{\gamma}{:}A$ and is connected to two sequents $\Gamma \vdash \mathbf{x}^{\alpha}{:}A$ and $\Gamma', \mathbf{w}^{\gamma}{:}A \vdash \mathbf{z}^{\beta}{:}C$. In the resulting tape M_{n+1}, $\mathbf{x}^{\alpha}{:}A \vdash \mathbf{w}^{\gamma}{:}A$ is deleted and by renaming the variables a new connection is created between $\Gamma \vdash \mathbf{x}^{\alpha}{:}A$ and $\Gamma', \mathbf{x}^{\alpha}{:}A \vdash \mathbf{z}^{\beta}{:}C$. If A is an atom this connection is not significant for our proof and can be dismissed. If A is not an atom then by assumption M_n satisfies the invariant and by transitivity of the lexicographic order we know that the path from the root to the nodes corresponding to sequents $\Gamma \vdash \mathbf{x}^{\alpha}{:}A$ and $\Gamma', \mathbf{x}^{\alpha}{:}A \vdash \mathbf{z}^{\beta}{:}C$ already satisfies the required condition.

Definition. An infinite branch in Ψ is a μ-branch (resp. ν-branch) if its corresponding path in the derivation is a μ-trace (resp. ν-trace).

Observation 2. Our validity condition implies that the (infinite) label of a ν-branch has only finitely many occurrences of 1.

Proof. Whenever we create a branch labeled by 1 (either when merging a cut or in a principal reduction for $\multimap$), we introduce a fresh variable as a succedent that does not relate to any other prior generational variable. As a result, a ν-trace whose definition depends on the chain of relationships formed between its succedents can only accept infinitely many 1 labels.

We prove the following three contradictory statements:

(i) An infinite branch of Ψ which is not less than any other infinite branches (a maximal infinite branch) exists and it is a μ-branch:

We first prove that such a maximal branch exists. Assume that we add $1_a < 1_b$ to the ordering, then the set of all branches of Ψ forms a complete lattice, and by Konig's lemma it has a greatest infinite branch γ. This branch is maximal if we dismiss the relation $1_a < 1_b$ from the ordering.

Consider a maximal infinite branch γ in Ψ. By validity of the derivation, it is either a μ- or a ν-branch. Assume it is a ν-branch. There is an infinite chain of inequalities for generational variables $\mathbf{x1}^{\alpha_1}, \mathbf{x2}^{\alpha_2}, \cdots$ on the succedents of γ:

$$\mathsf{snap}(\mathbf{x1}^{\alpha_1}) >_{\Omega_1^{\gamma}} \mathsf{snap}(\mathbf{x2}^{\alpha_2}) >_{\Omega_2^{\gamma}} \cdots .$$

Recall that no logical rule can be applied on an atomic formula. As a result, by the way we defined our validity condition, none of the variables $\mathbf{xi}^{\alpha_i}$ annotate an atomic formula. Moreover, by the definition of the Treat function each generational variable $\mathbf{xi}^{\alpha_i}$ occurs as an antecedent of a branch. For each generational variable $\mathbf{xi}^{\alpha_i}$ we can build a branch β_i: we start by connecting the root to the node corresponding to a sequent in which $\mathbf{xi}^{\alpha_i}$ is an antecedent. We then follow the path starting from the sequent to the next generational variable that it has in common with γ. By the structure of Ψ and duality of the left and right rules, we can either find the next common generational variable between β_i and γ, or β_i is a finite branch and terminates. In both cases, we can productively build a branch β_i. We form a set $\{\beta_i\}_{i\in I}$ from the branches we built. This set can have one or more elements. By Observation 1, we get $\gamma < \beta_i$ for every $i \in I$.

If there is an infinite branch β in $\{\beta_i\}_{i\in I}$, we can form a contradiction with the maximality of γ since $\gamma < \beta$.

Otherwise all branches in $\{\beta_i\}_{i\in I}$ are finite and thus the index set I has to be infinite. Next, we show in this case we can also build an infinite branch $\beta > \gamma$ productively. The procedure assumes that (a) the infinite branch β is constructed up to (not including) its i-th element, and (b) it receives an infinite set of branches as an input such that they all have the same prefix up to (not including) the i-th element. With these assumptions the following provides an algorithm to find the i-th element of β. The assumptions clearly hold when we call the procedure for the first time to construct the first element of β, and it is preserved by each recursive call:

Compare the i-th elements of the given branches and choose one that appears infinitely often (the number of distinct labels is finite and at least one of them has to appear infinitely often). Put the i-th element of β to be the chosen label. Next, discard all branches that their i-th element is any other label, and repeat the procedure on the remaining branches to find the $i+1$-th element.

By the way β is synthesized, it is greater or equal to γ. Assume $\gamma = \beta$, it means that each prefix of γ is the prefix of infinitely many branches in $\{\beta_i\}_{i\in I}$. By $\gamma < \beta_i$ for every $i \in I$, we conclude that $\gamma = \beta$ has infinitely many occurrences of 1 on its label. This forms a contradiction with γ being a ν-trace. As a result, we know that $\gamma < \beta$ and we can again form a contradiction with the maximality of γ.

(ii) Let γ be a maximal infinite branch (an infinite branch of Ψ which is not less than any other infinite branches with respect to the lexicographic ordering). Form a decreasing chain of μ-branches in Ψ starting from γ: $\cdots < \beta_2 < \beta_1 < \gamma$. Put E to be the elements of this chain. Then $\eta = \bigwedge E$ exists since Ψ is chain complete and it is a μ-branch: If $\eta \in E$ then it is trivially true. Otherwise, by the way we construct η each prefix of η is the prefix of infinitely many branches in E. By a similar reasoning to the previous case we get that η has infinitely many occurrences of 1 on its label. By Observation 2, it cannot be a ν-branch and thus is a μ-branch.

(iii) If β is a μ-branch, then there exists another μ-branch $\beta' < \beta$:

β is a μ-branch so for infinitely many generational variables $\mathbf{x1}^{\alpha_1}, \mathbf{x2}^{\alpha_2}, \cdots$ on the antecedents of β we can form an infinite chain of inequalities

$$\mathsf{snap}(\mathbf{x1}^{\alpha_1}) >_{\Omega_1^\beta} \mathsf{snap}(\mathbf{x2}^{\alpha_2}) >_{\Omega_2^\beta} \cdots .$$

Recall that none of the variables $\mathbf{xi}^{\alpha_i}$ annotate an atomic formula. Moreover, by the definition of the Treat function each generational variable $\mathbf{xi}^{\alpha_i}$ occurs as the succedent of a branch. We build the branch β_i for each $\mathbf{xi}^{\alpha_i}$ similar to part (i): for each generational variable $\mathbf{xi}^{\alpha_i}$ we can build the branch β_i: we start by connecting the root to the node corresponding to the sequent in which $\mathbf{xi}^{\alpha_i}$ is the succedent. We then follow the path starting from the sequent to the next generational variable that it has in common with β. By the structure of Ψ and duality of the left and right rules, we can either find the next common generational variable between β_i and β, or β_i is a finite branch and terminates. In both cases, we can productively build the branch β_i.

We form a set of branches $\{\beta_i\}_{i\in I}$ with one or more elements. By Observation 1, we get $\beta_1 < \beta$ for every $i \in I$. Observe that two branches β_i and β_{i+1} have a close relation: either (1) β_i is equal to β_{i+1}, or (2) β_i is finite and terminates by an identity rule that forward its antecedent $\mathbf{y}$ to its succedent $\mathbf{z}$. Case (2) has two sub-cases: either (2-1) $\mathbf{z}$ is an antecedent of β_{i+1}, or (2-2) $\mathbf{y}$ is the succedent of β_{i+1}.

In Case (2-1) we know that β_{i+1} has a common prefix with β at least up to $\mathbf{z}$ and thus β later spawns β_{i+1} by introducing a fresh variable $\mathbf{x}$ as the succedent of β_{i+1} which is not related to the variable $\mathbf{z}$. This results in a contradiction with the chain of inequalities formed above. In (2-2) we have $\beta_i > \beta_{i+1}$ where β_i is a finite branch.

As a result the set $\{\beta_i\}_{i\in I}$ of branches form a chain, and we can produce its greatest lower bound β'. It is strightforward to observe that β' is less than β, and also is infinite. By the way that we created it, one of the followings hold for β':

(a) $\beta' < \beta$ is an infinite branch with infinitely many generational variables

$$\mathbf{xi}^{\alpha_i}, \mathbf{x\{i+1\}}^{\alpha_{i+1}}, \cdots$$

as its succedents. These generational variables connect sequents in β to the sequents in β' infinitely many times. So every $\mu/\nu L$ rule in β reduces with a $\mu/\nu R$ rule in β'. This means that a μR rule with priority i is applied on the succedent of β' infinitely often but no priority $j < i$ has an infinitely many νR rule in β'.

(b) $\beta' < \beta$ is an infinite branch with infinitely many occurrences of 1 on its label.

In both cases β' cannot be a ν-branch and thus is a μ-branch.

Items (i)-(iii) form a contradiction. We can form the nonempty collection E of all μ-branches in Ψ that from a maximal decreasing chain starting from γ by (i) and (iii). By (ii) we get $(\eta = \bigwedge E) \in E$ is the minimum of this chain. This forms a contradiction with (iii) and maximality of E.

With a similar reasoning, we can prove that the output of the cut elimination algorithm is also a valid derivation. Since the reasoning of the proof is similar to the above, we only provide a high level description here. Consider a branch b in the output derivation of Algorithm 1. Using a similar set of rules in the above proof we can build a tree T_b for the full algorithm corresponding to branch b (including the treating part). Defining the flip (external reduction) rules for creating tree T_b is straightforward: If the flip rule creates two branches we continue with the branch corresponding to b and dismiss the other one. For example in RFlip for $\otimes R$, if b corresponds to the left branch, we add an edge from the node corresponding to the sequent $\Delta'_1, \Delta'_2 \vdash \mathbf{z}^\beta : A_1 \otimes A_2$ to the node corresponding to the sequent $\Delta'_1 \vdash \mathbf{u}^\eta : A_1$ labeled by 0. For nodes corresponding the sequents in $C_{\Delta'_1}$ we add an edge labeled by $\perp$ and for the nodes corresponding to the sequents in $C_{\Delta'_2}$ we do not add any edges and simply terminate their branches. For non-branching external reductions we use a set of rules similar to the non-branching principal ones. Consider Ψ_b to be the real trace built based on branch b in the derivation produced by collapsing the edges in T_b labeled by $\perp$. It is straightforward to see that Ψ_b is a chain complete partially ordered set, and Observations 1 and 2 hold in this setting too.

If b is finite we are done. Assume that branch b is not finite.

(i') Similar to item (i) we can show that the tree has a maximal infinite branch γ with regard to the lexicographic ordering. If this branch is a ν-branch then we can either form a set of branches $\{\beta_i\}_{i \in I}$ such that $\beta_i > \gamma$, or infinitely many RFlip rules are applied to the succedents of γ to create the branch b. In the first case, we can form a contradiction similar to the reasoning above and it means that γ has to be a μ-branch. In the second case, the proof is complete since b is a valid ν-trace.

(ii') In the previous case we established that γ is a μ-branch. Form a decreasing chain of μ-branches in Ψ starting from γ: $\dots < \beta_2 < \beta_1 < \gamma$. Put E to be the elements of this chain. Then $\eta = \bigwedge E$ exists since Ψ is chain complete and it is a μ-branch by a similar reasoning to item (ii).

(iii') Put E to be a *maximal* decreasing chain of μ-branches in Ψ starting from γ: $\dots < \beta_2 < \beta_1 < \gamma$. Let $\eta = \bigwedge E$ be the greatest lower bound of E. By a similar reasoning to item (iii), we get that either there is a μ-branch $\beta' < \beta$ or the antecedents that make β a μ-trace are the antecedents of branch b in the output derivation. In the first case, we form a contradiction. In the second case, the proof is complete since b is a μ-trace.

□

Theorem 1. A valid (infinite) derivation enjoys the cut elimination property.

Proof. We annotate a given derivation in the system of Figure 4.1 to get a derivation in the system of Figure 4.6 productively (as described in Section 4.3). As a corollary to Lemma 4.7 the cut elimination algorithm (Algorithm 1) produces a potentially infinite valid cut free proof for the annotated derivation. By simply ignoring the annotations of the output, we get a cut free proof in the calculus of Figure 4.1. □

Chapter 5

Session-typed processes

5.1 Background

Communication centered programming (CCP) is an alternative to sequential programming and a central element in software development. CCP is a computational model with the expressive power of λ-calculus [69] and has a broad range of applications. Its applications include networking, business protocols, and multicore programming.

Honda proposed session types as a potential typed foundation for structuring communication centered programming [51]. This model's central objects are interrelated units called sessions or processes, with their name originating from the networking community. The interactions between processes are governed by protocols associated with them. The protocols are called session types and describe the pattern in which processes interact with each other.

The original work introduced session types based on π-calculus and described interactions between sessions based on input/output communication and binary choice [51]. Channels connect processes and transfer the interactions between them. Binary channels, in particular, conduct the interaction between exactly two processes. Honda et al. [52] generalized the actions in session types to sending labels (as opposed to binary choice) and passing channels over channels (session delegation). A duality is central in all interactions between processes: one sends while the other receives.

Other variants of session types have been introduced since the original formulation. Honda et al. [53] proposed multiparty session types that describe interactions containing more than two parties. We are interested in binary session types, with every channel occurring exactly once in a collection of interrelated processes. *Binary session types* have been recognized as arising from linear logic (either in its intuitionistic [15, 16] or classical [98] formulation) by a Curry-Howard interpretation. The connectives in linear logic can model all actions in session types: additive connectives simulate choosing a label, while multiplicative ones simulate session delegation and termination. Under this correspondence, propositions correspond to types, logical proof rules to typing rules, proofs as programs, and cut reduction to communication along channels.

For the intuitionistic binary session types, a configuration of processes connected along their mutual channels forms a tree. The tree's acyclicity guarantees an important safety property called progress (deadlock-freedom): the configuration will never get stuck. Another key property is type preservation (session fidelity), ensuring a configuration remains well-typed after a step of computation [97].

This thesis focuses on subsingleton session-typed processes, which is the fragment corresponding to *subsingleton logic* [29, 31, 73]. In this fragment, the configuration of connected processes forms a chain rather than a tree. We extend the Curry-Howard interpretation of derivations in infinitary subsingleton logic with fixed points as recursive communicating processes. Along with the standard progress and preservation results, we present a strong version of the progress property that ensures each process communicates along its left or right channel in a finite number of steps. Interestingly, the subsingleton fragment with recursive types already has the computational power of Turing machines [31].

5.2 Session typed processes

Under the Curry-Howard interpretation a subsingleton judgment $A \vdash B$ is annotated as

$$x : A \vdash \mathtt{P} :: (y : B)$$

where x and y are two different channels and A and B are their corresponding session types. One can understand this judgment as [31]:

Process $\mathtt{P}$ provides a service of type B along channel y while using channel x of type A, a service that is provided by another process along channel x.

However, since a process might not use any service provided along its left channel, e.g. $\cdot \vdash P :: (y : B)$, or it might not provide any service along its right channel, e.g. $x : A \vdash Q :: (\cdot)$, the labelling of processes is generalized to be of the form:

$$\bar{x} : \omega \vdash P :: (y : C),$$

where $\bar{x}$ is either empty or x , and ω is empty given that $\bar{x}$ is empty.

We can form a chain of processes $\mathtt{P}_0, \mathtt{P}_1, \cdots, \mathtt{P}_n$ with the typing

$$\cdot \vdash \mathtt{P}_0 :: (x_0 : A_0), \quad x_0 : A_0 \vdash \mathtt{P}_1 :: (x_1 : A_1), \quad \cdots \quad x_{n-1} : A_{n-1} \vdash \mathtt{P}_n :: (x_n : A_n)$$

which we write as

$$\mathtt{P}_0 \mid_{x_0} \mathtt{P}_1 \mid_{x_1} \cdots \mid_{x_{n-1}} \mathtt{P}_n$$

in analogy with the notation for parallel composition for processes $P \mid Q$, although here it is not commutative. In such a chain, process $\mathtt{P}_{i+1}$ uses a service of type A_i provided by the process $\mathtt{P}_i$ along the channel x_i, and provides its own service of type A_{i+1} along the channel x_{i+1}. Process $\mathtt{P}_0$ provides a service of type A_0 along channel x_0 without using any services. So, a process in the session type system, instead of being reduced to a value as in functional programming, interacts with both its left and right interfaces by sending and receiving messages. Processes $\mathtt{P}_i$ and $\mathtt{P}_{i+1}$, for example, communicate with each other along the channel x_i of type A_i: if process $\mathtt{P}_i$ sends a message along channel x_i to the right and process $\mathtt{P}_{i+1}$ receives it from the left (along the same channel), session type A_i is called a *positive* type. Conversely, if process $\mathtt{P}_{i+1}$ sends a message along channel x_i to the left and process $\mathtt{P}_i$ receives it from the right (along the same channel), session type A_i is called a *negative* type.

In general, in a chain of processes, the leftmost type may not be empty. Also, strictly speaking, the names of the channels are redundant since every process has two distinguished ports: one to the left and one to the right, either one of which may be empty. Because of this, we may sometimes omit the channel name, but in the theory we present in this thesis it is convenient to always refer to communication channels by unique names.

Definition 5.1. We define *session types* with the following grammar, where L ranges over finite sets of labels denoted by ℓ and k.

$$A ::= \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid 1 \mid \bot$$

The binary disjunction and conjunction are defined as $A \oplus B = \oplus\{\pi_1 : A, \pi_2 : B\}$ and $A \& B = \&\{\pi_1 : A, \pi_2 : B\}$, respectively. Similarly, we define $0 = \oplus\{\}$ and $\top = \&\{\}$.

The restricted judgment of the subsingleton fragment cannot capture the binary multiplicative connectives, i.e. we cannot handle session delegation in this fragment.

All processes we consider in this thesis provide a service along their right channel so in the remainder of the thesis we restrict the sequents to be of the form $\bar{x} : \omega \vdash P :: (y : A)$. We therefore do not need to consider the rules for type $\bot$ anymore, but the results of this thesis can easily be generalized to the fully symmetric calculus.

A summary of the operational reading of session types is presented in Table 5.1. The first column indicates the session type before the message exchange, the second column the session type after the exchange. The corresponding process terms are listed in the third and fourth column, respectively. The fifth column provides the operational meaning of the type and the last column its polarity.

Session type (curr./cont.)		**Process term (curr./cont.)**		**Description**	**Pol**
$x{:} \oplus \{\ell{:}\mathtt{A}\}_{\ell \in L}$	$x{:}\mathtt{A}_k$	$Rx.k; \mathsf{P}$	P	provider sends label k along x	+
		$\mathbf{case} L\, x(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell \in L}$	Q_k	client receives label k along x	
$x{:}\&\{\ell{:}\mathtt{A}\}_{\ell \in L}$	$x{:}\mathtt{A}_k$	$\mathbf{case} R\, x(\ell \Rightarrow \mathsf{P}_\ell)_{\ell \in L}$	P_k	provider receives label k along x	-
		$Lx.k\ \mathsf{Q}$	Q	client sends label k along x	
$x : 1$	-	$\mathbf{close}\, Rx$	-	provider sends "close" along x and terminates	+
		$\mathbf{wait}\, Lx; \mathsf{Q}$	Q	provider receives "close" along x	

TABLE 5.1: Overview of intuitionistic linear session types with their operational meaning.

5.3 Typing rules

The process typing rules are based on the *sequent calculus* given in Section 2.2.1, which leads to a left and a right rule for each connective. The left and right rules for each connective describe the interaction from the point of view of the provider and client, respectively.

Internal ($\oplus$) and external ($\&$) choice are the branching constructs. An internal choice gives the choice to the provider, an external choice to the client.

$$\frac{\bar{x} : \omega \vdash P :: (y : A_k) \quad (k \in L)}{\bar{x} : \omega \vdash Ry.k; P :: (y : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\forall \ell \in L \quad x : A_\ell \vdash P_\ell :: (y : C)}{x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{case}\, Lx\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (y : C)}\ \oplus L$$

$$\frac{\bar{x} : \omega \vdash P_\ell :: (y : A_\ell) \quad \forall \ell \in L}{\bar{x} : \omega \vdash \mathbf{case}\, Ry\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (y : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\frac{k \in L \quad x : A_k \vdash P :: (y : C)}{x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash Lx.k; P :: (y : C}\ \& L$$

The multiplicative unit (1) denotes process termination.

$$\frac{}{\cdot \vdash \mathbf{close}\, Ry :: (y : 1)}\ 1R$$

$$\frac{. \vdash Q :: (y : C)}{x : 1 \vdash \mathbf{wait}\, Lx; Q :: (y : C)}\ 1L$$

Identity and cut are the two rules that do not result in any communication. Identity amounts to termination after identifying the involved channels and cut to process spawning. The process executing $(w \leftarrow P_w; Q_w)$ spawns a new process P_w and continues as Q_w. To ensure uniqueness of channels, we need w to be a fresh channel.

$$\frac{}{x : A \vdash y \leftarrow x :: (y : A)}\ \textsc{Id}$$

$$\frac{\bar{x} : \omega \vdash P_w :: (w : A) \quad w : A \vdash Q_w :: (y : C)}{\bar{x} : \omega \vdash (w \leftarrow P_w; Q_w) :: (y : C)}\ \textsc{Cut}^w$$

Since the system is entirely syntax-directed we may sometimes equate (well-typed) programs with their typing derivations.

5.4 Recursive types

In this section, we extend subsingleton session types with *recursive* types, allowing them to capture unbounded interactions. We differentiate between least and greatest fixed points to maintain a Curry-Howard correspondence between recursive session-typed processes and infinitary subsingleton logic presented in Section 3.3.

Definition 5.2. We extend the grammar of session types to include least and greatest fixed points:

$$A ::= \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid 1 \mid \bot \mid t$$

where t ranges over type variables whose definition is given in a signature Σ:

$$\Sigma ::= \cdot \mid \Sigma, t =^i_\mu A \mid \Sigma, t =^i_\nu A,$$

with the conditions that

- if $t =^i_a A \in \Sigma$ and $t' =^i_b B \in \Sigma$, then $a = b$, and
- if $t =^i_a A \in \Sigma$ and $t =^j_b B \in \Sigma$, then $i = j$ and $A = B$.

For a fixed point t defined as $t =^i_a A$ in Σ the subscript a is the *polarity* of t: if $a = \mu$, then t is a fixed point with *positive* polarity and if $a = \nu$, then it is of *negative* polarity. Finitely representable least fixed points (e.g., natural numbers and lists) can be represented in this system as defined propositional variables with positive polarity, while the potentially infinite greatest fixed points (e.g., streams and infinite depth trees) are represented as those with negative polarity.

As a first programming-related example, consider natural numbers in unary form (nat) and a type to demand access to a number if desired (ctrl).

Example 5.1 (Natural numbers on demand).

$$\begin{aligned}\mathsf{nat} &=^2_\mu \oplus\{z : 1, s : \mathsf{nat}\}\\ \mathsf{ctrl} &=^1_\nu \&\{now : \mathsf{nat},\ \ notyet : \mathsf{ctrl}\}\end{aligned}$$

*In this example, Σ consists of an inductive and a coinductive type; these are, respectively: (i) the type of natural numbers (*nat*) built using two constructors for $zero$ and $successor$, and (ii) a type* *to demand access to a number if desired* (ctrl) *defined using two destructors for* now *to request the number and* $notyet$ *to send a postpone message. With* ctrl *being a negative fixed point, the request for the number can be postponed indefinitely. To define* nat *nested in* ctrl, *we associate* 2 *and* 1 *as priorities of* nat *and* ctrl, *respectively (*"ctrl has higher priority than nat"*).*

Example 5.2 (Binary numbers in standard form)**.** *As another example consider the signature with two types with positive polarity and the same priority:* std *and* pos. *Here,* std *is the type of standard bit strings, i.e., bit strings terminated with* $\$$ *without any leading* 0 *bits, and* pos *is the type of positive standard bit strings, i.e., all standard bit strings except* $\$$. *Note that in our representation the least significant bit is sent first.*

$$\begin{aligned} \mathsf{std} &=^1_\mu \oplus\{b0 : \mathsf{pos}, b1 : \mathsf{std}, \$: 1\} \\ \mathsf{pos} &=^1_\mu \oplus\{b0 : \mathsf{pos}, b1 : \mathsf{std}\} \end{aligned}$$

Example 5.3 (Bits and cobits)**.**

$$\begin{aligned} \mathsf{bits} &=^1_\mu \oplus\{b0 : \mathsf{bits},\ b1 : \mathsf{bits}\} \\ \mathsf{cobits} &=^2_\nu \&\{b0 : \mathsf{cobits},\ b1 : \mathsf{cobits}\} \end{aligned}$$

In a functional language, the type cobits *would be a greatest fixed point (an infinite stream of bits), while* bits *is recognized as an empty type. However, in the session type system, we treat them in a symmetric way.* bits *is an infinite sequence of bits with positive polarity. And its dual type,* cobits, *is an infinite stream of bits with negative polarity.*

We treat fixed points in an isorecursive way, that is, a message is sent to unfold the definition of a fixed point t. This message is written as μ_t for a least fixed point and ν_t for a greatest fixed point. The language of process expressions dealing with fixed points and their operational readings is given in Table 5.2.

The typing rules for processes that receive or send fixed point messages is based on the fixed point rules presented in the sequent calculus of Section 3.3. A least fixed point receives from the left (client) and send to the right (provider), while the negative one sends to the left (client) and receive from the right (provider).

$$\frac{\bar{x} : \omega \vdash P_y :: (y : A) \quad t =^i_\mu A}{\bar{x} : \omega \vdash Ry.\mu_t; P_y :: (y : t)}\ \mu R$$

$$\frac{x : A \vdash Q_x :: (y : C) \quad t =^i_\mu A}{x : t \vdash \mathbf{case}\, Lx\ (\mu_t \Rightarrow Q_x) :: (y : C)}\ \mu L$$

$$\frac{\bar{x} : \omega \vdash P_y :: (y : A) \quad t =^i_\nu A}{\bar{x} : \omega \vdash \mathbf{case}\, Ry\ (\nu_t \Rightarrow P_y) :: (y : t)}\ \nu R$$

$$\frac{x : A \vdash Q_x :: (y : C) \quad t =^i_\nu A}{x : t \vdash Lx.\nu_t; Q_x :: (y : C)}\ \nu L$$

Session type (curr./cont.)		**Process term (curr./cont.)**		**Description**	**Pol**
$x{:}\mathtt{t}$	$x{:}\mathtt{A}$	$Rx.\mu_{\mathtt{t}};\mathsf{P}$	P	provider sends unfolding message $\mu_{\mathtt{t}}$ along x	+
$(\mathtt{t} =^i_\mu \mathtt{A} \in \mathbf{\Sigma})$		$\mathbf{case}L\,x(\mu_{\mathtt{t}} \Rightarrow \mathsf{Q})$	Q	client receives unfolding message $\mu_{\mathtt{t}}$ along x	
$x{:}\mathtt{t}$	$x{:}\mathtt{A}$	$\mathbf{case}R\,x(\nu_{\mathtt{t}} \Rightarrow \mathsf{P})$	P	provider receives unfolding message $\nu_{\mathtt{t}}$ along x	-
$(\mathtt{t} =^i_\nu \mathtt{A} \in \mathbf{\Sigma})$		$Lx.\nu_{\mathtt{t}}\ \mathsf{Q}$	Q	client sends unfolding message $\nu_{\mathtt{t}}$ along x	

Table 5.2: Intuitionistic linear fixed point session types with their operational meaning.

The above fixed point rules are not enough to capture recursive processes. Recall from Chapter 3 that in the logic with similar fixed point rules, we need to allow infinitary derivations to get the full power of fixed points in inductive and coinductive reasoning. In the context of processes, we follow a similar approach by introducing process variables $X, Y, \cdots$ to the syntax of processes. Process definitions are of the form $\bar{x} : \omega \vdash X = P_{\bar{x},\bar{y}} :: (y : C)$ representing that variable X is defined as process P.

$$\frac{\bar{x} : \omega \vdash P_{\bar{x},\bar{y}} :: (y : C) \quad \bar{u} : \omega \vdash X = P_{\bar{u},\bar{w}} :: (w : C) \in V}{\bar{x} : \omega \vdash \bar{y} \leftarrow X \leftarrow \bar{x} :: (y : C)}\ \textsc{Def}(X)$$

A *program* $\mathcal{P}$ is defined as a pair $\langle V, S\rangle$, where V is a finite set of process definitions, and S is the *main* process variable.

These typing rules interpret *pre-proofs*: a circular derivation is represented as a collection of mutually recursive process definitions in $\mathcal{P} = \langle V, S\rangle$, with S referring to the root of the derivation.

As can be seen in the rule Def, the typing rules inherit the infinitary nature of deductions from the logical rules in Section 3.3 and are therefore not directly useful for type checking. The rule Def corresponds to forming cycles in the circular derivations of the system of Figure 3.1. We obtain a finitary system to check *circular* pre-proofs by removing the first premise from the Def rule and checking each process definition in V separately, under the hypothesis that all process definitions are well-typed.

$$\frac{\bar{u} : \omega \vdash X = P_{\bar{u},\bar{w}} :: (w : C) \in V}{\bar{x} : \omega \vdash \bar{y} \leftarrow X \leftarrow \bar{x} :: (y : C)}\ \textsc{Def}_f(X)$$

Example 5.4. *With signature*

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}$$

we define process Copy

$$x : \mathsf{nat} \vdash \mathsf{Copy} :: (y : \mathsf{nat})$$

as

```
$y \leftarrow \texttt{Copy} \leftarrow x =$
    $\textbf{case}\ Lx\ (\mu_{nat} \Rightarrow$          % receive $\mu_{nat}$ from left      (i)
        $\textbf{case}\ Lx$                         % receive a label from left    (ii)
            $(\ z \Rightarrow Ry.\mu_{nat};$               % send $\mu_{nat}$ to right         (ii-a)
                    $Ry.z;$                 % send label $z$ to right
                    $\textbf{wait}\ Lx;$             % wait for $x$ to close
                      $\textbf{close}\ Ry$          % close $y$
            $\mid s \Rightarrow Ry.\mu_{nat};$               % send $\mu_{nat}$ to right         (ii-b)
                    $Ry.s;$                 % send label $s$ to right
                    $y \leftarrow \texttt{Copy} \leftarrow x))$      % recursive call
```

This is an example of a recursive process over the signature Σ_1. The computational content of Copy *is to simply copy a natural number given from the left to the right:*
(i) It waits until it receives a positive fixed point unfolding message from the left, (ii) waits for another message from the left to determine the path it will continue with:
(a) If the message is a z label, (ii-a) the program sends a positive fixed point unfolding message to the right, followed by the label z, and then waits until a closing message is received from the left. Upon receiving that message, it closes the right channel.
(b) If the message is an s label, (ii-b) the program sends a positive fixed point unfolding message to the right, followed by the label s, and then calls itself and loops back to (i).

5.5 Operational semantics

The operational semantics for process expressions under the proofs-as-programs interpretation of linear logic has been treated exhaustively elsewhere [15, 16, 46, 97]. We therefore only briefly sketch the operational semantics here.

5.5.1 Configuration typing

A configuration $\mathcal{C}$ is a list of processes that communicate with each other along their private channels. It is defined with the grammar $\mathcal{C} ::= \cdot \mid \textbf{proc}(x, P) \mid (\mathcal{C}_1 \mid_{x:A} \mathcal{C}_2)$, where $\mid$ is an associative, noncommutative operator and $(\cdot)$ is the unit. The typing judgment for a configuration is of the form $\bar{x} : \omega \Vdash \mathcal{C} :: (y : B)$. We call $\bar{x}$ and y the external channels of configuration $\mathcal{C}$. The type checking rules for configurations are:

$$\frac{}{x : A \Vdash \cdot :: (x : A)}\ \textsc{Emp}$$

$$\frac{\bar{x} : \omega \Vdash \mathcal{C}_1 :: (z : A) \quad z : A \Vdash \mathcal{C}_2 :: (y : B)}{\bar{x} : \omega \Vdash \mathcal{C}_1 |_{z:A} \, \mathcal{C}_2 :: (y : B)} \ \textsc{comp}$$

$$\frac{\bar{x} : \omega \vdash P :: (y : B)}{\bar{x} : \omega \Vdash \mathbf{proc}(y, P) :: (y : B)} \ \textsc{proc}$$

In COMP we introduce a fresh channel z internal to the composition of $\mathcal{C}_1$ and $\mathcal{C}_2$.

A given *program* $\mathcal{P} = \langle V, S \rangle$ is well-typed, if for every process definition $\bar{x} : \omega \vdash X = P :: (y : B)$, we have:

$$\bar{x} : \omega \Vdash \mathbf{proc}(y, P) :: (y : B)$$

5.5.2 Synchronous semantics

In a *synchronous* computation, both sender and receiver block until they synchronize. A significant difference to much prior work is that we treat types in an isorecursive way, that is, a message is sent to unfold the definition of a type t. This message is written as μ_t for a least fixed point and ν_t for a greatest fixed point.

The computational semantics is defined on a configuration $\mathcal{C}$. The transitions given in Figure 5.1 can be applied anywhere in a configuration. The forward rule removes process $y \leftarrow x$ from the configuration and replaces channel x in the rest of the configuration with channel y. The rule for $x \leftarrow P\,; Q$ spawns process $[z/x]P$ and continues as $[z/x]Q$. To ensure uniqueness of channels, we need z to be a fresh channel. For internal choice, $Rx.k; P$ sends label k along channel x to the process on its right and continues as P. The process on the right, **case** $Lx\ (\ell \Rightarrow Q_\ell)$, receives the label k sent from the left along channel x, and chooses the kth alternative Q_k to continue with accordingly. The last transition rule unfolds the definition of a process variable X while instantiating the left and right channels $\bar{u}$ and w in the process definition with proper channel names, $\bar{x}$ and y respectively.

5.5.3 Asynchronous semantics

In this section we define an *asynchronous* dynamics for subsingleton logic. Asynchronous communication is a more practical model of computation and we will see in Section 5.7 that it allows a more realistic statement of a strong progress property. In an asynchronous semantics only receivers can be blocked, while senders output the message and proceed with their continuation.

The grammar and typing rule of configurations are extended to allow appearance of such outputted messages as follows:

$$\mathcal{C} ::= \cdot \mid \mathbf{msg}(M) \mid \mathbf{proc}(x, P) \mid (\mathcal{C}_1 \ |_{x:A} \ \mathcal{C}_2),$$

$$\frac{\bar{x} : \omega \vdash M :: (y : B)}{\bar{x} : \omega \Vdash \mathbf{msg}(M) :: (y : B)} \ \textsc{msg}$$

$$
\begin{array}{lcl}
\mathcal{C} \mid_x \mathbf{proc}(y, y \leftarrow x) \mid_y \mathcal{C}' & \mapsto & [y/x]\mathcal{C} \mid_y \mathcal{C}' \\
 & & \hfill \text{forward} \\
\mathbf{proc}(w, x \leftarrow P \,;\, Q) & \mapsto & \mathbf{proc}(z, [z/x]P) \mid_z \mathbf{proc}(w, [z/x]Q) \\
 & & \hfill (z \text{ fresh}), \text{ spawn} \\
\mathbf{proc}(x, \mathbf{close}\, Rx) \mid_x \mathbf{proc}(w, \mathbf{wait}\, Lx \,;\, Q) & \mapsto & \mathbf{proc}(w, Q) \\
 & & \hfill \text{close channel} \\
\mathbf{proc}(x, Rx.k \,;\, P) \mid_x \mathbf{proc}(w, \mathbf{case}\, Lx\, (\ell \Rightarrow Q_\ell)_{\ell \in L}) & \mapsto & \mathbf{proc}(x, P) \mid_x \mathbf{proc}(w, Q_k) \\
 & & \hfill \text{send label } k \in L \text{ right} \\
\mathbf{proc}(x, \mathbf{case}\, Rx\, (\ell \Rightarrow P_\ell)_{\ell \in L}) \mid_x \mathbf{proc}(w, Lx.k \,;\, Q) & \mapsto & \mathbf{proc}(x, P_k) \mid_x \mathbf{proc}(w, Q) \\
 & & \hfill \text{send label } k \in L \text{ left} \\
\mathbf{proc}(x, Rx.\mu_t \,;\, P) \mid_x \mathbf{proc}(w, \mathbf{case}\, Lx\, (\mu_t \Rightarrow Q)) & \mapsto & \mathbf{proc}(x, P) \mid_x \mathbf{proc}(w, Q) \\
 & & \hfill \text{send } \mu_t \text{ unfolding message right} \\
\mathbf{proc}(x, \mathbf{case}\, Rx\, (\nu_t \Rightarrow P)) \mid_x \mathbf{proc}(w, Lx.\nu_t \,;\, Q) & \mapsto & \mathbf{proc}(x, P) \mid_x \mathbf{proc}(w, Q) \\
 & & \hfill \text{send } \nu_t \text{ unfolding message left} \\
\mathbf{proc}(x, y \leftarrow X \leftarrow \bar{x}) & \mapsto & \mathbf{proc}(x, [y/w, \bar{x}/\bar{u}]P) \\
 & & \hfill \text{where } \bar{u} : \omega \vdash X = P :: (w : C)
\end{array}
$$

Figure 5.1: Synchronous computational semantics

where M is a special process defined with the grammar

$$M ::= Lx.k; w \leftarrow x \mid Rx.k; x \leftarrow w \mid Lx.\mu_t; w \leftarrow x \mid Rx.\nu_t; x \leftarrow w.$$

We model messages as special processes that contain the value of a particular message followed by a forwarding [9, 21, 30, 38]. The forwarding is necessary to ensure that an outputted message is properly sequenced with the sender's continuation.

The asynchronous dynamics is given in Figure 5.2. It is defined in terms of rewriting rules that can be applied anywhere in the configuration. A fresh channel is allocated whenever a new message is spawned, except for closing channels because there is no continuation. The forward then links the fresh channel and the previous one.

5.6 Type safety

In this section we present the usual preservation and progress theorems. The preservation theorem ensures types of a configuration are preserved during computation in both synchronous and asynchronous semantics [31]. For simplicity, we only present preservation for a closed configuration, i.e. a configuration that does not use any resources.

Theorem 5.3. *(Preservation) For a configuration* $\cdot \Vdash \mathcal{C} :: (y : A)$*, if* $\mathcal{C} \mapsto \mathcal{C}'$ *by one step of computation, then* $\cdot \Vdash \mathcal{C}' : (y : A)$.

Proof. The proof is by considering cases of $\mapsto$ and inversion on the typing derivation. □

$$
\begin{array}{lcl}
\mathcal{C} \mid_x \mathbf{proc}(y, y \leftarrow x) \mid_y \mathcal{C}' & \mapsto & [z/x]\mathcal{C} \mid_y \mathcal{C}' \\
 & & \hfill \text{forward} \\
\mathbf{proc}(w, x \leftarrow P \,;\, Q) & \mapsto & \mathbf{proc}(z, [z/x]P) \mid_z \mathbf{proc}(w, [z/x]Q) \\
 & & \hfill (z \text{ fresh}), \text{ spawn} \\
\mathbf{proc}(x, \mathbf{close}\, Rx) & \mapsto & \mathbf{msg}(\mathbf{close}\, Rx) \\
 & & \hfill \text{send close message} \\
\mathbf{msg}(\mathbf{close}\, Rx) \mid_x \mathbf{proc}(w, \mathbf{wait} Lx; P) & \mapsto & \mathbf{proc}(w, P) \\
 & & \hfill \text{close channel} \\
\mathbf{proc}(x, Rx.k; P) & \mapsto & \mathbf{proc}(z, [z/x]P) \mid_z \mathbf{msg}(Rx.k; z \leftarrow x)) \\
 & & \hfill \text{send label } k \in L \text{ right} \\
\mathbf{msg}(Rx.k; x \leftarrow z) \mid_x \mathbf{proc}(w, \mathbf{case} Lx(\ell \Rightarrow P_\ell)_{\ell \in L}) & \mapsto & \mathbf{proc}(w, [z/x]P_k) \\
 & & \hfill \text{receive label } k \in L \text{ from left} \\
\mathbf{proc}(w, Lx.k; P) & \mapsto & \mathbf{msg}(Lx.k; z \leftarrow x) \mid_z \mathbf{proc}(w, [z/x]P) \\
 & & \hfill \text{send label } k \in L \text{ to left} \\
\mathbf{proc}(x, \mathbf{case}\, Rx(\ell \Rightarrow P_\ell)_{\ell \in L}) \mid_x \mathbf{msg}(Lx.k; z \leftarrow x) & \mapsto & \mathbf{proc}(x, [z/x]P_k) \\
 & & \hfill \text{receive label } k \in L \text{ from right} \\
\mathbf{proc}(x, Rx.\mu_t; P) & \mapsto & \mathbf{proc}(w, [w/x]P) \mid_w \mathbf{msg}(Rx.\mu_t; x \leftarrow w) \\
 & & \hfill \text{send } \mu_t \text{ unfolding message right} \\
\mathbf{msg}(Rx.\mu_t; x \leftarrow z) \mid_x \mathbf{proc}(w, \mathbf{case}\, Lx(\mu_t \Rightarrow P)) & \mapsto & \mathbf{proc}(w, [z/x]P) \\
 & & \hfill \text{receive } \mu_t \text{ unfolding message from left} \\
\mathbf{proc}(w, Lx.\nu_t; P) & \mapsto & \mathbf{msg}(Lx.\nu_t; z \leftarrow x) \mid_z \mathbf{proc}(w, [z/x]P) \\
 & & \hfill \text{send } \nu_t \text{ unfolding message left} \\
\mathbf{proc}(x, \mathbf{case}\, Rx(\nu_t \Rightarrow P)) \mid_x \mathbf{msg}(Lx.\nu_t; z \leftarrow x) & \mapsto & \mathbf{proc}(z, [z/x]P) \\
 & & \hfill \text{receive } \nu_t \text{ unfolding message from right} \\
\mathbf{proc}(x, y \leftarrow X \leftarrow \bar{x}) & \mapsto & \mathbf{proc}(x, [y/w, \bar{x}/\bar{u}]P) \\
 & & \hfill \text{where } \bar{u} : \omega \vdash X = P :: (w : C)
\end{array}
$$

FIGURE 5.2: Asynchronous computational semantics

The progress property as stated below ensures that computation makes progress or it attempts to communicate with an external process [73].

Theorem 5.4. *(Progress) If $\bar{x} : \omega \Vdash \mathcal{C} :: (y : A)$, then either*

1. *$\mathcal{C}$ can make a transition,*
2. *or $\mathcal{C} = (\cdot)$ is empty,*
3. *or $\mathcal{C}$ cannot make a transition and attempts to communicate either to the left or to the right; in a synchronous semantics it has one of the following forms:*

$$
\begin{array}{ll}
\mathbf{proc}(w, \mathbf{case}\, Lx(\mu_t \Rightarrow P)) \mid_w \mathcal{C}' & \mathbf{proc}(w, \mathbf{case}\, Lx(\ell \Rightarrow P)_{\ell \in L}) \mid_w \mathcal{C}' \\
\mathbf{proc}(w, Lx.\mu_t; P) \mid_w \mathcal{C}' & \mathbf{proc}(w, Lx.k; P) \mid_w \mathcal{C}' \\
\mathcal{C}' \mid_w \mathbf{proc}(x, \mathbf{case}\, Rx(\nu_t \Rightarrow P)) & \mathcal{C}' \mid_w \mathbf{proc}(x, \mathbf{case}\, Rx(\ell \Rightarrow P)_{\ell \in L}) \\
\mathcal{C}' \mid_w \mathbf{proc}(x, Rx.\nu_t; P) & \mathcal{C}' \mid_w \mathbf{proc}(x, Rx.k; P)
\end{array}
$$

where $\mathcal{C}'$ cannot make any transitions.

Proof. The proof is by structural induction on the configuration typing from right to left. □

5.7 Strong progress

The progress property ensures that a configuration can always take a step or terminates. In the presence of (mutual) recursion, it is not strong enough to ensure the termination of a configuration either in an empty configuration or one attempting to communicate with its external channels. As a result, a well-typed configuration may fall into an infinite inner communication loop and never communicate with its external channels. This section defines a stronger form of progress that prevents such non-terminating behavior.

Example 5.5. *Take the signature*

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}.$$

We define a process

$$\cdot \vdash \mathtt{Loop} :: (y : \mathsf{nat}),$$

where Loop *is defined as*

$$\begin{array}{llr}
y \leftarrow \mathtt{Loop} \leftarrow \cdot = Ry.\mu_{nat}; & \textit{\% send } \mu_{nat} \textit{ to right} & \text{(i)}\\
\qquad Ry.s; & \textit{\% send label } s \textit{ to right} & \text{(ii)}\\
\qquad y \leftarrow \mathtt{Loop} \leftarrow \cdot & \textit{\% recursive call} & \text{(iii)}
\end{array}$$

$\mathcal{P}_1 := \langle\{\mathtt{Loop}\}, \mathtt{Loop}\rangle$ *forms a program over the signature* Σ_1*. It (i) sends a positive fixed point unfolding message to the right, (ii) sends the label* s*, as another message corresponding to* successor*, to the right, (iii) calls itself and loops back to (i).*

In an asynchronous semantics, Loop runs forever, sending an infinite stream of *successor* labels to the right, without receiving any messages from the left or the right. In the synchronous semantics, the process is blocked before each send by waiting for another process willing to receive. Even in the synchronous semantics Loop has a non-terminating nature: we will see that composing $\cdot \vdash \mathtt{Loop} :: (y : \mathsf{nat})$ with process $y : \mathsf{nat} \vdash \mathtt{Block} :: (z : 1)$, defined in the next example, results in exchanging an infinite number of messages between them.

Example 5.6. *Define process*

$$y : \mathsf{nat} \vdash \mathtt{Block} :: (z : 1)$$

over the signature Σ_1 as

```
$z \leftarrow \texttt{Block} \leftarrow y =$
    $\textbf{case}\ Ly\ (\mu_{nat} \Rightarrow$                 % receive $\mu_{nat}$ from left     (i)
        $\textbf{case}\ Ly$                                 % receive a label from left     (ii)
            $(\ z \Rightarrow \textbf{wait}\ Ly;$                   % wait for $y$ to close        (ii-a)
                    $\textbf{close}\ Rz$                        % close $z$
             $\mid s \Rightarrow z \leftarrow \texttt{Block} \leftarrow y))$       % recursive call              (ii-b)
```

$\mathcal{P}_2 := \langle\{\texttt{Block}\}, \texttt{Block}\rangle$ forms a program over the signature Σ_1:
(i) `Block` **waits***, until it receives a positive fixed point unfolding message from the left, (ii) waits for another message from the left to determine the path it will continue with:*
(a) If the message is a z label, (ii-a) the program waits until a closing message is received from the left. Upon receiving that message, it closes the left and then the right channel.
(b) If the message is an s label, (ii-b) the program calls itself and loops back to (i).

The infinite computation of the composition $\cdot \Vdash y \leftarrow \texttt{Loop} \mid_y z \leftarrow \texttt{Block} \leftarrow y :: (z : 1)$ in the synchronous semantics can be depicted as follows:

$$\begin{array}{ll}
y \leftarrow \texttt{Loop} \mid_y z \leftarrow \texttt{Block} \leftarrow y & \mapsto \\
Ry.\mu_{\mathsf{nat}}; Ry.s; y \leftarrow \texttt{Loop} \mid_y z \leftarrow \texttt{Block} \leftarrow y & \mapsto \\
Ry.\mu_{\mathsf{nat}}; Ry.s; y \leftarrow \texttt{Loop} \mid_y \textbf{case}\ Ly\ (\mu_{\mathsf{nat}} \Rightarrow \textbf{case}\ Ly\ \cdots) & \mapsto \\
Ry.s; y \leftarrow \texttt{Loop} \mid_y \textbf{case}\ Ly\ (s \Rightarrow z \leftarrow \texttt{Block} \leftarrow y \mid z \Rightarrow \textbf{wait}\ Ly; \textbf{close}\ Rz) & \mapsto \\
y \leftarrow \texttt{Loop} \mid_y z \leftarrow \texttt{Block} \leftarrow y & \mapsto \\
\ldots &
\end{array}$$

In this computation, the configuration can always take a step, but it does not communicate to the left or right and a never ending series of internal communications takes place. To avoid such infinitary computations, we define strong progress as follows.

Definition 5.5. (Strong Progress) Configuration $\bar{x} : \omega \Vdash \mathcal{C} :: (y : A)$ satisfies the strong progress property if after finite number of steps, either

1. $\mathcal{C} = (\cdot)$ is empty,
2. or $\mathcal{C}$ is blocked by waiting to communicate to the left or right.

The definition of strong progress in the asynchronous setting is more realistic. In the asynchronous semantics, only receive can block a configuration; a process that keeps sending unfolding messages, e.g. Loop, does not satisfy strong progress. In the synchronous semantics, a process can be blocked by both send and receive. Therefore process Loop satisfies strong progress, even though it clearly sends infinitely many messages when composed with a processes willing to receive, e.g. `Block`.

Clearly, not all well-typed programs satisfy strong progress. We devote the rest of this thesis to proving the strong progress property for a subset of well-typed programs that can be identified algorithmically.

Chapter 6

Strong progress as termination of cut elimination

In Chapter 5 we showed that by defining type variables in the signature and process variables in the program, we can generate (mutually) recursive processes which correspond to circular pre-proofs in the sequent calculus.

In Section 5.5, we introduced process configuration $\mathcal{C}$ as a list of processes connected by the associative, noncommutative parallel composition operator $|_x$. Alternatively, considering $\mathcal{C}_1$ and $\mathcal{C}_2$ as two processes, configuration $\mathcal{C}_1 \mid_z \mathcal{C}_2$ can be read as their composition by a cut rule $(z \leftarrow \mathcal{C}_1; \mathcal{C}_2)$. In Section 5.5, we defined a synchronous operational semantics on configurations using transition rules. Similarly, these computational transitions can be interpreted as the internal cut reductions in the infinitary calculus of subsingleton logic with fixed points. For example, for configuration

$$\mathcal{C} = \mathcal{C}_1 \mid_z \mathbf{proc}(x, Rx.\mu_t; P) \mid_x \mathbf{proc}(w, \mathbf{case}\, Lx\, (\mu_t \Rightarrow Q)) \mid_w \mathcal{C}_2$$

the internal communication transition

$$\begin{array}{l} \mathcal{C}_1 \mid_z \mathbf{proc}(x, Rx.\mu_t; P) \mid_x \mathbf{proc}(w, \mathbf{case}\, Lx\, (\mu_t \Rightarrow Q)) \mid_w \mathcal{C}_2 \quad \mapsto \\ \mathcal{C}_1 \mid_z \mathbf{proc}(x, P) \mid_x \mathbf{proc}(w, Q) \mid_w \mathcal{C}_2 \end{array}$$

can be interpreted as the following cut reduction step:

$$\dfrac{\mathcal{C}_1 \quad \dfrac{z : A \vdash P :: (x : B) \quad t =_\mu B}{z : A \vdash Rx.\mu_t; P :: (x : t)}\,\mu R \quad \dfrac{x : A \vdash Q :: (w : C) \quad t =_\mu B}{x : t \vdash \mathbf{case}\, Lx\, (\mu_t \Rightarrow Q) :: (w : C)}\,\mu L \quad \mathcal{C}_2}{\bar{x} : \omega \vdash \mathcal{C} :: (v : D)}\,nCut \quad \xRightarrow{\mathsf{PRd}}$$

$$\dfrac{\mathcal{C}_1 \quad z : A \vdash P :: (x : B) \quad x : B \vdash Q :: (w : C) \quad \mathcal{C}_2}{\bar{x} : \omega \vdash \mathcal{C} :: (v : D)}\,nCut$$

Recall from Section 3.3 that Fortier and Santocanale [36] introduced a cut-elimination algorithm for derivations in infinitary singleton logic with fixed points. As a part of their cut-elimination algorithm, they defined a function Treat that applies internal cut reductions on infinitary derivations. They proved that this function terminates on a list of pre-proofs fused by consecutive cuts if all of them satisfy their validity condition. In our system, the function Treat corresponds to computation on a configuration of processes. Termination of this function corresponds to the computation's termination either in an empty configuration or one attempting to communicate with an external channel, i.e. the strong progress property.

In this chapter, we introduce a type system as an algorithm to check a stricter version of Fortier and Santocanale's validity condition (FS validity condition) generalized to the subsingleton fragment. Our algorithm is local in the sense that we check the guard condition for each process definition separately, and it is stricter in the sense that it accepts a proper subset of the proofs recognized by the FS validity condition. Since our local guard condition implies the FS validity condition, we can use their results to show that a locally guarded program satisfies strong progress. The results of this chapter are built upon the correspondence between internal cut reductions and synchronous semantics of session types, and are confined to the synchronous semantics.

We develop a local guard condition through a sequence of refinements in Sections 6.2–6.5. We capture this condition on infinitary proofs in Section 6.6 and reduce it to a finitary algorithm in Section 6.7. We prove that our local guard condition implies Fortier and Santocanale's validity condition (Section 6.8) and therefore cut elimination. In Section 6.9 we explore the computational consequences of this, including the strong progress property, which states that every guarded configuration of processes will either be empty or attempt to communicate along external channels after a finite number of steps. We conclude by illustrating some limitations of our algorithm (Section 6.10) and pointing to some additional related and future work (Section 9).

A key aspect of our type system is that our guard condition is a compositional property (as we generally expect from type systems) so that the composition of guarded programs defined over the same signature are also guarded and therefore also satisfy strong progress. In other words, we identify a set of processes such that their corresponding derivations are not only closed under cut elimination, but also closed under cut introduction (i.e., strong progress is preserved when processes are joined by cut).

6.1 Ensuring communication and a local guard condition

In this section we motivate our algorithm as an effectively decidable compositional and local criterion which ensures that a program always terminates either in an empty configuration or one attempting to communicate along external channels.

Example 6.1. *Take the signature*

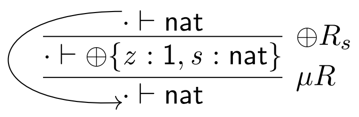
$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\},$$

and process

$$\cdot \vdash \texttt{Loop} :: (y : \mathsf{nat}),$$

$$\begin{array}{rlr}
y \leftarrow \texttt{Loop} \leftarrow \cdot = Ry.\mu_{nat}; & \%\ \textit{send } \mu_{nat} \textit{ to right} & \text{(i)} \\
Ry.s; & \%\ \textit{send label } s \textit{ to right} & \text{(ii)} \\
y \leftarrow \texttt{Loop} \leftarrow \cdot & \%\ \textit{recursive call} & \text{(iii)}
\end{array}$$

We can obtain the following infinite derivation in the system of subsingleton logic with fixed points via the Curry-Howard correspondence of the unique typing derivation of process Loop*:*


$\cdot \vdash \mathsf{nat}$
$\cdot \vdash \oplus\{z : 1, s : \mathsf{nat}\}$
$\oplus R_s$
μR
$\cdot \vdash \mathsf{nat}$

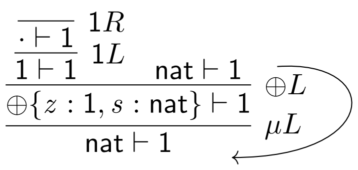
Process

$$x : \mathsf{nat} \vdash \texttt{Block} :: (y : 1)$$

over the signature Σ_1 *defined as*

$$\begin{array}{llr}
y \leftarrow \texttt{Block} \leftarrow x = & & \\
\quad \textbf{case } Lx\ (\mu_{nat} \Rightarrow & \%\ \textit{receive } \mu_{nat} \textit{ from left} & \text{(i)} \\
\quad\quad \textbf{case } Lx & \%\ \textit{receive a label from left} & \text{(ii)} \\
\quad\quad\quad (\ z \Rightarrow \textbf{wait } Lx; & \%\ \textit{wait and close } x & \text{(ii-a)} \\
\quad\quad\quad\quad \textbf{close } Ry & \%\ \textit{close } y & \\
\quad\quad\quad \mid s \Rightarrow y \leftarrow \texttt{Block} \leftarrow x)) & \%\ \textit{recursive call} & \text{(ii-b)}
\end{array}$$

corresponds to the following infinite derivation:



Derivations corresponding to both of these programs are cut-free. Also no internal loop takes place during their computation, in the sense that they both communicate with their left or right channels after finite number of steps. For process Loop this communication is restricted to sending infinitely many unfolding and successor messages to the right. Process Block, on the other hand, receives the same type of messages after finite number of steps as long as they are provided by a process on its left. Composing these two processes as in $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ results in an internal loop: process `Loop` keeps providing unfolding and successor messages for process `Block` so that they both can continue the computation and call themselves recursively. Because of this internal loop, the composition is not acceptable: it never communicates with its left (empty channel) or right (channel y). The infinite derivation corresponding to the composition $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ therefore should be rejected as unguarded:

$$
\dfrac{
\dfrac{\dfrac{\cdot \vdash \mathsf{nat}}{\cdot \vdash \oplus\{z : 1, s : \mathsf{nat}\}}\,\oplus R_s}{\cdot \vdash \mathsf{nat}}\,\mu R
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\,1R}{1 \vdash 1}\,1L \quad \mathsf{nat} \vdash 1}{\oplus\{z : 1, s : \mathsf{nat}\} \vdash 1}\,\oplus L}{\mathsf{nat} \vdash 1}\,\mu L
}{\cdot \vdash 1}\,\textsc{Cut}_{nat}
$$

The cut elimination algorithm introduced by Fortier and Santocanale, similar to the generalization of it that we introduced in Section 4.4, uses a reduction function TREAT and may never halt. They proved that for derivations satisfying the validity condition TREAT is locally terminating since it always halts on valid proofs [36]. The above derivation is an example of one that does not satisfy the FS validity condition and the cut elimination algorithm does not locally terminate on it.

Like cut elimination, strong progress is not compositional. Processes `Loop` and `Block` both satisfy the strong progress property but their composition $x \leftarrow$ `Loop` $\leftarrow \cdot \mid y \leftarrow$ `Block` $\leftarrow x$ does not. We will show in Section 6.9 that FS validity implies strong progress. But, in contrast to strong progress, FS validity is compositional in the sense that composition of two disjoint valid proofs is also valid. However, the FS validity condition is not local. Locality is particularly important from the programming point of view. It is the combination of two properties that are pervasive and often implicit in the study of programming languages. First, the algorithm is syntax-directed, following the structure of the program and second, it checks each process definition separately, requiring only the signature and the types of other processes but not their definition. One advantage of locality is asymptotic complexity, and, furthermore, a practically very efficient implementation. In Remark 6.19 we show that the time complexity of our guard algorithm is linear in the total input, which consists of the signature and the process definitions. Another is precision of error messages: locality implies that there is an exact program location where the condition is violated. The guard condition is a complex property, so the value of precise error messages cannot be overestimated. The final advantage is modularity: all we care about a process is its interface, not its definition, which means we can revise definitions individually without breaking the guard condition for the rest of the program as long as we respect their interface. Our goal is to construct a locally checkable guard condition that accepts (a subset of) programs satisfying strong progress and is compositional.

In functional programming languages a program is called *terminating* if it reduces to a value in a finite number of steps, and is called *productive* if every piece of the output is generated in finite number of steps (even if the program potentially runs forever). As in the current work, the theoretical underpinnings for terminating and productive programs are also least and greatest fixed points, respectively, but due to the functional nature of computation they take a different and less symmetric form than here (see, for example, [10, 45]).

Going back to Examples 5.5 and 5.6, process `Loop` seems less acceptable than process `Block`: process `Loop` does not receive any least or greatest fixed point unfolding messages. It is neither a terminating nor a productive process. We want our algorithm to accept process `Block` rather than `Loop`, since it cannot accept both. This motivates a definition of finite reactivity on session-typed processes.

Definition 6.1. A program defined over a signature Σ is *(finitely) reactive to the left* if for a positive fixed point $t \in \Sigma$ with priority i it does not continue forever without receiving a fixed point unfolding message μ_t from the left infinitely often. Moreover, for any negative fixed point $s \in \Sigma$ with priority $j < i$, the program does not send infinitely many ν_s messages to the left.
A program is *(finitely) reactive to the right* if for a negative fixed point $t \in \Sigma$ with priority i it does not continue forever without receiving a fixed point unfolding message ν_t from the right infinitely often. Moreover, for any positive fixed point $s \in \Sigma$ with priority $j < i$, the program does not send infinitely many μ_s messages to the right.
A program is called *(finitely) reactive* if it is either *reactive to the right* or *to the left*.

By this definition, process `Block` is reactive while process `Loop` is not. Finite reactivity corresponds to the FS validity condition on the underlying circular derivation of a process. Although reactivity is not local we use it as a motivation behind our algorithm. We construct our local guard condition one step at a time. In each step, we expand the condition to accept one more family of interesting finitely reactive programs, provided that we can check the condition locally. We first establish a local algorithm for programs with only direct recursion. We expand the algorithm further to support mutual recursions as well. Then we examine a subtlety regarding the cut rule to accept more programs locally. The reader may skip to Section 6.7 which provides our complete finitary algorithm. Later, in Sections 6.8 and 6.9 we prove that our algorithm ensures the FS validity condition and strong progress.

Priorities of type variables in a signature are central to ensure that a process defined based on them satisfies strong progress. Throughout the thesis we assume that the priorities are assigned (by a programmer) based on the intuition of why strong progress holds.

We conclude this section with an example of a reactive process `Copy`. This process, similar to `Block`, receives a natural number from the left but instead of consuming it, sends it over to the right along a channel of type `nat`.

Example 6.2. *With signature* $\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}$ *we define process* `Copy`

$$x : \mathsf{nat} \vdash \mathtt{Copy} :: (y : \mathsf{nat})$$

as

```
$y \leftarrow \texttt{Copy} \leftarrow x =$
    $\textbf{case}\ Lx\ (\mu_{nat} \Rightarrow$          % receive $\mu_{nat}$ from left      (i)
        $\textbf{case}\ Lx$                               % receive a label from left        (ii)
            $(\ z \Rightarrow Ry.\mu_{nat};$              % send $\mu_{nat}$ to right        (ii-a)
                    $Ry.z;$                               % send label $z$ to right
                    $\textbf{wait}\ Lx;$                  % wait for $x$ to close
                      $\textbf{close}\ Ry$                % close $y$
            $\mid s \Rightarrow Ry.\mu_{nat};$            % send $\mu_{nat}$ to right        (ii-b)
                    $Ry.s;$                               % send label $s$ to right
                    $y \leftarrow \texttt{Copy} \leftarrow x))$   % recursive call
```

This is an example of a recursive process, and forms a left reactive program over the signature Σ_1. Process Copy *does not involve spawning (its underlying derivation is cut-free) and satisfies the strong progress property. This property is preserved when composed with* Block *as $y \leftarrow \texttt{Copy} \leftarrow x \mid z \leftarrow \texttt{Block} \leftarrow y$.*

6.2 A local guard algorithm: naive version

In this section we develop a first naive version of our local guard algorithm using Examples 6.3-6.4.

Example 6.3. *Let the signature be*

$$\Sigma_2 := \mathsf{bits} =^1_\mu \oplus\{\mathit{b0} : \mathsf{bits},\ \mathit{b1} : \mathsf{bits}\}$$

and define the process BitNegate

$$x : \mathsf{bits} \vdash \texttt{BitNegate} :: (y : \mathsf{bits})$$

with

```
$y \leftarrow \texttt{BitNegate} \leftarrow x =$
    $\textbf{case}\ Lx\ (\mu_{bits} \Rightarrow$                      % receive $\mu_{bits}$ from left      (i)
        $\textbf{case}\ Lx$                                        % receive a label from left     (ii)
          $(\ b0 \Rightarrow Ry.\mu_{bits};$                        % send $\mu_{bits}$ to right          (ii-a)
                  $Ry.b1;$                                         % send label $b1$ to right
                  $y \leftarrow \texttt{BitNegate} \leftarrow x$   % recursive call
          $\mid b1 \Rightarrow Ry.\mu_{bits};$                      % send $\mu_{bits}$ to right          (ii-b)
                  $Ry.b0;$                                         % send label $b0$ to right
                  $y \leftarrow \texttt{BitNegate} \leftarrow x))$  % recursive call
```

$\mathcal{P}_4 := \langle\{\texttt{BitNegate}\}, \texttt{BitNegate}\rangle$ *forms a left reactive program over the signature* Σ_2 *quite similar to* `Copy`*. Computationally,* `BitNegate` *is a buffer with one bit capacity that receives a bit from the left and stores it until a process on its right asks for it. After that, the bit is negated and sent to the right and the buffer becomes free to receive another bit.*

Example 6.4. *Dual to Example 6.3, we can define* `coBitNegate`*. Let the signature be*

$$\Sigma_3 := \mathsf{cobits} =^1_\nu \&\{b0 : \mathsf{cobits},\ b1 : \mathsf{cobits}\}$$

with process

$$x : \mathsf{cobits} \vdash \texttt{coBitNegate} :: (y : \mathsf{cobits})$$

where `coBitNegate` *is defined as*

```
$y \leftarrow \texttt{coBitNegate} \leftarrow x =$
    $\textbf{case}\ Ry\ (\nu_{cobits} \Rightarrow$                      % receive $\nu_{cobits}$ from right     (i)
        $\textbf{case}\ Ry$                                          % receive a label from right    (ii)
          $(\ b0 \Rightarrow Lx.\nu_{cobits};$                        % send $\nu_{cobits}$ to left          (ii-a)
                  $Lx.b1;$                                           % send label $b1$ to left
                  $y \leftarrow \texttt{coBitNegate} \leftarrow x$   % recursive call
          $\mid b1 \Rightarrow Lx.\nu_{cobits};$                      % send $\nu_{cobits}$ to left          (ii-b)
                  $Lx.b0;$                                           % send label $b0$ to left
                  $y \leftarrow \texttt{coBitNegate} \leftarrow x))$  % recursive call
```

$\mathcal{P}_5 := \langle\{\texttt{coBitNegate}\}, \texttt{coBitNegate}\rangle$ *forms a right reactive program over the signature* Σ_3*. Computationally,* `coBitNegate` *is a buffer with one bit capacity. In contrast to* `BitNegate` *in Example 6.3, its types have negative polarity: it receives a bit from the right, and stores it until a process on its left asks for it. After that the bit is negated and sent to the left and the buffer becomes free to receive another bit.*

Remark 6.2. The property that assures the reactivity of the previous examples lies in their step (i) in which the program *blocks* until an unfolding message is received, i.e., the program can only continue the computation if it *receives* a message at step (i), and even after receiving the message it can only take finitely many steps further before another unfolding message is needed.

We first develop a naive version of our algorithm which captures the property explained in Remark 6.2: associate an initial integer value (say 0) with each channel and define the basic step of our algorithm to be *decreasing* the value associated to a channel *by one* whenever it *receives* a fixed point unfolding message. Also, for a reason that is explained later in Remark 6.3, whenever a channel *sends* a fixed point unfolding message its value is *increased by one*. Then at each recursive call, the value of the left and right channels are compared to their initial value.

For instance, in Example 6.2, in step (i) where the process receives a μ_{nat} message via the left channel (x), the value associated with x is decreased by one, while in steps (ii-a) and (ii-b) in which the process sends a μ_{nat} message via the right channel (y) the value associated with y is increased by one:

$$\begin{array}{lrr}
 & x & y \\
y \leftarrow \texttt{Copy} \leftarrow x = & 0 & 0 \\
\qquad\qquad \textbf{case}\, Lx\, (\mu_{nat} \Rightarrow & -1 & 0 \\
\qquad\qquad\qquad \textbf{case}\, Lx\, (z \Rightarrow Ry.\mu_{nat}; & -1 & 1 \\
\qquad\qquad\qquad\qquad\qquad R.z; \textbf{wait}\, Lx; \textbf{close}\, Ry & -1 & 1 \\
\qquad\qquad\qquad\qquad s \Rightarrow Ry.\mu_{nat}; & -1 & 1 \\
\qquad\qquad\qquad\qquad\qquad Ry.s; y \leftarrow \texttt{Copy} \leftarrow x)) & -1 & 1
\end{array}$$

When the recursive call occurs, channel x has the value $-1 < 0$, meaning that at some point in the computation it received a positive fixed point unfolding message. We can simply compare the value of the list $[x, y]$ lexicographically at the beginning and just before the recursive call: $[-1, 1]$ being less than $[0, 0]$ exactly captures the property observed in Remark 6.2 for the particular signature Σ_1. Note that by the definition of Σ_1, y never receives a fixed point unfolding message, so its value never decreases, and x never sends a fixed point unfolding message, thus its value never increases.

The same criteria works for the program $\mathcal{P}_3$ over the signature Σ_2 defined in Example 6.3, since Σ_2 also contains only one positive fixed point:

$$\begin{array}{lrr}
 & x & y \\
y \leftarrow \texttt{BitNegate} \leftarrow x = & 0 & 0 \\
\qquad\qquad\textbf{case}\ Lx\ (\mu_{bits} \Rightarrow & -1 & 0 \\
\qquad\qquad\qquad\textbf{case}\ Lx\ (b0 \Rightarrow Ry.\mu_{bits}; & -1 & 1 \\
\qquad\qquad\qquad\qquad Ry.b1; y \leftarrow \texttt{BitNegate} \leftarrow x & -1 & 1 \\
\qquad\qquad\qquad\quad b1 \Rightarrow Ry.\mu_{bits}; & -1 & 1 \\
\qquad\qquad\qquad\qquad Ry.b0; y \leftarrow \texttt{BitNegate} \leftarrow x)) & -1 & 1
\end{array}$$

At both recursive calls the value of the list $[x, y]$ is less than $[0, 0]$: $[-1, 1] < [0, 0]$.

However, for a program defined on a signature with a negative polarity such as the one defined in Example 6.4, this condition does not work:

$$\begin{array}{lrr}
 & x & y \\
y \leftarrow \texttt{coBitNegate} \leftarrow x = & 0 & 0 \\
\qquad\qquad\textbf{case}\ Ry\ (\nu_{cobits} \Rightarrow & 0 & -1 \\
\qquad\qquad\qquad\textbf{case}\ Ry\ (b0 \Rightarrow Lx.\nu_{cobits}; & 1 & -1 \\
\qquad\qquad\qquad\qquad Lx.b1; y \leftarrow \texttt{coBitNegate} \leftarrow x & 1 & -1 \\
\qquad\qquad\qquad\quad b1 \Rightarrow Lx.\nu_{cobits}; & 1 & -1 \\
\qquad\qquad\qquad\qquad Lx.b0; y \leftarrow \texttt{coBitNegate} \leftarrow x)) & 1 & -1
\end{array}$$

By the definition of Σ_3, y only receives unfolding fixed point messages, so its value only decreases. On the other hand, x cannot receive an unfolding fixed point from the left and thus its value never decreases. In this case the property in Remark 6.2 is captured by comparing the initial value of the list $[y, x]$, instead of $[x, y]$, with its value just before the recursive call: $[-1, 1] < [0, 0]$.

For a signature with only a single recursive type we can form a list by looking at the polarity of its type such that the value of the channel that receives the unfolding message comes first, and the value of the other one comes second. With this generalization, we can check all three programs that we have seen so far, `Copy`, `BitNegate`, and `coBitNegate`.

6.3 Priorities in the local guard algorithm

The property explained in Remark 6.2 of previous section is not strict enough, particularly when the signature has more than one recursive type. In that case not all programs that are waiting for a fixed point unfolding message before a recursive call are reactive.

Example 6.5. *Consider the signature*

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\quad s : nat\}\end{aligned}$$

astream *is a type with negative polarity of a potentially infinite stream where its* head *is always followed by an acknowledgement while* tail *is not.* ack *is a type with positive polarity that, upon unfolding, describes a protocol requiring an acknowledgment message to be sent to the right (or be received from the left).*

$\mathcal{P}_6 := \langle\{\texttt{Ping}, \texttt{Pong}, \texttt{PingPong}\}, \texttt{PingPong}\rangle$ *forms a program over the signature* Σ_4 *with the typing of its processes*

$$\begin{aligned}x : \mathsf{nat} &\vdash \texttt{Ping} :: (w : \mathsf{astream})\\ w : \mathsf{astream} &\vdash \texttt{Pong} :: (y : \mathsf{nat})\\ x : \mathsf{nat} &\vdash \texttt{PingPong} :: (y : \mathsf{nat})\end{aligned}$$

We define processes Ping, Pong, *and* PingPong *over* Σ_4 *as:*

```
$y \leftarrow \texttt{PingPong} \leftarrow x =$
    $w \leftarrow \texttt{Ping} \leftarrow x;$                      % spawn process Ping                    (i)
        $y \leftarrow \texttt{Pong} \leftarrow w$                  % continue with a tail call

$y \leftarrow \texttt{Pong} \leftarrow w =$
    $Lw.\nu_{astream};$                                             % send $\nu_{astream}$ to left          (ii-Pong)
      $Lw.head;$                                                    % send label head to left               (iii-Pong)
        case $Lw$ $(\mu_{ack} \Rightarrow$                          % receive $\mu_{ack}$ from left         (iv-Pong)
          case $Lw$ (                                               % receive a label from left
                $ack \Rightarrow Ry.\mu_{nat};$                     % send $\mu_{nat}$ to right
                  $Ry.s;$                                           % send label $s$ to right
                    $y \leftarrow \texttt{Pong} \leftarrow w))$     % recursive call

$w \leftarrow \texttt{Ping} \leftarrow x =$
    case $Rw$ $(\nu_{astream} \Rightarrow$                          % receive $\nu_{astream}$ from right    (ii-Ping)
      case $Rw$ (                                                   % receive a label from right
            $head \Rightarrow Rw.\mu_{ack};$                        % send $\mu_{ack}$ to right             (iii-Ping)
                  $Rw.ack;$                                         % send label ack to right
                    $w \leftarrow \texttt{Ping} \leftarrow x$       % recursive call
            $\mid tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x))$   % recursive call
```

(i) Program $\mathcal{P}_6$ starting from `PingPong`*, spawns a new process* `Ping` *and continues as* `Pong`*:*
(ii-`Pong`*) Process* `Pong` *sends an* astream *unfolding and then a $head$ message to the left, and then (iii-*`Pong`*) waits for an acknowledgment, i.e., ack, from the left.*
(ii-`Ping`*) At the same time process* `Ping` *waits for an* astream *fixed point unfolding message from the right, which becomes available after step (ii-*`Pong`*). Upon receiving the message, it waits to receive either $head$ or $tail$ from the right, which is also available from (ii-*`Pong`*) and is actually a $head$. So (iii-*`Ping`*) it continues with the path corresponding to $head$, and acknowledges receipt of the previous messages by sending an unfolding messages and the label ack to the right, and then it calls itself (ii-*`Ping`*).*
(iv-`Pong`*) Process* `Pong` *now receives the two messages sent at (iii-*`Ping`*) and thus can continue by sending a* nat *unfolding message and the label s to the right, and finally calling itself (ii-*`Pong`*).*
Although both recursive processes `Ping` *and* `Pong` *at some point wait for a fixed point unfolding message, this program runs infinitely without receiving any messages from the outside, and thus is not reactive.*

The back-and-forth exchange of fixed point unfolding messages between two processes in the previous example can arise when at least two mutually recursive types with different polarities are in the signature. This is why we need to incorporate priorities of the type variables into the guard algorithm.

Remark 6.3. In Example 6.5, for instance, waiting to *receive* an unfolding message $\nu_{astream}$ of priority 2 in line (ii-`Ping`) is not enough to ensure that the recursive call is guarded because later in line (iii-`Ping`) the process *sends* an unfolding message of a higher priority (1).

To prevent such a call from being guarded we form a list for each process. This list stores the information of the fixed point unfolding messages that the process received and sent before a recursive call for each type variable in their order of priority.

Example 6.6. *Consider the signature and program $\mathcal{P}_6$ as defined in Example 6.5. For the process $x : \mathsf{nat} \vdash w \leftarrow \texttt{Ping} \leftarrow x :: (w : \mathsf{astream})$ form the list*

$[\mathsf{ack} - received, \mathsf{ack} - sent, \mathsf{astream} - received, \mathsf{astream} - sent, \mathsf{nat} - received, \mathsf{nat} - sent].$

Types with positive polarity, i.e., ack *and* nat*, receive messages from the left channel (x) and send messages to the right channel (w), while those with negative polarity, i.e.,* astream*, receive from the right channel (w) and send to the left one (x). Thus, the above list can be rewritten as*

$$[x_{\mathsf{ack}}, w_{\mathsf{ack}}, w_{\mathsf{astream}}, x_{\mathsf{astream}}, x_{\mathsf{nat}}, w_{\mathsf{nat}}].$$

To keep track of the sent/received messages, we start with $[0, 0, 0, 0, 0, 0]$ as the value of the list, when the process $x : \mathsf{nat} \vdash \texttt{Ping} :: (w : \mathsf{astream})$ is first spawned. Then, similar to the first version of our algorithm, on the steps in which the process receives a fixed point unfolding message, the value of the corresponding element of the list is decreased by one. And on the steps it sends a fixed point unfolding message, the corresponding value is increased by one:

$$\begin{array}{lr}
w \leftarrow \texttt{Ping} \leftarrow x = & [0,0,0,0,0,0] \\
\quad \textbf{case}\, Rw\ (\nu_{astream} \Rightarrow & [0,0,-1,0,0,0] \\
\qquad \textbf{case}\, Rw\ (head \Rightarrow Rw.\mu_{ack}; & [0,1,-1,0,0,0] \\
\qquad\qquad Rw.ack; w \leftarrow \texttt{Ping} \leftarrow x & [0,1,-1,0,0,0] \\
\qquad\quad \mid tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x)) & [0,0,-1,0,0,0]
\end{array}$$

The two last lines are the values of the list on which process `Ping` *calls itself recursively. The guard condition as described in Remark 6.3 holds iff the value of the list at the time of the recursive call is less than the value the process started with, in lexicographical order. Here, for example,* $[0,1,-1,0] \not< [0,0,0,0]$*, and the guard condition does not hold for this recursive call.*

We leave it to the reader to verify that no matter how we assign priorities of the type variables in Σ_4*, our condition rejects* `PingPong`.

The following definition captures the idea of forming lists described above. Rather than directly referring to type variables such as ack or astream we just refer to their priorities, since that is the relevant information.

Definition 6.4. For a process
$$\bar{x} : \omega \vdash P :: (y : B),$$
over the signature Σ, define $list(\bar{x}, y) = [f_i]_{i \leq n}$ such that

1. $f_i = (\bar{x}_i, y_i)$ if $\epsilon(i) = \mu$, and
2. $f_i = (y_i, \bar{x}_i)$ if $\epsilon(i) = \nu$,

where n is the lowest priority in Σ.

In the remainder of this section we use n to denote the lowest priority in Σ (which is numerically maximal).

Example 6.7. *Consider the signature* Σ_1 *and program* $\mathcal{P}_3 := \langle\{\texttt{Copy}\}, \texttt{Copy}\rangle$*, from Example 6.2:*
$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\}$*, and*

$$\begin{array}{l}
y \leftarrow \texttt{Copy} \leftarrow x = \textbf{case}\, Lx\ (\mu_{nat} \Rightarrow \textbf{case}\, Lx\ (\ z \Rightarrow Ry.\mu_{nat}; Ry.z; \textbf{wait}\, Lx; \textbf{close}\, Ry \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \texttt{Copy} \leftarrow x))
\end{array}$$

By Definition 6.4, for process $x : \mathsf{nat} \vdash \texttt{Copy} :: (y : \mathsf{nat})$*, we have* $n = 1$*, and* $list(x, y) = [(x_1, y_1)]$ *since* $\epsilon(1) = \mu$*. Just as for the naive version of the algorithm, we can trace the value of*

$list(x, y)$:

$$
\begin{array}{ll}
y \leftarrow \texttt{Copy} \leftarrow x = & [0,0] \\
\quad \textbf{case}\, Lx\ (\mu_{nat} \Rightarrow & [-1,0] \\
\quad\quad \textbf{case}\, Lx\ (z \Rightarrow Ry.\mu_{nat}; & [-1,1] \\
\quad\quad\quad R.z; \textbf{wait}\, Lx; \textbf{close}\, Ry & [-1,1] \\
\quad\quad \mid s \Rightarrow Ry.\mu_{nat}; & [-1,1] \\
\quad\quad\quad Ry.s; y \leftarrow \texttt{Copy} \leftarrow x & [-1,1]
\end{array}
$$

Here, $[-1, 1] < [0, 0]$ and the recursive call is classified as guarded.

To capture the idea of *decreasing/increasing* the value of the elements of $list(\_, \_)$ by *one*, as depicted in Example 6.6 and Example 6.7, we distinguish between different generation of channels. A channel transforms into a new generation of itself after sending or receiving a fixed point unfolding message.

Example 6.8. *Process $x : \mathsf{nat} \vdash y \leftarrow \texttt{Copy} \leftarrow x :: (y : \mathsf{nat})$ in Example 6.7 starts its computation with the initial generation of its left and right channels:*

$$x^0 : \mathsf{nat} \vdash y^0 \leftarrow \texttt{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat}).$$

The channels evolve as the process sends or receives a fixed point unfolding message along them:

$$
\begin{array}{ll}
y^0 \leftarrow \texttt{Copy} \leftarrow x^0 = & \\
\quad \textbf{case}\, Lx^0\ (\mu_{nat} \Rightarrow & x^0 \rightsquigarrow x^1 \\
\quad\quad \textbf{case}\, Lx^1\ (z \Rightarrow Ry^0.\mu_{nat}; & y^0 \rightsquigarrow y^1 \\
\quad\quad\quad Ry^1.z; \textbf{wait}\, Ly^1; \textbf{close}\, Rx^1 & \\
\quad\quad \mid s \Rightarrow Ry^0.\mu_{nat}; & y^0 \rightsquigarrow y^1 \\
\quad\quad\quad Ry^1.s; y^1 \leftarrow \texttt{Copy} \leftarrow x^1)) &
\end{array}
$$

On the last line the process

$$x^1 : \mathsf{nat} \vdash y^1 \leftarrow \texttt{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})$$

is called recursively with a new generation of variables.

In the inference rules introduced in Section 6.6, instead of recording the value of each element of $list(\_, \_)$ as we did in Example 6.6 and Example 6.7, we introduce Ω to track the relation between different generations of a channel indexed by their priority of types.

Remark 6.5. Generally speaking, $x_i^{\alpha+1} < x_i^{\alpha}$ is added to Ω, when x^{α} *receives* a fixed point unfolding message for a type with priority i and transforms to $x^{\alpha+1}$. This corresponds to the *decrease by one* in the previous examples.

If x^α *sends* a fixed point unfolding message for a type with priority i is *sent* on x^α, which then evolves to $x^{\alpha+1}$, x^α_i and $x^{\alpha+1}_i$ are considered to be incomparable in Ω. This corresponds to *increase by one* in the previous examples, since for the sake of lexicographically comparing the value of $list(\_, \_)$ at the *first call* of a process to its value just before a *recursive call*, there is no difference whether $x^{\alpha+1}$ is greater than x^α or incomparable to it.

When x^α receives/sends a fixed point unfolding message of a type with priority i and transforms to $x^{\alpha+1}$, for any type with priority $j \neq i$, the value of x^α_j and $x^{\alpha+1}_j$ must remain equal. In these steps, we add $x^\alpha_j = x^{\alpha+1}_j$ for $j \neq i$ to Ω.

A process in the formalization of the intuition above is therefore typed as

$$x^\alpha : A \vdash_\Omega P :: (y^\beta : B),$$

where x^α is the α-th generation of channel x. The syntax and operational semantics of the processes with generational channels are the same as the corresponding definitions introduced in Section 5.5; we simply ignore generations over the channels to match processes with the previous definitions. We enforce the assumption that channel x^α transforms to its next generation $x^{\alpha+1}$ upon sending/receiving a fixed point unfolding message in the typing rules of Section 6.6.

The relation between the channels indexed by their priority of types is built step by step in Ω and represented by $\le$. The reflexive transitive closure of Ω forms a partial order $\le_\Omega$. We extend $\le_\Omega$ to the *list* of channels indexed by the priority of their types considered lexicographically. We may omit subscript Ω from $\le_\Omega$ whenever it is clear from the context. In the next examples, we present the set of relations Ω in the rightmost column.

6.4 Mutual Recursion in the Local guard algorithm

In examples of previous sections, the recursive calls were not *mutual*. In the general case, a process may call any other process variable in the program, and this call can be mutually recursive. In this section, we incorporate mutual recursive calls into our algorithm.

Example 6.9. *Recall signature Σ_4 from Example 6.5*

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : nat\}\end{aligned}$$

Define program $\mathcal{P}_7 = \langle\{\texttt{Idle}, \texttt{Producer}\}, \texttt{Producer}\rangle$, where

$$\begin{aligned}&z : \mathsf{ack} \vdash w \leftarrow \texttt{Idle} \leftarrow z :: (w : \mathsf{nat})\\ &x : \mathsf{astream} \vdash y \leftarrow \texttt{Producer} \leftarrow x :: (y : \mathsf{nat}),\end{aligned}$$

and processes `Idle` *(or simply* `I`*) and* `Producer` *(or simply* `P` *) are defined as:*

$$w \leftarrow \mathtt{I} \leftarrow z = \textbf{case}\ Lz\ (\mu_{ack} \Rightarrow \textbf{case}\ Lz\ (ack \Rightarrow Rw.\mu_{nat}; Rw.s; w \leftarrow \mathtt{P} \leftarrow z))$$
$$y \leftarrow \mathtt{P} \leftarrow x = Lx.\nu_{astream}; Lx.head; y \leftarrow \mathtt{I} \leftarrow x.$$

We have $list(x, y) = [(x_1, y_1), (y_2, x_2), (x_3, y_3)]$ *and* $list(z, w) = [(z_1, w_1), (w_2, z_2), (z_3, w_3)]$ *since* $\epsilon(1) = \epsilon(3) = \mu$ *and* $\epsilon(2) = \nu$.

By analyzing the behavior of this program step by step, we see that it is a reactive program that counts the number of acknowledgements received from the left. The program starts with the process

$$x^0 : \mathsf{astream} \vdash_{\emptyset} y^0 \leftarrow \mathtt{Producer} \leftarrow x^0 :: (y^0 : \mathsf{nat}).$$

It first sends one message to left to unfold the negative fixed point type, and its left channel evolves to a next generation. Then another message is sent to the left to request the head *of the stream and after that it calls process* $y^0 \leftarrow \mathtt{Idle} \leftarrow x^1$.

$y^0 \leftarrow \mathtt{Producer} \leftarrow x^0 =$	$[0, 0, 0, 0, 0, 0]$	
$\quad Lx^0.\nu_{astream};$	$[0, 0, 0, 1, 0, 0]$	$x^1_1 = x^0_1, x^1_3 = x^0_3$
$\quad\quad Lx^1.head; y^0 \leftarrow \mathtt{Idle} \leftarrow x^1$	$[0, 0, 0, 1, 0, 0]$	

Process $x^1 : \mathsf{ack} \vdash y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 :: (y^0 : \mathsf{nat})$, *then waits to receive an acknowledgment from the left via a positive fixed point unfolding message for* ack *and its left channel transforms into a new generation upon receiving it. Then it waits for the label* ack, *upon receiving it, sends one message to the right to unfold the positive fixed point* nat *(and this time the right channel evolves). Then it sends the label* s *to the right and calls* $y^1 \leftarrow \mathtt{Producer} \leftarrow x^2$ *recursively:*

$y^0 \leftarrow \mathtt{Idle} \leftarrow x^1 =$	$[0, 0,\ 0,\ 1,\ 0, 0]$	
$\quad \textbf{case}\ Lx^1\ (\mu_{ack} \Rightarrow$	$[-1, 0, 0, 1, 0, 0]$	$x^2_1 < x^1_1, x^2_2 = x^1_2, x^2_3 = x^1_3$
$\quad\quad \textbf{case}\ Lx^2\ (ack \Rightarrow Ry^0.\mu_{nat};$	$[-1, 0, 0, 1, 0, 1]$	$y^1_1 = y^0_1, y^1_2 = y^0_2$
$\quad\quad\quad Ry^1.s; y^1 \leftarrow \mathtt{Producer} \leftarrow x^2))$	$[-1, 0, 0, 1, 0, 1]$	

Observe that the actual recursive call for `Producer` *occurs at the last line (in red) above, where* `Producer` *eventually calls itself. At that point the value of* $list(x^2, y^1)$ *is recorded as* $[-1, 0, 0, 1, 0, 1]$, *which is less than the value of* $list(x^0, y^0)$ *when* `Producer` *was called for the first time:*

$$[-1, 0, 0, 1, 0, 1] < [0, 0, 0, 0, 0, 0].$$

The same observation can be made by considering the relations introduced in the last column

$$list(x^2, y^1) = [(x^2_1, y^1_1), (y^1_2, x^2_2), (x^2_3, y^1_3)] < [(x^0_1, y^0_1), (y^0_2, x^0_2), (x^0_3, y^0_3)] = list(x^0, y^0)$$

since $x_1^2 < x_1^1 = x_1^0$*. This recursive call is guarded regardless of the fact that* $[0,0,0,1,0,0] \not< [0,0,0,0,0,0]$*, i.e.*

$$list(x^1, y^0) = [(x_1^1, y_1^0), (y_2^0, x_2^1), (x_3^1, y_3^0)] \not< [(x_1^0, y_1^0), (y_2^0, x_2^0), (x_3^0, y_3^0)] = list(x^0, y^0)$$

since $x_1^1 = x_1^0$ *but* x_2^1 *is incomparable to* x_2^0*. Similarly, we can observe that the actual recursive call on* `Idle`*, where* `Idle` *eventually calls itself, is guarded.*

To account for this situation, we introduce an order on process variables and trace the last seen variable on the path leading to the recursive call. In this example we define `Idle` *to be less than* `Producer` *at position* 2 ($\texttt{I} \subset_2 \texttt{P}$)*. We incorporate process variables* `Producer` *and* `Idle` *into the lexicographical order on* $list(\_,\_)$ *such that their values are placed exactly before the element in the list corresponding to the sent unfolding messages of the type with priority* 2*.*

We now trace the ordering as follows:

$$\begin{array}{lll}
y^0 \leftarrow \texttt{Producer} \leftarrow x^0 = & [0,0,0,\texttt{P},0,0,0] & \\
\quad Lx^0.\nu_{astream}; & [0,0,0,\texttt{P},1,0,0] & x_1^1 = x_1^0, x_3^1 = x_3^0 \\
\qquad Lx^1.head; y^0 \leftarrow \texttt{Idle} \leftarrow x^1 & [0,0,0,\texttt{I},1,0,0] & \\
 & & \\
y^0 \leftarrow \texttt{Idle} \leftarrow x^1 = & [0,0,0,\texttt{I},1,0,0] & \\
\quad \textbf{case}\, Lx^1(\mu_{ack} \Rightarrow & [-1,0,0,\texttt{I},1,0,0] & x_1^2 < x_1^1, x_2^2 = x_2^1, x_3^2 = x_3^1 \\
\qquad \textbf{case}\, Lx^2(ack \Rightarrow Ry^0.\mu_{nat}; & [-1,0,0,\texttt{I},1,0,1] & y_1^1 = y_1^0, y_2^1 = y_2^0 \\
\qquad\quad Ry^1.s; y^1 \leftarrow \texttt{Producer} \leftarrow x^2 & [-1,0,0,\texttt{P},1,0,1] &
\end{array}$$

$[-1,0,0,\texttt{P},1,0,1] < [0,0,0,\texttt{I},1,0,0]$ *and* $[0,0,0,\texttt{I},1,0,0] < [0,0,0,\texttt{P},0,0,0]$ *hold, and both mutually recursive calls are recognized to be guarded, as they are, without a need to substitute process definitions.*

However, not every relation over the process variables forms a partial order. For instance, having both $\texttt{P} \subset_2 \texttt{I}$ and $\texttt{I} \subset_2 \texttt{P}$ violates the antisymmetry condition. Introducing the position of process variables into $list(\_,\_)$ is also a delicate issue. For example, if we have both $\texttt{I} \subset_1 \texttt{P}$ and $\texttt{I} \subset_2 \texttt{P}$, it is not determined where to insert the value of `Producer` and `Idle` on the $list(\_,\_)$. Definition 6.6 captures the idea of Example 6.9. It defines the relation $\subseteq$, given that the programmer introduces a family of partial orders such that their domains partition the set of process variables V. We again assume that the programmer defines this family based on the intuition of why a program satisfies strong progress. Definition 6.9 ensures that $\subseteq$ is a well-defined partial order and it is uniquely determined in which position of $list(\_,\_)$ the process variables shall be inserted. Definition 6.8 gives the lexicographic order on $list(\_,\_)$ augmented with the $\subseteq$ relation.

Definition 6.6. Consider a program $\mathcal{P} = \langle V, S \rangle$ defined over a signature Σ. Let $\{\subseteq_i\}_{0 \leq i \leq n}$ be a disjoint family of partial orders whose domains partition the set of process variables V, where (a) $X \cong_i Y$ iff $X \subseteq_i Y$ and $Y \subseteq_i X$, and (b) $X \subset_i Y$ iff $X \subseteq_i Y$ but $X \ncong_i Y$.

We define $\subseteq$ as $\bigcup_{i \leq n} \subseteq_i$, i.e. $F \subseteq G$ iff $F \subseteq_i G$ for some (unique) $i \leq n$. It is straightforward to see that $\subseteq$ is a partial order over the set of process variables V. Moreover, we define (c) $X \cong Y$ iff $X \cong_i Y$ for some (unique) i, and (d) $X \subset Y$ iff $X \subset_i Y$ for some (unique) i.

To integrate the order on process variables ($\subset$) with the order $<$, we need a prefix of the list from Definition 6.4. We give the following definition of $list(x, y, j)$ to crop $list(x, y)$ exactly before the element corresponding to a *sent* fixed point unfolding message for types with priority j.

Definition 6.7. For a process

$$\bar{x} : A \vdash P :: y : B,$$

over signature Σ, and $0 \leq j \leq n$, define $list(\bar{x}, y, j)$, as a prefix of the list $list(\bar{x}, y) = [v_i]_{i \leq n}$ by

1. $[]$ if $i = 0$,
2. $[[v_i]_{i<j},\ (\bar{x}_j)]$ if $\epsilon(j) = \mu$,
3. $[[v_i]_{i<j},\ (y_j)]$ if $\epsilon(j) = \nu$.

We use these prefixes in the following definition.

Definition 6.8. Using the orders $\subset$ and $\leq$, we define a new combined order $(\subset, <)$ (used in the local guard condition in Section 6.7).

$$F, list(\bar{x}, y)\ (\subset, <)\ G, list(\bar{z}, w)$$

iff

1. If $F \subset G$, i.e., $F \subset_i G$ for a unique i, then $list(\bar{x}, y, i) \leq list(\bar{z}, w, i)$, otherwise,
2. if $F \cong G$ and $list(\bar{x}, y) < list(\bar{z}, w)$, otherwise
3. $list(\bar{x}, y, \min(i, j)) < list(\bar{z}, w, \min(i, j))$, where F is in the domain of $\subseteq_i$ and G is in the domain of $\subseteq_j$.

By conditions of Definition 6.6, $(\subset, <)$ is an irreflexive and transitive relation and thus a strict partial order.

Example 6.10. *Consider the signature of Example 6.9*

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{\mathit{ack} : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{\mathit{head} : \mathsf{ack},\ \ \mathit{tail} : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : \mathit{nat}\}\end{aligned}$$

and program $\mathcal{P}_7 := \langle\{\texttt{Idle}, \texttt{Producer}\}, \texttt{Producer}\rangle$ *with the relation* $\subset$ *defined over process variables as* $\texttt{Idle} \subset_2 \texttt{Producer}$*. For process* $x : \mathsf{astream} \vdash \texttt{Producer} :: (y : \mathsf{nat})$*:*

$$\begin{aligned}&\mathit{list}(x,y) = [(x_1,y_1),(y_2,x_2),(x_3,y_3)],\\ &\mathit{list}(x,y,3) = [(x_1,y_1),(y_2,x_2),(x_3)],\\ &\mathit{list}(x,y,2) = [(x_1,y_1),(y_2)],\\ &\mathit{list}(x,y,1) = [(x_1)],\ \ \mathit{and}\\ &\mathit{list}(x,y,0) = [].\end{aligned}$$

To check that the recursive calls in Example 6.9 are guarded we observe that

- $\texttt{Producer}, \mathit{list}(x^2,y^1)\,(\subset,<)\,\texttt{Idle}, \mathit{list}(x^1,y^0)$ *since* $\mathit{list}(x^2,y^1,2) < \mathit{list}(x^1,y^0,2)$*, and*
- $\texttt{Idle}, \mathit{list}(x^1,y^0)\,(\subset,<)\,\texttt{Producer}, \mathit{list}(x^0,y^0)$ *since* $\mathit{list}(x^1,y^0,2) = \mathit{list}(x^0,y^0,2)$ *and* $\texttt{Idle} \subset_2 \texttt{Producer}$.

6.5 A modified rule for cut

There is a subtle aspect of the local guard condition that we have not discussed yet. We need to relate a fresh channel, created by spawning a new process, with the previously existing channels. Process $y^\alpha : A \vdash (x \leftarrow P_x; Q_x) :: (z^\beta : B)$, for example, creates a fresh channel w^0, spawns process P_{w^0} providing along channel w^0, and then continues as Q_{w^0}. For the sake of our algorithm, we need to identify the relation between w^0, y^α, and z^β. Since w^0 is a fresh channel, a naive idea is to make w^0 incomparable to any other channel for any type variable $t \in \Sigma$. To represent this incomparability in our examples we write "∞" for the value of the fresh channel. While sound, we will see in Example 6.11 that we can improve on this naive approach to cover more guarded processes.

Example 6.11. *Define the signature*

$$\begin{aligned}\Sigma_5 := \mathsf{ctr} &=^1_\nu \&\{\mathit{inc} : \mathsf{ctr},\ \ \mathit{val} : \mathsf{bin}\},\\ \mathsf{bin} &=^2_\mu \oplus\{\mathit{b0} : \mathsf{bin}, \mathit{b1} : \mathsf{bin}, \$: 1\}.\end{aligned}$$

which provides numbers in binary representation as well as an interface to a counter. We explore the following program $\mathcal{P}_8 = \langle\{\texttt{BinSucc}, \texttt{Counter}, \texttt{NumBits}, \texttt{BitCount}\}, \texttt{BitCount}\rangle$, *where*

$$\begin{array}{l}
x : \mathsf{bin} \vdash y \leftarrow \texttt{BinSucc} \leftarrow x :: (y : \mathsf{bin}) \\
x : \mathsf{bin} \vdash y \leftarrow \texttt{Counter} \leftarrow x :: (y : \mathsf{ctr}) \\
x : \mathsf{bin} \vdash y \leftarrow \texttt{NumBits} \leftarrow x :: (y : \mathsf{bin}) \\
x : \mathsf{bin} \vdash y \leftarrow \texttt{BitCount} \leftarrow x :: (y : \mathsf{ctr})
\end{array}$$

We define the relation $\subset$ *on process variables as* $\texttt{BinSucc} \subset_0 \texttt{Counter} \subset_0 \texttt{BitCount}$ *and* $\texttt{BinSucc} \subset_0 \texttt{NumBits} \subset_0 \texttt{BitCount}$. *Process* $y^\beta \leftarrow \texttt{Counter} \leftarrow w^\alpha$ *as its name suggests works as a counter where* $w : bin$ *is the current value of the counter. When it receives an increment message* inc *it computes the successor of* w*, accessible through channel* z*. If it receives a* val *message it simply forwards the current value* (w) *to the client* (y)*. Note that in this process, both calls are guarded according to the condition developed so far. This is also true for the binary successor process* BinSucc*, which presents no challenges. The only recursive call represents the "carry" of binary addition when a number with lowest bit* $b1$ *has to be incremented.*

The process $w^\beta \leftarrow \texttt{NumBits} \leftarrow x^\alpha$ *counts the number of bits in the binary number* x *and sends the result along* w*, also in the form of a binary number. It calls itself recursively for every bit received along* x *and increments the result* z *to be returned along* w*. Note that if there are no leading zeros, this computes essentially the integer logarithm of* x*.*

The process definitions are as follows, shown here already with their termination analysis.

$$\begin{array}{lll}
w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^\alpha = & [0, 0,\ 0,\ 0] & \\
\quad \mathbf{case}\, Lz^\alpha\ (\mu_{bin} \Rightarrow & [0, 0, -1, 0] & z_1^{\alpha+1} = z_1^\alpha, z_2^{\alpha+1} < z_2^\alpha \\
\quad \mathbf{case}\, Lz^{\alpha+1}\ (b0 \Rightarrow Rw^\beta.\mu_{bin}; & [0, 0, -1, 1] & w_1^{\beta+1} = w_1^\beta \\
\qquad\qquad Rw^{\beta+1}.b1; w^{\beta+1} \leftarrow z^{\alpha+1} & [0, 0, -1, 1] & \\
\qquad \mid b1 \Rightarrow Rw^\beta.\mu_{bin}; & [0, 0, -1, 1] & w_1^{\beta+1} = w_1^\beta \\
\qquad\qquad Rw^{\beta+1}.b0; w^{\beta+1} \leftarrow \texttt{BinSucc} \leftarrow z^{\alpha+1} & [0, 0, -1, 1] & \\
\qquad \mid \$ \Rightarrow Rw^\beta.\mu_{\mathsf{bin}}; Rw^{\beta+1}.b1; & [0, 0, -1, 1] & w_1^{\beta+1} = w_1^\beta \\
\qquad\qquad Rw^{\beta+1}.\mu_{\mathsf{bin}}; Rw^{\beta+2}.\$;\ w^{\beta+2} \leftarrow z^{\alpha+1})) & [0, 0, -2, 2] & w_1^{\beta+2} = w_1^{\beta+1}
\end{array}$$

$$\begin{array}{lll}
y^\beta \leftarrow \texttt{Counter} \leftarrow w^\alpha = & [0, 0,\ 0,\ 0] & \\
\quad \mathbf{case}\, Ry^\beta\ (\nu_{ctr} \Rightarrow & [-1, 0, 0, 0] & y_1^{\beta+1} < y_1^\beta, y_2^{\beta+1} = y_2^\beta \\
\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow z^0 \leftarrow \texttt{BinSucc} \leftarrow w^\alpha; & & \texttt{BinSucc} \subset_0 \texttt{Counter} \\
\qquad\qquad y^{\beta+1} \leftarrow \texttt{Counter} \leftarrow z^0 & [-1,\ \infty,\ \infty, 0] & \\
\qquad \mid val \Rightarrow y^{\beta+1} \leftarrow w^\alpha)) & [-1, 0, 0, 0] &
\end{array}$$

```
$w^\beta \leftarrow \texttt{NumBits} \leftarrow x^\alpha =$                                      $[0, 0,\ 0,\ 0]$
 $\textbf{case}\ Lx^\alpha\ (\mu_{bin} \Rightarrow$                                          $[0, 0, -1, 0]$  $x_1^{\alpha+1} = x_1^\alpha, x_2^{\alpha+1} < x_2^\alpha$
   $\textbf{case}\ Lx^{\alpha+1}\ (b0 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$      $[?, 0, -1, ?]$   $z_1^0 \overset{?}{=} w_1^\beta, z_2^0 \overset{?}{=} w_2^\beta$
                 $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                                  $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
            $\mid b1 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$            $[?, 0, -1, ?]$   $z_1^0 \overset{?}{=} w_1^\beta, z_2^0 \overset{?}{=} w_2^\beta$
                 $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                                  $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
            $\mid \$ \Rightarrow Rw^\beta.\mu_{\mathsf{bin}}; Rw^{\beta+1}.\$;\ w^{\beta+1} \leftarrow x^{\alpha+1}))$     $[0, 0, -1, 1]$
```

$y^\beta \leftarrow \texttt{BitCount} \leftarrow x^\alpha = w^0 \leftarrow \texttt{NumBits} \leftarrow x^\alpha; y^\beta \leftarrow \texttt{Counter} \leftarrow w^0$

The program starts with process BitCount *which creates a fresh channel* w^0*, spawns a new process* $w^0 \leftarrow \texttt{NumBits} \leftarrow x^\alpha$*, and continues as* $y^\beta \leftarrow \texttt{Counter} \leftarrow w^0$.

The process NumBits *is reactive. However with our approach toward spawning a new process, the recursive calls have the list value* $[\infty, 0, -1, \infty] \not< [0, 0, 0, 0]$*, meaning that the local guard condition developed so far fails.*

Note that we cannot just define $z_1^0 = w_1^\beta$ *and* $z_2^0 = w_2^\beta$*, or* $z_1^0 = z_2^0 = 0$*. Channel* z^0 *is a fresh one and its relation with the future generations depends on how it evolves in the process* $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$*. But by definition of type* bin*, no matter how* $z^0 : \mathsf{bin}$ *evolves to some* z^η *in process* BinSucc*, it won't be the case that* $z^\eta : \mathsf{ctr}$*. In other words, the type* ctr *is not visible from* bin *and for any generation* η*, channel* z^η *does not send or receive a* ctr *unfolding message. So in this recursive call, the value of* z_1^η *is not important anymore and we safely put* $z_1^0 = w_1^\beta$*. In the improved version of the condition we have:*

```
$w^\beta \leftarrow \texttt{NumBits} \leftarrow x^\alpha =$                                  $[0, 0,\ 0,\ 0]$
 $\textbf{case}\ Lx^\alpha\ (\mu_{bin} \Rightarrow$                                      $[0, 0, -1, 0]$  $x_1^{\alpha+1} = x_1^\alpha, x_2^{\alpha+1} < x_2^\alpha$
     $\textbf{case}\ Lx^{\alpha+1}\ (b0 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$  $[0, 0, -1, \infty]$   $z_1^0 = w_1^\beta$
                  $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                          $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
             $\mid b1 \Rightarrow z^0 \leftarrow \texttt{NumBits} \leftarrow x^{\alpha+1};$        $[0, 0, -1, \infty]$   $z_1^0 = w_1^\beta$
                  $w^\beta \leftarrow \texttt{BinSucc} \leftarrow z^0$                          $\texttt{BinSucc} \subset_0 \texttt{NumBits}$
             $\mid \$ \Rightarrow Rw^\beta.\mu_{\mathsf{bin}}; Rw^{\beta+1}.\$;\ w^{\beta+1} \leftarrow x^{\alpha+1}))$  $[0, 0, -1, 1]$
```

This version of the algorithm recognizes both recursive calls as guarded. In the following definition we capture the idea of visibility from a type more formally.

Definition 6.9. For type A in a given signature Σ and a set of type variables Δ, we define $\mathsf{c}(A;\Delta)$ inductively as:

$$\begin{aligned}
&\mathsf{c}(1;\Delta) = \emptyset,\\
&\mathsf{c}(\oplus\{\ell : A_\ell\}_{\ell\in L};\Delta) = \mathsf{c}(\&\{\ell : A_\ell\}_{\ell\in L};\Delta) = \textstyle\bigcup_{\ell\in L}\mathsf{c}(A_\ell;\Delta),\\
&\mathsf{c}(t;\Delta) = \{t\}\cup\mathsf{c}(A;\Delta\cup\{t\}) \text{ if } t =_a A \in \Sigma \text{ and } t \notin \Delta,\\
&\mathsf{c}(t;\Delta) = \{t\} \text{ if } t =_a A \in \Sigma \text{ and } t \in \Delta.
\end{aligned}$$

We put priority i in the set $\mathsf{c}(A)$ iff for some type variable t with $i = p(t)$, $t \in \mathsf{c}(A;\emptyset)$. We say that *priority* i *is visible from type* A if and only if $i \in \mathsf{c}(A)$.

In Example 6.11, we have $\mathsf{c}(\mathsf{bin}) = \{p(\mathsf{bin})\} = \{2\}$ and $\mathsf{c}(\mathsf{ctr}) = \{p(\mathsf{bin}), p(\mathsf{ctr})\} = \{1, 2\}$ which means that bin is visible from ctr but not the other way around. This expresses that the definition of ctr references bin, but the definition of bin does not reference ctr.

6.6 Typing rules for session-typed processes with channel ordering

In this section we introduce infinitary inference rules for session-typed processes corresponding to derivations in subsingleton logic with fixed points. This is a refinement of the process typing rules presented in Chapter 5 to account for channel generations and orderings introduced in previous sections. This system rules out communication mismatches without forcing processes to actually communicate along their external channels. It is the basis for our finitary system for the local guard condition in Section 6.7, and Section 6.8 where we prove that our local guard condition is stricter than Fortier and Santocanale's validity condition.

The judgments are of the form

$$\bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : A),$$

where P is a process, and x^α (the α-th generation of channel x) and y^β (the β-th generation of channel y) are its left and right channels of types ω and A, respectively. The order relation between the generations of left and right channels indexed by their priority of types is built step by step in Ω when reading the rules from the conclusion to the premises. We only consider judgments in which all variables $x^{\alpha'}$ occurring in Ω are such that $\alpha' \leq \alpha$ and, similarly, for $y^{\beta'}$ in Ω we have $\beta' \leq \beta$. This presupposition guarantees that if we construct a derivation bottom-up, any future generations for x and y are fresh and not yet constrained by Ω. All our rules, again read bottom-up, will preserve this property.

We fix a signature Σ as in Definition 5.2, a finite set of process definitions V over Σ, and define $\bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : A)$ with the rules in Figure 6.1. To preserve freshness of channels and their future generations in Ω, the channel introduced by Cut rule must be distinct from any variable mentioned in Ω. This system is infinitary, i.e., an infinite derivation may be produced for a given program. However, we can remove the first premise from the Def rule and check typing for each process definition in V separately.

6.7 A local guard condition

In Sections 6.1 to 6.4, using several examples, we developed an algorithm for identifying *guarded* programs. Illustrating the full algorithm based on the inference rules in Section 6.6 was postponed to this section. We reserve for the next section our main result that the programs accepted by this algorithm satisfy the validity condition introduced by Fortier and Santocanale [36].

The condition checked by our algorithm is a *local* one in the sense that we check the guard condition for each process definition in a program separately. The algorithm works on the sequents of the form

$$\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} P :: (w^\beta : C),$$

where $\bar{u}^\gamma$ is the left channel of the process the algorithm started with and can be either empty or u^γ. Similarly, v^δ is the right channel of the process the algorithm started with (which cannot be empty). And X is the last process variable a definition rule has been applied to (reading the rules bottom-up). Again, in this judgment the (in)equalities in Ω can only relate variables z and w from earlier generations to guarantee freshness of later generations.

Generally speaking, when analysis of the program starts with $\bar{u}^\gamma : \omega \vdash v^\delta \leftarrow X \leftarrow \bar{u}^\gamma :: (v^\delta : B)$, a snapshot of the channels $\bar{u}^\gamma$ and v^δ and the process variable X are saved. Whenever the process reaches a call $\bar{z}^\alpha : \_ \vdash w^\beta \leftarrow Y \leftarrow \bar{z}^\alpha :: (w^\beta : \_)$, the algorithm compares $X, list(\bar{u}^\gamma, v^\delta)$ and $Y, list(\bar{z}^\alpha, w^\beta)$ using the $(\subset, <)$ order to determine if the call is (locally) guarded. This comparison is made by the Call rule in the rules in Figure 6.2, and is local in the sense that only the interface of a process is consulted at each call site, not its definition. Since it otherwise follows the structure of the program it is also local in the sense of Pierce and Turner [75].

Definition 6.10. A program $\mathcal{P} = \langle V, S \rangle$ over signature Σ and a fixed order $\subset$ satisfying the properties in Definition 6.6 is *locally guarded* iff for every $\bar{z} : A \vdash X = P_{\bar{z},w} :: (w : C) \in V$, there is a derivation for

$$\langle \bar{z}^0, X, w^0 \rangle; \bar{z}^0 : \omega \vdash_{\emptyset,\subset} P_{\bar{z}^0,w^0} :: (w^0 : C)$$

in the rule system in Figure 6.2. This set of rules is *finitary* so it can be directly interpreted as an algorithm. This results from substituting the Def rule (of Figure 6.1) with the Call rule (of Figure 6.2). Again, to guarantee freshness of future generations of channels, the channel introduced by Cut rule is distinct from other variables mentioned in Ω.

$$\frac{}{x^\alpha : A \vdash_\Omega y^\beta \leftarrow x^\alpha :: (y^\beta : A)}\ \textsc{Id}$$

$$\frac{\begin{array}{l} \mathtt{r}(v) = \{w_i^0 = v_i \mid i \notin \mathtt{c}(A) \text{ and } i \le n\} \\ \quad \bar{x}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(y^\beta)} P_{w^0} :: (w^0 : A) \qquad w^0 : A \vdash_{\Omega\cup\mathtt{r}(\bar{x}^\alpha)} Q_{w^0} :: (y^\beta : C)\end{array}}{\bar{x}^\alpha : \omega \vdash_\Omega (w \leftarrow P_w; Q_w) :: (y^\beta : C)}\ \textsc{Cut}^w$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega P :: (y^\beta : A_k) \quad (k \in L)}{\bar{x}^\alpha : \omega \vdash_\Omega R y^\beta.k; P :: (y^\beta : \oplus\{\ell : A_\ell\}_{\ell\in L})}\ \oplus R$$

$$\frac{\forall \ell \in L \quad x^\alpha : A_\ell \vdash_\Omega P_\ell :: (y^\beta : C)}{x^\alpha : \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash_\Omega \textbf{case}\, L x^\alpha\ (\ell \Rightarrow P_\ell) :: (y^\beta : C)}\ \oplus L$$

$$\frac{\forall \ell \in L \quad \bar{x}^\alpha : \omega \vdash_\Omega P_\ell :: (y^\beta : A_\ell)}{\bar{x}^\alpha : \omega \vdash_\Omega \textbf{case}\, R y^\beta\ (\ell \Rightarrow P_\ell) :: (y^\beta : \&\{\ell : A_\ell\}_{\ell\in L})}\ \& R$$

$$\frac{k \in L \quad x^\alpha : A_k \vdash_\Omega P :: (y^\beta : C)}{x^\alpha : \&\{\ell : A_l\}_{\ell\in L} \vdash_\Omega L x^\alpha.k; P :: (y^\beta : C)}\ \& L$$

$$\frac{}{. \vdash_\Omega \textbf{close}\, R y^\beta :: (y^\beta : 1)}\ 1R$$

$$\frac{. \vdash_\Omega Q :: (y^\beta : A)}{x^\alpha : 1 \vdash_\Omega \textbf{wait}\, L x^\alpha; Q :: (y^\beta : A)}\ 1L$$

$$\frac{\begin{array}{l} \Omega' = \Omega \cup \{(y^\beta)_{p(s)} = (y^{\beta+1})_{p(s)} \mid p(s) \neq p(t)\} \\ \quad \bar{x}^\alpha : \omega \vdash_{\Omega'} P_{y^{\beta+1}} :: (y^{\beta+1} : A) \qquad\qquad t =_\mu A \end{array}}{\bar{x}^\alpha : \omega \vdash_\Omega R y^\beta.\mu_t; P_{y^\beta} :: (y^\beta : t)}\ \mu R$$

$$\frac{\begin{array}{l} \Omega' = \Omega \cup \{x^{\alpha+1}_{p(t)} < x^\alpha_{p(t)}\} \cup \{x^{\alpha+1}_{p(s)} = x^\alpha_{p(s)} \mid p(s) \neq p(t)\} \\ \quad x^{\alpha+1} : A \vdash_{\Omega'} Q_{x^{\alpha+1}} :: (y^\beta : C) \qquad\qquad t =_\mu A \end{array}}{x^\alpha : t \vdash_\Omega \textbf{case}\, L x^\alpha\ (\mu_t \Rightarrow Q_{x^\alpha}) :: (y^\beta : C)}\ \mu L$$

$$\frac{\begin{array}{l} \Omega' = \Omega \cup \{y^{\beta+1}_{p(t)} < y^\beta_{p(t)}\} \cup \{y^{\beta+1}_{p(s)} = y^\beta_{p(s)} \mid p(s) \neq p(t)\} \\ \quad \bar{x}^\alpha : \omega \vdash_{\Omega'} P_{y^{\beta+1}} :: (y^{\beta+1} : A) \qquad\qquad t =_\nu A \end{array}}{\bar{x}^\alpha : \omega \vdash_\Omega \textbf{case}\, R y^\beta\ (\nu_t \Rightarrow P_{y^\beta}) :: (y^\beta : t)}\ \nu R$$

$$\frac{\begin{array}{l} \Omega' = \Omega \cup \{(x^{\alpha+1})_{p(s)} = (x^\alpha)_{p(s)} \mid p(s) \neq p(t)\} \\ \quad x^{\alpha+1} : A \vdash_{\Omega'} Q_{x^{\alpha+1}} :: (y^\beta : C) \qquad\qquad t =_\nu A \end{array}}{x^\alpha : t \vdash_\Omega L x^\alpha.\nu_t; Q_{x^\alpha} :: (y^\beta : C)}\ \nu L$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega P_{\bar{x}^\alpha, y^\beta} :: (y^\beta : C) \quad \bar{u} : \omega \vdash X = P_{\bar{u},w} :: (w : C) \in V}{\bar{x}^\alpha : \omega \vdash_\Omega y^\beta \leftarrow X \leftarrow \bar{x}^\alpha :: (y^\beta : C)}\ \textsc{Def}(X)$$

FIGURE 6.1: Infinitary Typing Rules for Processes with Channel Ordering

$$\dfrac{}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : A \vdash_{\Omega,\subset} w^\beta \leftarrow z^\alpha :: (w^\beta : A)}\ \textsc{Id}$$

$$\dfrac{\begin{array}{l}\mathtt{r}(y) = \{x_i^0 = y_i \mid i \notin \mathtt{c}(A) \,\text{and}\, i \leq n\} \\ \langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega \cup \mathtt{r}(w^\beta),\subset} P_{x^0} :: (x^0 : A) \ \ \langle \bar{u}^\gamma, X, v^\delta \rangle; x^0 : A \vdash_{\Omega \cup \mathtt{r}(\bar{z}^\alpha),\subset} Q_{x^0} :: (w^\beta : C)\end{array}}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega\subset} (x \leftarrow P_x; Q_x) :: (w^\beta : C)}\ \textsc{Cut}^x$$

$$\dfrac{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} P :: (w^\beta : A_k) \quad (k \in L)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} R w^\beta.k; P :: (w^\beta : \oplus\{\ell : A_l\}_{\ell \in L})}\ \oplus R$$

$$\dfrac{\forall \ell \in L \quad \langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : A_\ell \vdash_{\Omega,\subset} P_\ell :: (w^\beta : C)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : \oplus\{\ell : A\}_{\ell \in L} \vdash_{\Omega,\subset} \mathbf{case}\, L z^\alpha\ (\ell \Rightarrow P_\ell) :: (w^\beta : C)}\ \oplus L$$

$$\dfrac{\forall \ell \in L \quad \langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} P_\ell :: (w^\beta : A_\ell)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} \mathbf{case}\, R w^\beta\ (\ell \Rightarrow P_\ell) :: (w^\beta : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\dfrac{(k \in L) \quad \langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : A_k \vdash_{\Omega,\subset} P :: (w^\beta : C)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : \&\{\ell : A_\ell\}_{\ell \in L} \vdash_{\Omega,\subset} L z^\alpha.k; P :: (w^\beta : C)}\ \& L$$

$$\dfrac{}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \cdot \vdash_{\Omega,\subset} \mathbf{close}\, R :: (w^\beta : 1)}\ 1R \qquad \dfrac{\langle \bar{u}^\gamma, X, v^\delta \rangle; \cdot \vdash_{\Omega,\subset} Q :: (w^\beta : A)}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : 1 \vdash_{\Omega,\subset} \mathbf{wait}\, L z^\alpha; Q :: (w^\beta : A)}\ 1L$$

$$\dfrac{\begin{array}{l}\Omega' = \Omega \cup \{w_{p(s)}^\beta = w_{p(s)}^{\beta+1} \mid p(s) \neq p(t)\} \\ \langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega',\subset} P_{w^{\beta+1}} :: (w^{\beta+1} : A) \quad t =_\mu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} R w^\beta.\mu_t; P_{w^\beta} :: (w^\beta : t)}\ \mu R$$

$$\dfrac{\begin{array}{l}\Omega' = \Omega \cup \{z_{p(t)}^{\alpha+1} < z_{p(t)}^\alpha\} \cup \{z_{p(s)}^{\alpha+1} = z_{p(s)}^\alpha \mid p(s) \neq p(t)\} \\ \langle \bar{u}^\gamma, X, v^\delta \rangle; z^{\alpha+1} : A \vdash_{\Omega',\subset} Q_{z^{\alpha+1}} : (w^\beta :: C) \qquad t =_\mu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : t \vdash_{\Omega,\subset} \mathbf{case}\, L z^\alpha\ (\mu_t \Rightarrow Q_{z^\alpha}) :: (w^\beta : C)}\ \mu L$$

$$\dfrac{\begin{array}{l}\Omega' = \Omega \cup \{w_{p(t)}^{\beta+1} < w_{p(t)}^\beta\} \cup \{w_{p(s)}^{\beta+1} = w_{p(s)}^\beta \mid p(s) \neq p(t)\} \\ \langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega',\subset} P_{w^{\beta+1}} :: (w^{\beta+1} : A) \qquad t =_\nu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} \mathbf{case}\, R w^\beta\ (\nu_t \Rightarrow P_{w^\beta}) :: (w^\beta : t)}\ \nu R$$

$$\dfrac{\begin{array}{l}\Omega' = \Omega \cup \{z_{p(s)}^{\alpha+1} = z_{p(s)}^\alpha \mid p(s) \neq p(t)\} \\ \langle \bar{u}^\gamma, X, v^\delta \rangle; z^{\alpha+1} : A \vdash_{\Omega',\subset} Q_{z^{\alpha+1}} :: (w^\beta : C) \quad t =_\nu A\end{array}}{\langle \bar{u}^\gamma, X, v^\delta \rangle; z^\alpha : t \vdash_{\Omega,\subset} L z^\alpha.\nu_t; Q_{z^\alpha} :: (w^\beta : C)}\ \nu L$$

$$\dfrac{Y, list(\bar{z}^\alpha, w^\beta)\ (\subset, <_\Omega)\ X, list(\bar{u}^\gamma, v^\delta) \quad \bar{x} : \omega \vdash Y = P_{\bar{x},y} :: (y : C) \in V}{\langle \bar{u}^\gamma, X, v^\delta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega,\subset} w^\beta \leftarrow Y \leftarrow \bar{z}^\alpha :: (w^\beta : C)}\ \textsc{Call}$$

FIGURE 6.2: Finitary rules for the local guard condition

The starting point of the algorithm can be of an arbitrary form

$$\langle \bar{z}^\alpha, X, w^\beta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega, \subset} P_{z^\alpha, w^\beta} :: (w^\beta : C),$$

as long as $\bar{z}^{\alpha+i}$ and $w^{\beta+i}$ do not occur in Ω for every $i > 0$. In both the inference rules and the algorithm, the next generation of channels introduced in the $\mu/\nu - R/L$ rules do not occur in Ω. Having this condition we can convert a proof for

$$\langle \bar{z}^0, X, w^0 \rangle; \bar{z}^0 : \omega \vdash_{\emptyset, \subset} P_{z^0, w^0} :: (w^0 : C),$$

to a proof for

$$\langle \bar{z}^\alpha, X, w^\beta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega, \subset} P_{z^\alpha, w^\beta} :: (w^\beta : C),$$

by rewriting each $\bar{z}^\gamma$ and w^δ in the proof as $\bar{z}^{\gamma+\alpha}$ and $w^{\delta+\beta}$, respectively. This simple proposition is used in the next section where we prove that every locally guarded process accepted by our algorithm is a valid proof according to the FS validity condition.

Proposition 6.11. *If there is a deduction of*

$$\langle \bar{z}^0, X, w^0 \rangle; \bar{z}^0 : \omega \vdash_{\emptyset, \subset} P_{z^0, w^0} :: (w^0 : C),$$

then there is also a deduction of

$$\langle \bar{z}^\alpha, X, w^\beta \rangle; \bar{z}^\alpha : \omega \vdash_{\Omega, \subset} P_{z^\alpha, w^\beta} :: (w^\beta : C),$$

if for all $0 < i$, $\bar{z}^{\alpha+i}$ *and* $w^{\beta+i}$ *do not occur in* Ω.

Proof. By substitution, as explained above. □

To show the algorithm in action we run it over program $\mathcal{P}_3 := \langle \{\mathsf{Copy}\}, \mathsf{Copy} \rangle$ previously defined in Example 6.2.

Example 6.12. *Consider program* $\mathcal{P}_3 := \langle \{\mathsf{Copy}\}, \mathsf{Copy} \rangle$ *over signature* Σ_1 *where* Copy *has types* $x : \mathsf{nat} \vdash \mathsf{Copy} :: (y : \mathsf{nat})$.

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, s : \mathsf{nat}\},$$

$$\begin{aligned} y \leftarrow \mathsf{Copy} \leftarrow x = \mathbf{case}\, Lx\, (\mu_{nat} \Rightarrow \mathbf{case}\, Lx\, (\, & z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry \\ \mid\, & s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathsf{Copy} \leftarrow x)). \end{aligned}$$

In this example, following Definition 6.6 the programmer has to define $\mathsf{Copy} \subseteq_1 \mathsf{Copy}$ *since the only priority in* Σ *is* 1. *To verify local the guard condition for this program we run our algorithm over the definition of* Copy. *Here we show the interesting branch of the constructed derivation:*

$$
\dfrac{x^1 : 1 \vdash_{\{x_1^1<x_1^0\}} Ry^0.\mu_{nat};\cdots :: (y^0 : \mathsf{nat}) \quad \dfrac{\dfrac{\dfrac{[x_1^1, \mathtt{Copy}, y_1^1]\,(\subset, <_{\{x_1^1<x_1^0\}})\,[x_1^0, \mathtt{Copy}, y_1^0] \quad x : \mathsf{nat} \vdash \mathtt{Copy} = (\mathbf{case}\, Lx\,(\cdots))_{x,y} :: (y : \mathsf{nat}) \in V}{x^1 : \mathsf{nat} \vdash_{\{x_1^1<x_1^0\}} y^1 \leftarrow \mathtt{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})}\textsc{Call}}{x^1 : \mathsf{nat} \vdash_{\{x_1^1<x_1^0\}} Ry^1.s;\cdots :: (y^1 : 1 \oplus \mathsf{nat})}\oplus R}{x^1 : \mathsf{nat} \vdash_{\{x_1^1<x_1^0\}} Ry^0.\mu_{nat};\cdots :: (y^0 : \mathsf{nat})}\mu R}{\dfrac{x^1 : 1 \oplus \mathsf{nat} \vdash_{\{x_1^1<x_1^0\}} \mathbf{case}\, Lx^1\,(\cdots) :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_{\emptyset} \mathbf{case}\, Lx^0\,(\mu_{nat} \Rightarrow \cdots) :: (y^0 : \mathsf{nat})}\mu L}\oplus L
$$

As being checked by the Call *rule,* $[x_1^1, \mathtt{Copy}, y_1^1]\,(\subset, <_{\{x_1^1<x_1^0\}})\,[x_1^0, \mathtt{Copy}, y_1^0]$ *and the recursive call is accepted. In this particular setting in which* Copy *calls* itself *recursively, the condition of the* Call *rule can be reduced to* $[x_1^1, y_1^1] <_{\{x_1^1<x_1^0\}} [x_1^0, y_1^0]$.

Note that at a meta-level the generations on channel names and the set Ω are both used for bookkeeping purposes. We showed in this example that using the rules of Figure 6.2 as an algorithm we can annotate the given definition of a process variable with the generations and the set Ω.

6.8 Local guard condition and FS validity

Fortier and Santocanale [36] introduced a *validity condition* for identifying valid circular proofs among all infinite pre-proofs in the singleton logic with fixed points. They showed that the pre-proofs satisfying this condition, which is based on the definition of left μ- and right ν-traces, enjoy the cut elimination property. In Chapter 4, we generalized their results to the infinitary first-order intuitionistic multiplicative additive linear logic. In this section, we translate their validity condition into the context of session-typed concurrency and generalize it for subsingleton logic. It is straightforward to show that the cut elimination property holds for a proof in subsingleton logic if it satisfies the generalized version of the validity condition. The key idea is that cut reductions for individual rules stay untouched in subsingleton logic and rules for the new constant 1 only provide more options for the cut reduction algorithm to terminate. We prove that all locally guarded programs in the session typed system, determined by the algorithm in Section 6.7, also satisfy the validity condition. We conclude that our algorithm imposes a stricter but local version of validity on the session-typed programs corresponding to circular pre-proofs.

Here we adapt definitions of the *left* and *right traceable* paths, *left* μ- and *right* ν*-traces,* and then *validity* to our session type system. These definitions are phrased differently from their counterparts in Section 4.3. Here, we phrase them similar to Fortier and Santocanale's paper [36]. Theorem 6.18 proves that our earlier definition of μ- and ν-traces in Section 4.3, when restricted to the subsingleton fragment, implies the definitions as phrased here.

Definition 6.12. Consider path $\mathbb{P}$ in the (infinite) typing derivation of a program $\mathcal{Q} = \langle V, S\rangle$ defined on a signature Σ:

$$\dfrac{\dfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} Q' :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} Q :: (w^{\beta} : C)}$$

$\mathbb{P}$ is called *left traceable* if $\bar{z}$ and $\bar{x}$ are non-empty and $\bar{z} = \bar{x}$. It is called *right traceable* if $w = y$.

Moreover, $\mathbb{P}$ is called a *cycle* over program $\mathcal{Q}$, if for some $X \in V$, we have $Q = w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha}$ and $Q' = y^{\delta} \leftarrow X \leftarrow \bar{x}^{\gamma}$.

Definition 6.13. A path $\mathbb{P}$ in the (infinite) typing derivation of a program $\mathcal{Q} = \langle V, S\rangle$ defined over signature Σ is a left μ-trace if (i) it is left-traceable, (ii) there is a left fixed point rule applied on it, and (iii) the highest priority of its left fixed point rule is $i \leq n$ such that $\epsilon(i) = \mu$. Dually, $\mathbb{P}$ is a right ν-trace if (i) it is right-traceable, (ii) there is a right fixed point rule applied on it, and (iii) the highest priority of its right fixed point is $i \leq n$ such that $\epsilon(i) = \nu$.

Definition 6.14 (FS validity condition on cycles). A program $\mathcal{Q} = \langle V, S\rangle$ defined on signature Σ satisfies the FS validity condition if every cycle $\mathbb{C}$

$$\dfrac{\dfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} y^{\delta} \leftarrow X \leftarrow \bar{x}^{\gamma} :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : C)}$$

over $\mathcal{Q}$ is either a left μ-trace or a right ν-trace. Similarly, we say a single cycle $\mathbb{C}$ satisfies the validity condition if it is either a left μ-trace or a right ν-trace.

Definitions 6.12-6.14 are equivalent to the definitions of the same concepts by Fortier and Santocanale using our own notation. As an example, consider program $\mathcal{P}_3 := \langle\{\mathtt{Copy}\}, \mathtt{Copy}\rangle$ over signature Σ_1, defined in Example 6.2, where Copy has types $x : \mathsf{nat} \vdash \mathtt{Copy} :: (y : \mathsf{nat})$.

$$\Sigma_1 := \mathsf{nat} =^{1}_{\mu} \oplus\{z : 1, s : \mathsf{nat}\}$$

$$\begin{aligned} y \leftarrow \mathtt{Copy} \leftarrow x = \mathbf{case}\, Lx\ (\mu_{nat} \Rightarrow \mathbf{case}\, Lx\ (\ & z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lx; \mathbf{close}\, Ry \\ \mid\ & s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \mathtt{Copy} \leftarrow x)) \end{aligned}$$

Consider the first several steps of the derivation of the program starting with $x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathtt{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$:

$$\dfrac{\dfrac{x^1 : 1 \vdash_{\{x_1^1 < x_1^0\}} Ry^0.\mu_{nat}; \cdots :: (y^0 : \mathsf{nat}) \qquad \dfrac{\dfrac{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} y^1 \leftarrow \mathsf{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})}{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} Ry^1.s; \cdots :: (y^1 : 1 \oplus \mathsf{nat})}\oplus R}{x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} Ry^0.\mu_{nat}; \cdots :: (y^0 : \mathsf{nat})}\mu R}{\dfrac{x^1 : 1 \oplus \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} \mathbf{case}\, Lx^1\ (\cdots) :: (y^0 : \mathsf{nat})}{\dfrac{x^0 : \mathsf{nat} \vdash_{\emptyset} \mathbf{case}\, Lx^0\ (\mu_{nat} \Rightarrow \cdots) :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}(\mathsf{Copy})}\mu L}\oplus L$$

The path between

$$x^0 : \mathsf{nat} \vdash_{\emptyset} y^0 \leftarrow \mathsf{Copy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$$

and

$$x^1 : \mathsf{nat} \vdash_{\{x_1^1 < x_1^0\}} y^1 \leftarrow \mathsf{Copy} \leftarrow x^1 :: (y^1 : \mathsf{nat})$$

is by definition both left traceable and right traceable, but it is only a left μ-trace and not a right ν-trace: the highest priority of a fixed point applied on the left-hand side on this path belongs to a positive type; this application of the μL rule added $x_1^1 < x_1^0$ to the set defining the $<$ order. However, there is no negative fixed point rule applied on the right, and y_1^1 and y_1^0 are incomparable to each other.

This cycle satisfies the *validity condition* by being a left μ-trace. We showed in Example 6.12 that it is also accepted by our algorithm since $list(x^1, y^1) = [(x_1^1, y_1^1)] < [(x_1^0, y_1^0)] = list(x^0, y^0)$.

Here, we can observe that being a left μ-trace coincides with having the relation $x_1^1 < x_1^0$ between the left channels, and not being a right ν-trace coincides with not having the relation $y_1^1 < y_1^0$ for the right channels. We can generalize this observation to every path and every signature with n priorities.

Definition 6.15. Consider a signature Σ and a channel x^γ. We define the *snapshot* of a channel x^α as a list $\mathsf{snap}(x^\alpha) = [\mathbf{x}_i^\alpha]_{i \le n} = [x_1^\gamma, \cdots, x_n^\gamma]$, where n is the maximum priority in Σ. For brevity, we write $[x^\gamma]$ instead of $\mathsf{snap}(x^\alpha)$.

As explained in Section 6.3, the reflexive transitive closure of Ω in judgment $x^\gamma : \omega \vdash_\Omega P :: (y^\delta : C)$ forms a partial order $\le_\Omega$. To enhance readability of proofs, throughout this section we may use entailment $\Omega \Vdash x \le y$ instead of $x \le_\Omega y$.

Lemma 6.16. *Consider a finite path $\mathbb{P}$ in the (infinite) typing derivation of a program $\mathcal{Q} = \langle V, S \rangle$ defined on a signature Σ,*

$$\begin{array}{c} x^\gamma : \omega' \vdash_{\Omega'} P' :: (y^\delta : C') \\ \hline \vdots \\ \hline z^\alpha : \omega \vdash_\Omega P :: (w^\beta : C) \end{array}$$

with n the maximum priority in Σ.

(a) *For every $i \in \mathtt{c}(\omega')$ with $\epsilon(i) = \mu$, if $x_i^\gamma \leq_{\Omega'} z_i^\alpha$ then $x = z$ and $i \in \mathtt{c}(\omega)$.*

(b) *For every $i < n$, if $x_i^\gamma <_{\Omega'} z_i^\alpha$, then $i \in \mathtt{c}(\omega)$ and a μL rule with priority i is applied on $\mathbb{P}$.*

(c) *For every $c \leq n$ with $\epsilon(c) = \nu$, if $x_c^\gamma \leq_{\Omega'} z_c^\alpha$, then no νL rule with priority c is applied on $\mathbb{P}$.*

Proof. Proof is by induction on the structure of $\mathbb{P}$. We consider each case for last (**topmost**) step in $\mathbb{P}$. The judgment $x^\gamma : \omega' \vdash_{\Omega'} P' :: (y^\delta : C')$ is a premise of the last step.

Case

$$\frac{x^\gamma : \omega' \vdash_{\Omega'} P' :: (y^\delta : C') \quad x : \omega' \vdash X = P' :: (y : C') \in V}{x^\gamma : \omega' \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow x^\gamma :: (y^\delta : C')}\textsc{Def}(X)$$

None of the conditions in the conclusion are different from the premise. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case We need to distinguish two cases for the cut rule, since the typing judgment for P' can be the the first or left premise of Cut.

Subcase.

$$\frac{x^\gamma : \omega' \vdash_{\Omega''\cup\mathtt{r}(v^\theta)} P'_{y^0} :: (y^0 : C') \quad y^0 : C' \vdash_{\Omega''\cup\mathtt{r}(x^\gamma)} Q_{y^0} :: (v^\theta : C'')}{x^\gamma : \omega' \vdash_{\Omega''} (y \leftarrow P'_y; Q_y) :: (v^\theta : C'')}\textsc{Cut}^y$$

where $\mathtt{r}(u) = \{y_j^0 = u_j \mid j \notin \mathtt{c}(C') \,\text{and}\, j \leq n\}$ and $\Omega' = \Omega'' \cup r(v^\theta)$. All conditions in the conclusion are the same as the first premise ($x^\gamma : \omega' \vdash_{\Omega''} (y \leftarrow P'_y; Q_y) :: (v^\theta : C'')$): the equations in $r(v^\theta)$ only include channels y^0 and v^θ. As a result $\Omega'' \cup \mathtt{r}(v^\theta) \Vdash x_i^\gamma \leq z_i^\alpha$ implies $\Omega'' \Vdash x_i^\gamma \leq z_i^\alpha$, and $\Omega'' \cup \mathtt{r}(v^\theta) \Vdash x_i^\gamma < z_i^\alpha$ implies $\Omega'' \Vdash x_i^\gamma < z_i^\alpha$. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Subcase.

$$\frac{u^\eta : \omega'' \vdash_{\Omega''\cup\mathtt{r}(y^\delta)} Q_{a^0} :: (x^0 : A) \quad x^0 : A \vdash_{\Omega''\cup\mathtt{r}(u^\eta)} P'_{x^0} :: (y^\delta : C'')}{u^\eta : \omega'' \vdash_{\Omega''} (x \leftarrow Q_x; P'_x) :: (y^\delta : C')}\textsc{Cut}^x$$

where $\mathtt{r}(v) = \{x_j^0 = v_j \mid j \notin \mathtt{c}(A) \,\text{and}\, j \leq n\}$ and $\Omega' = \Omega'' \cup r(u^\eta)$.

(a) $\Omega'' \cup r(u^\eta) \Vdash x_i^0 \leq z_i^\alpha$ does not hold for any $i \in c(A)$: x is a fresh channel and does not occur in the equation of Ω''. Moreover, since $i \in c(A)$, there is no equation in the set $r(u^\eta)$ including x_i^0. Therefore, this part is vacuously true.

(b) By freshness of x, if $\Omega'' \cup r(u^\eta) \Vdash x_i^0 < z_i^\alpha$, then $x_i^0 = u_i^\eta \in r(u^\eta)$ and $\Omega'' \Vdash u_i^\eta < z_i^\alpha$. By the induction hypothesis, $i \in \mathtt{c}(\omega)$ and a μL rule with priority i is applied on $\mathbb{P}$.

(c) By freshness of x, if $\Omega'' \cup r(u^\eta) \Vdash x_c^0 \leq z_c^\alpha$, then $x_c^0 = u_c^\eta \in r(u^\eta)$ and $\Omega'' \Vdash u_c^\eta \leq z_c^\alpha$. By the induction hypothesis, no νL rule with priority c is applied on $\mathbb{P}$.

Case

$$\frac{. \vdash_{\Omega'} P' :: (y^\delta : C')}{u^\eta : 1 \vdash_{\Omega'} \textbf{wait}\, Lu^\eta; P' :: (y^\delta : C')}\, 1L$$

This case is not applicable since by the typing rules $\Omega' \not\Vdash . \leq z_i^\alpha$ for any $i \leq n$.

Case

$$\frac{x^\alpha : \omega' \vdash_{\Omega'} P :: (y^{\delta'+1} : C') \quad t =_\mu C' \quad \Omega' = \Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\}}{x^\alpha : \omega' \vdash_{\Omega''} Ry^{\delta'}.\mu_t; P :: (y^{\delta'} : t)}\, \mu R$$

For every $i \leq n$, if $\Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\} \Vdash x_i^\gamma \leq z_i^\alpha$, then $\Omega'' \Vdash x_i^\gamma \leq z_i^\alpha$. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case

$$\frac{\begin{matrix} t =_\mu \omega' \\ x^{\gamma'+1} : \omega' \vdash_{\Omega'} P' :: (y^\delta : C') \quad \Omega' = \Omega'' \cup \{x_{p(t)}^{\gamma'+1} < x_{p(t)}^{\gamma'}\} \cup \{x_{p(s)}^{\gamma'+1} = x_{p(s)}^{\gamma'} \mid p(s) \neq p(t)\} \end{matrix}}{x^{\gamma'} : t \vdash_{\Omega''} \textbf{case}\, Lx^{\gamma'}\, (\mu_t \Rightarrow P') :: (y^\delta : C')}\, \mu L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(\omega') \subseteq \mathsf{c}(t)$. By μL rule, for all $i \leq n$, $x_i^{\gamma'+1} \leq x_i^{\gamma'} \in \Omega'$. But by freshness of channels and their generations, $x^{\gamma'+1}$ is not involved in any relation in Ω''.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon(i) = \mu$, if $\Omega' \Vdash x_i^{\gamma'+1} \leq z_i^\alpha$ then $\Omega'' \Vdash x_i^{\gamma'} \leq z_i^\alpha$. By the induction hypothesis, we have $x = z$ and $i \in \mathsf{c}(\omega)$.

(b) We consider two subcases: (1) If $\Omega' \Vdash x_i^{\gamma'+1} < z_i^\alpha$ for $i \neq p(t)$, then $\Omega' \Vdash x_i^{\gamma'+1} = x_i^{\gamma'}$ and $\Omega'' \Vdash x_i^{\gamma'} < z_i^\alpha$. Now we can apply the induction hypothesis. (2) If $\Omega' \Vdash x_{p(t)}^{\gamma'+1} < z_{p(t)}^\alpha$, then $\Omega' \Vdash x_{p(t)}^{\gamma'+1} < x_{p(t)}^{\gamma'}$ and $\Omega'' \Vdash x_{p(t)}^{\gamma'} \leq z_{p(t)}^\alpha$. Since a μL rule is applied in this step on the priority $p(t)$, we only need to prove that $p(t) \in \mathsf{c}(\omega)$. By definition of c, we have $p(t) \in \mathsf{c}(t)$ and we can use the induction hypothesis on part (a) to get $p(t) \in \mathsf{c}(\omega)$.

(c) For every $c \leq n$ with $\epsilon(c) = \nu$, $\Omega' \Vdash x_c^{\gamma'+1} = x_c^{\gamma'}$ as $c \neq p(t)$. Therefore, if $\Omega' \Vdash x_c^{\gamma'+1} = z_c^\alpha$, then $\Omega'' \Vdash x_c^{\gamma'} = z_c^\alpha$. By the induction hypothesis no νL rule with priority c is applied on $\mathbb{P}$.

Case

$$\frac{\begin{matrix} t =_\nu C' \\ x^\gamma : \omega' \vdash_{\Omega'} P' :: y^{\delta'+1} : C' \quad \Omega' = \Omega'' \cup \{y_{p(t)}^{\delta'+1} < y_{p(t)}^{\delta'}\} \cup \{y_{p(s)}^{\delta'+1} = y_{p(s)}^{\delta'} \mid p(s) \neq p(t)\} \end{matrix}}{x^\gamma : \omega' \vdash_{\Omega''} \textbf{case}\, Ry^{\delta'}\, (\nu_t \Rightarrow P') :: (y^{\delta'} : t)}\, \nu R$$

For every $i \leq n$, if $\Omega'' \cup \{y^{\delta'+1}_{p(t)} < y^{\delta'}_{p(t)}\} \cup \{y^{\delta'+1}_{p(s)} = y^{\delta'}_{p(s)} \mid p(s) \neq p(t)\} \Vdash x^{\gamma}_i \leq z^{\alpha}_i$, then $\Omega'' \Vdash x^{\gamma}_i \leq z^{\alpha}_i$. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case

$$\frac{x^{\gamma'+1} : \omega' \vdash_{\Omega'} Q :: (y^{\delta} : C') \quad t =_{\nu} \omega' \quad \Omega' = \Omega'' \cup \{x^{\gamma'+1}_{p(s)} = x^{\gamma'}_{p(s)} \mid p(s) \neq p(t)\}}{x^{\gamma'} : t \vdash_{\Omega''} Lx^{\gamma'}.\nu_t; P' :: (y^{\delta} : C')}\nu L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(\omega') \subseteq \mathsf{c}(t)$. By νL rule, for all $i \neq p(t) \leq n$, $x^{\gamma'+1}_i = x^{\gamma'}_i \in \Omega'$. In particular, for every $i \leq n$ with $\epsilon(i) = \mu$, $x^{\gamma'+1}_i = x^{\gamma'}_i \in \Omega'$. But by freshness of channels and their generations, $x^{\gamma'+1}$ is not involved in any relation in Ω''.

(a) For every $i \in \mathsf{c}(\omega')$ with $\epsilon(i) = \mu$, if $\Omega' \Vdash x^{\gamma'+1}_i \leq z^{\alpha}_i$, then $\Omega' \Vdash x^{\gamma'+1}_i = x^{\gamma'}_i$ and $\Omega'' \Vdash x^{\gamma'}_i \leq z^{\alpha}_i$. By the induction hypothesis $x = z$ and $i \in \mathsf{c}(\omega)$.

(b) If $\Omega' \Vdash x^{\gamma'+1}_i < z^{\alpha}_i$, then by freshness of channels and their generations we have $i \neq p(t)$, $\Omega' \Vdash x^{\gamma'+1}_i = x^{\gamma'}_i$ and $\Omega'' \Vdash x^{\gamma'}_i < z^{\alpha}_i$. By the induction hypothesis $i \in \mathsf{c}(\omega)$ and a μL rule with priority i is applied on the path.

(c) For every $c \leq n$ with $\epsilon(c) = \nu$ and $c \neq p(t)$, if $\Omega' \Vdash x^{\gamma'+1}_c \leq z^{\alpha}_c$, then $\Omega' \Vdash x^{\gamma'+1}_c = x^{\gamma'}_c$ and $\Omega'' \Vdash x^{\gamma'} \leq z^{\alpha}_c$. Therefore, by induction hypothesis, no νL rule with priority c is applied on the path. Note that $\Omega' \not\Vdash x^{\gamma'+1}_p(t) \leq z^{\alpha}_p(t)$.

Case

$$\frac{x^{\gamma} : \omega \vdash_{\Omega'} Q_k :: y^{\delta} : A_k \quad \forall k \in L}{x^{\gamma} : \omega \vdash_{\Omega'} \textbf{case}\, Ry^{\delta}\ (\ell \Rightarrow Q_\ell) :: (y^{\delta} : \&\{\ell : A_\ell\}_{\ell \in L})}\&R$$

None of the conditions in the conclusion are different from the premise. Therefore, by the induction hypothesis, statements (a)-(c) hold.

Case

$$\frac{x^{\gamma} : A_k \vdash_{\Omega'} Q :: (y^{\delta} : C')}{x^{\gamma} : \&\{\ell : A_\ell\}_{\ell \in L} \vdash_{\Omega'} Lx^{\gamma}.k; Q :: (y^{\delta} : C')}\&L$$

By definition of $\mathsf{c}(x)$, we have $\mathsf{c}(A_k) \subseteq \mathsf{c}(\&\{\ell : A_\ell\}_{\ell \in L})$. Therefore, statements (a)-(c) follow from the induction hypothesis.

Cases The statements are trivially true if the last step of the proof is either $1R$ or Id rules.

□

Lemma 6.17. *Consider a path $\mathbb{P}$ in the (infinite) typing derivation of a program $\mathcal{Q} = \langle V, S\rangle$ defined on a signature Σ,*

$$\cfrac{\cfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} P' :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} P :: (w^{\beta} : C)}$$

with n the maximum priority in Σ.

(a) *For every $i \in \mathsf{c}(\omega')$ with $\epsilon(i) = \nu$, if $y_i^\delta \leq_{\Omega'} w_i^\beta$, then $y = w$ and $i \in \mathsf{c}(\omega)$.*

(b) *If $y_i^\delta <_{\Omega'} w_i^\beta$, then $i \in \mathsf{c}(\omega)$ and a νL rule with priority i is applied on $\mathbb{P}$.*

(c) *For every $c \leq n$ with $\epsilon(c) = \mu$, if $y_c^\delta \leq_{\Omega'} w_c^\beta$, then no μR rule with priority c is applied on $\mathbb{P}$.*

Proof. Dual to the proof of Lemma 6.16. □

Theorem 6.18. *A cycle $\mathbb{C}$*

$$\dfrac{\bar{x}^\gamma : \omega' \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow \bar{x}^\gamma :: (y^\delta : C')}{\dfrac{\vdots}{\bar{z}^\alpha : \omega \vdash_{\Omega} w^\beta \leftarrow X \leftarrow \bar{z}^\alpha :: (w^\beta : C)}}$$

on a program $\mathcal{Q} = \langle V, S\rangle$ defined over signature Σ is a left μ-trace if $\bar{x}$ and $\bar{z}$ are non-empty and the list $[x^\gamma] = [x_1^\gamma, \cdots, x_n^\gamma]$ is lexicographically less than the list $[z^\alpha] = [z_1^\alpha, \cdots, z_n^\alpha]$ by the order $<_{\Omega'}$ built in Ω'. Dually, it is a right ν-trace, if the list $[y^\delta] = [y_1^\delta, \cdots, y_n^\delta]$ is lexicographically less than the list $[w^\beta] = [w_1^\beta, \cdots, w_n^\beta]$ by the strict order $<_{\Omega'}$ built in Ω'

Proof. This theorem is a corollary of Lemmas 6.16 and 6.17. □

We provide a few additional examples to elaborate Theorem 6.18 further. Define a program $\mathcal{P}_9 := \langle\{\texttt{Succ}, \texttt{Copy}, \texttt{SuccCopy}\}, \texttt{SuccCopy}\rangle$, over the signature Σ_1, using the process $w : \mathsf{nat} \vdash \texttt{Copy} :: (y : \mathsf{nat})$ and two other processs: $x : \mathsf{nat} \vdash \texttt{Succ} :: (w : \mathsf{nat})$ and $x : \mathsf{nat} \vdash \texttt{SuccCopy} :: (y : \mathsf{nat})$. The processes are defined as

$$w \leftarrow \texttt{Succ} \leftarrow x = Rw.\mu_{nat}; Rw.s; w \leftarrow x$$

$$\begin{aligned} y \leftarrow \texttt{Copy} \leftarrow w = \textbf{case}\, Lw\ (\mu_{nat} \Rightarrow \textbf{case}\, Lw\ (\ & s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \texttt{Copy} \leftarrow w \\ \mid\ & z \Rightarrow Ry.\mu_{nat}; Ry.z; \textbf{wait}\, Lw; \textbf{close}\, Ry)) \end{aligned}$$

$$y \leftarrow \texttt{SuccCopy} \leftarrow x = w \leftarrow \texttt{Succ} \leftarrow x; y \leftarrow \texttt{Copy} \leftarrow w,$$

Process SuccCopy spawns a new process Succ and continues as Copy. The Succ process prepends an s label to the beginning of the finite string representing a natural number on its left hand side and then forwards the string as a whole to the right. Copy receives this finite string representing a natural number, and *forwards* it to the right label by label.

The only recursive process in this program is Copy. So program $\mathcal{P}_9$, itself, does not have a further interesting point to discuss. We consider a bogus version of this program in Example 6.13 that provides further intuition for Theorem 6.18.

Example 6.13. *Define program* $\mathcal{P}_{10} := \langle\{\texttt{Succ}, \texttt{BogusCopy}, \texttt{SuccCopy}\}, \texttt{SuccCopy}\rangle$ *over the signature*

$$\Sigma_1 := \mathsf{nat} =^1_\mu \oplus\{z : 1, \;\; s : \mathsf{nat}\},$$

The processes $x : \mathsf{nat} \vdash \texttt{Succ} :: (w : \mathsf{nat})$, $w : \mathsf{nat} \vdash \texttt{BogusCopy} :: (y : \mathsf{nat})$, *and* $x : \mathsf{nat} \vdash \texttt{SuccCopy} :: (y : \mathsf{nat})$, *are defined as*

$$w \leftarrow \texttt{Succ} \leftarrow x = Rw.\mu_{nat}; Rw.s; w \leftarrow x$$

$$y \leftarrow \texttt{BogusCopy} \leftarrow w = \mathbf{case}\, Lw\, (\mu_{nat} \Rightarrow \mathbf{case}\, Lw\, (\, s \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \texttt{SuccCopy} \leftarrow w \\ \mid z \Rightarrow Ry.\mu_{nat}; Ry.z; \mathbf{wait}\, Lw; \mathbf{close}\, Ry))$$

$$y \leftarrow \texttt{SuccCopy} \leftarrow x = w \leftarrow \texttt{Succ} \leftarrow x; y \leftarrow \texttt{BogusCopy} \leftarrow w$$

Program $\mathcal{P}_{10}$ *is a non-reactive* bogus *program, since* BogusCopy *instead of calling itself recursively, calls* SuccCopy*. At the very beginning* SuccCopy *spawns* Succ *and continues with* BogusCopy *for a fresh channel* w*.* Succ *then sends a fixed point unfolding message and a successor label via* w *to the right, while* BogusCopy *receives the two messages just sent by* Succ *through* w *and calls* SuccCopy *recursively again. This loop continues forever, without any messages being received from the outside.*

The first several steps of the derivation of $x^0 : \mathsf{nat} \vdash_\emptyset \texttt{SuccCopy} :: (y^0 : \mathsf{nat})$ *in our inference system (Section 6.6) are given below.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x^0 : \mathsf{nat} \vdash_\emptyset w^1 \leftarrow x^0 :: (w^1 : \mathsf{nat})}\textsc{Id}}{x^0 : \mathsf{nat} \vdash_\emptyset Rw^1.s; \cdots :: (w^1 : 1 \oplus \mathsf{nat})}\oplus R}{x^0 : \mathsf{nat} \vdash_\emptyset Rw^0.\mu_{nat}; \cdots :: (w^0 : \mathsf{nat})}\mu R}{x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \texttt{Succ} \leftarrow x^0 :: (w^0 : \mathsf{nat})}\textsc{Def} \quad \dfrac{\dfrac{\dfrac{\cdots \quad w^1 : \mathsf{nat} \vdash_{\{w^1_1 < w^0_1\}} y^0 \leftarrow \texttt{SuccCopy} \leftarrow w^1 :: (y^0 : \mathsf{nat})}{w^1 : 1 \oplus \mathsf{nat} \vdash_{\{w^1_1 < w^0_1\}} \mathbf{case}\, Lw^1\, (\cdots) :: (y^0 : \mathsf{nat})}\oplus L}{w^0 : \mathsf{nat} \vdash_\emptyset \mathbf{case}\, Lw^0\, (\mu_{nat} \Rightarrow \cdots) :: (y^0 : \mathsf{nat})}\mu L}{w^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \texttt{BogusCopy} \leftarrow w^0 :: (y^0 : \mathsf{nat})}\textsc{Def}}{\dfrac{x^0 : \mathsf{nat} \vdash_\emptyset w \leftarrow \texttt{Succ}; y^0 \leftarrow \texttt{BogusCopy} \leftarrow w :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \texttt{SuccCopy} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}}\textsc{Cut}^w$$

Consider the cycle between

$$x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \texttt{SuccCopy} \leftarrow x^0 :: (y^0 : \mathsf{nat})$$

and

$$w^1 : \mathsf{nat} \vdash_{\{w^1_1 < w^0_1\}} y^0 \leftarrow \texttt{SuccCopy} \leftarrow w^1 :: (y^0 : \mathsf{nat}).$$

By Definition 6.13, this path is right traceable, but not left traceable. And by Definition 6.12, the path is neither a right ν-trace nor a left μ-trace:

1. *No negative fixed point unfolding message is received from the right and y^0 does not evolve to a new generation that has a smaller value in its highest priority than y_1^0. In other words, $y_1^0 \not< y_1^0$ since no negative fixed point rule has been applied on the right channel.*
2. *The positive fixed point unfolding message that is received from the left is received through the channel w^0, which is a fresh channel created after* `SuccCopy` *spawns the process* `Succ`*. Although $w_1^1 < w_1^0$, since x_1^0 is incomparable to w_1^0, the relation $w_1^1 < x_1^0$ does not hold. This path is not even a left-traceable path.*

Neither $[w^1] = [w_1^1] < [x_1^0] = [x^0]$, nor $[y^0] = [y_1^0] < [y_1^0] = [y^0]$ hold, and this cycle does not satisfy the validity condition. This program is not locally guarded either since $[w_1^1, y_1^0] \not< [x_1^0, y_1^0]$.

As another example consider the program $\mathcal{P}_6 = \{\texttt{Ping}, \texttt{Pong}, \texttt{PingPong}\}, \texttt{PingPong}\rangle$ over the signature Σ_4 as defined in Example 6.5. We discussed in Section 6.3 that this program is not accepted by our algorithm as locally guarded.

Example 6.14. *Recall the definition of signature Σ_4:*

$$\begin{aligned}\Sigma_4 := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\ \ tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\ \ s : \mathsf{nat}\}\end{aligned}$$

Processes

$$\begin{aligned}&x : \mathsf{nat} \vdash \texttt{Ping} :: (w : \mathsf{astream}),\\ &w : \mathsf{astream} \vdash \texttt{Pong} :: (y : \mathsf{nat}),\\ &x : \mathsf{nat} \vdash \texttt{PingPong} :: (y : \mathsf{nat})\end{aligned}$$

are defined as

$$\begin{aligned}w \leftarrow \texttt{Ping} \leftarrow x = \mathbf{case}\, Rw\, (\nu_{astream} \Rightarrow \mathbf{case}\, Rw\, (\ &head \Rightarrow Rw.\mu_{ack}; Rw.ack; w \leftarrow \texttt{Ping} \leftarrow x\\ \mid\ &tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x))\end{aligned}$$

$$\begin{aligned}y \leftarrow \texttt{Pong} \leftarrow w =\ &Lw.\nu_{astream}; Lw.head;\\ &\mathbf{case}\, Lw\, (\mu_{ack} \Rightarrow \mathbf{case}\, Lw\, (ack \Rightarrow Ry.\mu_{nat}; Ry.s; y \leftarrow \texttt{Pong} \leftarrow w))\end{aligned}$$

$$y \leftarrow \texttt{PingPong} \leftarrow x = w \leftarrow \texttt{Ping} \leftarrow x; y \leftarrow \texttt{Pong} \leftarrow w$$

The first several steps of the proof of $x^0 : nat \vdash_\emptyset$ `PingPong` $:: (y^0 : nat)$ in our inference system (Section 6.6) are given below (with some abbreviations).

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x^0 : \mathsf{nat} \vdash_B w^2 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^2 : \mathsf{astream})}{x^0 : \mathsf{nat} \vdash_B Rw^2.ack; \cdots :: (w^2 : \oplus\{\mathsf{astream}\})}\oplus R}{x^0 : \mathsf{nat} \vdash_A Rw^1.\mu_{ack}; \cdots :: (w^1 : \mathsf{ack})}\mu R \quad x^0 : \mathsf{nat} \vdash_A \cdots :: (w^1 : \mathsf{astream})}{x^0 : \mathsf{nat} \vdash_A \mathbf{case}\, Rw^1\ (\cdots) :: (w^1 : \mathsf{ack} \,\&\, \mathsf{astream})}\&R}{x^0 : \mathsf{nat} \vdash_\emptyset \mathbf{case}\, Rw^0\ (\nu_{astream} \Rightarrow \cdots) :: (w^0 : \mathsf{astream})}\nu R}{x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^0 : \mathsf{astream})}\textsc{Def} \quad w^0 : \mathsf{astream} \vdash_\emptyset \cdots :: (y^0 : \mathsf{nat})}{\dfrac{x^0 : \mathsf{nat} \vdash_\emptyset w \leftarrow \texttt{Ping} \leftarrow x^0; y^0 \leftarrow \texttt{Pong} \leftarrow w :: (y^0 : \mathsf{nat})}{x^0 : \mathsf{nat} \vdash_\emptyset y^0 \leftarrow \texttt{PingPong} \leftarrow x^0 :: (y^0 : \mathsf{nat})}\textsc{Def}}\textsc{Cut}
$$

where $A = \{w^1_1 = w^0_1, w^1_2 < w^0_2, w^1_3 = w^0_3\}$*, and* $B = \{w^1_1 = w^0_1, w^2_2 = w^1_2 < w^0_2, w^2_3 = w^1_3 = w^0_3\}$*. The cycle between the processes*

$$x^0 : \mathsf{nat} \vdash_\emptyset w^0 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^0 : \mathsf{astream})$$

and

$$x^0 : \mathsf{nat} \vdash_B w^2 \leftarrow \texttt{Ping} \leftarrow x^0 :: (w^2 : \mathsf{astream})$$

is neither a left μ*-trace, nor a right* ν*-trace:*

1. *No fixed point unfolding message is received or sent through the left channels in this path and thus* $[x^0] = [x^0_1, x^0_2, x^0_3] \not< [x^0_1, x^0_2, x^0_3] = [x^0]$.

2. *On the right, fixed point unfolding messages are both sent and received: (i)* w^0 *receives an unfolding message for a negative fixed point with priority* 2 *and evolves to* w^1*, and then later (ii)* w^1 *sends an unfolding message for a positive fixed point with priority* 1 *and evolves to* w^2*. But the positive fixed point has a higher priority than the negative fixed point, and thus this path is not a right* ν*-trace either.*

This reasoning can also be reflected in our observation about the list of channels in Theorem 6.18: When, first, w^0 *evolves to* w^1 *by receiving a message in (i) the relations* $w^1_1 = w^0_1$*,* $w^1_2 < w^0_2$*, and* $w^1_3 = w^0_3$ *are recorded. And, later, when* w^1 *evolves to* w^2 *by sending a message in (ii) the relations* $w^2_2 = w^1_2$*, and* $w^2_3 = w^1_3$ *are added to the set. This means that* w^2_1 *as the first element of the list* $[w^2]$ *remains incomparable to* w^0_1 *and thus* $[w^2] = [w^2_1, w^2_2, w^2_3] \not< [w^0_1, w^0_2, w^0_3] = [w^0]$.

By Theorem 6.18, a cycle $\mathbb{C}$

$$\dfrac{\dfrac{\bar{x}^\gamma : \omega' \vdash_{\Omega'} y^\delta \leftarrow X \leftarrow \bar{x}^\gamma :: (y^\delta : C')}{\vdots}}{\bar{z}^\alpha : \omega \vdash_\Omega w^\beta \leftarrow X \leftarrow \bar{z}^\alpha :: (w^\beta : C)}$$

is either a *left μ-trace* or a *right ν-trace* if either $[x^\gamma] <_{\Omega'} [z^\alpha]$ **or** $[y^\delta] <_{\Omega'} [w^\beta]$ holds. Checking a disjunctive condition for each cycle implies that the FS validity condition cannot simply analyze each path from the beginning of a definition to a call site in isolation and then compose the results—instead it must unfold the definitions and examine every possible cycle in the infinitary derivation separately.

In our algorithm, however, we merge the lists of left and right channels, e.g. $[x^\gamma]$ and $[y^\delta]$ respectively, into a single list $list(x^\gamma, y^\delta)$. The values in $list(x^\gamma, y^\delta)$ from Definition 6.4 are still recorded in their order of priorities, but for the same priority the value corresponding to *receiving* a message precedes the one corresponding to *sending* a message. As described in Definition 6.8 we merge this list with process variables to check all (immediate) calls even those that do not form a cycle in the sense of the FS validity condition (that is, when process X calls process $Y \neq X$).

By adding process variables to our guard condition there is no need to search for every possible cycle in the infinitary derivation. Instead, our algorithm only checks the condition for the immediate calls that a process makes. As this condition enjoys transitivity, it also holds for all possible non-immediate recursive calls, including any cycles.

Remark 6.19. We briefly analyze the asymptotic complexity of our algorithm. Let n be the number of priorities and s the size of the signature, where we add in the sizes of all types A appearing in applications of the Cut rule. In time $O(n\, s)$ we can compute a table to look up $i \in c(A)$ for all priorities i and types A appearing in cuts.

Now let m be the size of the program (not counting the signature). We traverse each process definition just once, maintaining a list of relations between the current and original channel pairs for each priority. We need to update at most $2n$ entries in the list at each step and compare at most $2n$ entries at each Call rule. Furthermore, for each Cut rule we have a constant-time table lookup to determine if $i \in c(A)$ for each priority i. Therefore, analysis of the process definitions takes time $O(m\, n)$.

Putting it all together, the time complexity is bounded by $O(m\, n + n\, s) = O(n\,(m + s))$. In practice the number of priorities, n, is a small constant so checking the guard condition is linear in the total input, which consists of the signature and the process definitions. As far as we are aware of, the best upper bound for the complexity of the FS validity condition is $PSPACE$ [33].

It is also interesting to note that the complexity of type-checking itself is bounded below by $O(m + s^2)$ since, in the worst case, we need to compute equality between each pair of types. That is, checking the guard condition is faster than type-checking.

Another advantage of locality derives from the fact that our algorithm checks each process definition independent of the rest of the program: we can safely reuse a previously checked locally guarded process in other programs defined over the same signature and order $\subset$ without the need to verify the local guard condition again.

We are now ready to state our main theorem that proves the local guard algorithm introduced in Section 6.7 is *stricter* than the FS validity condition. Since the FS validity condition is defined over an infinitary system, we need to first map our local condition into the infinitary calculus given in Figure 6.1.

Lemma 6.20. *Consider a path $\mathbb{P}$ on a program $\mathcal{Q} = \langle V, S\rangle$ defined on a signature Σ, with n the maximum priority in Σ.*

$$\cfrac{\cfrac{\bar{x}^{\gamma} : \omega' \vdash_{\Omega'} P' :: (y^{\delta} : C')}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} P :: (w^{\beta} : C)}$$

Ω' preserves the (in)equalities in Ω. In other words, for channels u, v, generations $\eta, \eta' \in \mathbb{N}$ and type priorities $i, j \leq n$,

(a) If $\Omega \Vdash u_i^{\eta} < v_j^{\eta'}$, then $\Omega' \Vdash u_i^{\eta} < v_j^{\eta'}$.

(b) If $\Omega \Vdash u_i^{\eta} \leq v_j^{\eta'}$, then $\Omega' \Vdash u_i^{\eta} \leq v_j^{\eta'}$.

(c) If $\Omega \Vdash u_i^{\eta} = v_j^{\eta'}$, then $\Omega' \Vdash u_i^{\eta} = v_j^{\eta'}$.

Proof. Proof is by induction on the structure of $\mathbb{P}$. We consider each case for topmost step in $\mathbb{P}$. Here, we only give one non-trivial case. The proof of other cases is similar.

Case

$$\frac{x^{\alpha} : \omega' \vdash_{\Omega'} P :: (y^{\delta'+1} : C') \quad t =_{\mu} C' \quad \Omega' = \Omega'' \cup \{(y^{\delta'})_{p(s)} = (y^{\delta'+1})_{p(s)} \mid p(s) \neq p(t)\}}{x^{\alpha} : \omega' \vdash_{\Omega''} R y^{\delta'}.\mu_t; P :: (y^{\delta'} : t)}\mu R$$

(a) If $\Omega \Vdash u_i^{\eta} < v_j^{\eta'}$, then by the inductive hypothesis, $\Omega'' \Vdash u_i^{\eta} < v_j^{\eta'}$. By freshness of channels and their generations, we know that $y^{\delta'+1}$ does not occur in any (in)equalities in Ω'' and thus $y^{\delta'+1} \neq u^{\eta}, v^{\eta'}$. Therefore $\Omega' \Vdash u_i^{\eta} < v_j^{\eta'}$.

Following the same reasoning, we can prove statements (b) and (c).

□

Lemma 6.21. *Consider a finitary derivation (Figure 6.2) for*

$$\langle \bar{u}, X, v\rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega,\subset} P :: (y^{\beta} : C),$$

on a locally guarded program $\mathcal{Q} = \langle V, S\rangle$ defined on signature Σ and order $\subset$. There is a (potentially infinite) derivation $\mathbb{D}$ for

$$\bar{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : C),$$

in the infinitary system of Figure 6.1.

Moreover, for every $\bar{w}^{\gamma} : \omega' \vdash_{\Omega'} z^{\delta} \leftarrow Y \leftarrow \bar{w}^{\gamma} :: (z^{\delta} : C')$ *on* $\mathbb{D}$*, we have*

$$Y, \mathit{list}(\bar{w}^{\gamma}, z^{\delta})\ (\subset, <_{\Omega'})\ X, \mathit{list}(\bar{x}^{\alpha}, y^{\beta}).$$

Proof. We prove this by coinduction, producing the derivation of $\bar{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : C)$. We proceed by case analysis of the first rule applied on $\langle \bar{u}, X, v \rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega} P :: (y^{\beta} : C)$, in its finite derivation.

Case

$$\dfrac{Y, \mathit{list}(\bar{x}^{\alpha}, y^{\beta})\ (\subset, <_{\Omega})\ X, \mathit{list}(\bar{u}^{\gamma}, v^{\delta}) \quad \bar{x} : \omega \vdash Y = P'_{\bar{x},y} :: (y : C) \in V}{\langle \bar{u}^{\gamma}, X, v^{\delta} \rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega,\subset} y^{\beta} \leftarrow Y \leftarrow \bar{x}^{\alpha} :: (y^{\beta} : C)}\textsc{Call}(Y)$$

The program is guarded, so there is a finitary derivation for

$$\langle \bar{x}^{0}, Y, y^{0} \rangle; \bar{x}^{0} : \omega \vdash_{\emptyset,\subset} P'_{\bar{x}^{0},y^{0}} :: (y^{0} : C).$$

Having Proposition 6.11 and freshness of future generations of channels in Ω, there is also a finitary derivation for

$$\langle \bar{x}^{\alpha}, Y, y^{\beta} \rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega,\subset} P'_{\bar{x}^{\alpha},y^{\beta}} :: (y^{\beta} : C).$$

We apply the coinductive hypothesis to get an infinitary derivation $\mathbb{D}'$ for

$$\bar{x}^{\alpha} : \omega \vdash_{\Omega,\subset} P'_{\bar{x}^{\alpha},y^{\beta}} :: (y^{\beta} : C),$$

and then produce the last step of derivation

$$\dfrac{\overset{\mathbb{D}'}{\bar{x}^{\alpha} : \omega \vdash_{\Omega} P'_{\bar{x}^{\alpha},y^{\beta}} :: (y^{\beta} : C)} \quad \bar{x} : \omega \vdash Y = P'_{\bar{x},y} :: (y : C) \in V}{\bar{x}^{\alpha} : \omega \vdash_{\Omega} y^{\beta} \leftarrow Y \leftarrow \bar{x}^{\alpha} :: (y^{\beta} : C)}\textsc{Def}(Y)$$

in the infinitary rule system.

Moreover, by the coinductive hypothesis, we know that for every

$$\bar{w}^{\gamma'} : \omega' \vdash_{\Omega'} z^{\delta'} \leftarrow W \leftarrow w^{\gamma'} :: (z^{\delta'} : C')$$

on $\mathbb{D}'$, we have

$$W, \mathit{list}(w^{\gamma'}, z^{\delta'})\ (\subset, <_{\Omega'})\ Y, \mathit{list}(\bar{x}^{\alpha}, y^{\beta}).$$

By Lemma 6.20, we conclude from $Y, \mathit{list}(\bar{x}^{\alpha}, y^{\beta})\ (\subset, <_{\Omega})\ X, \mathit{list}(\bar{u}^{\gamma}, v^{\delta})$ that

$$Y, \mathit{list}(\bar{x}^{\alpha}, y^{\beta})\ (\subset, <_{\Omega'})\ X, \mathit{list}(\bar{u}^{\gamma}, v^{\delta}).$$

By transitivity of $(\subset, <_{\Omega'})$, we get

$$W, list(w^{\gamma'}, z^{\delta'})\ (\subset, <_{\Omega'})\ X, list(\bar{u}^{\gamma}, v^{\delta}).$$

This completes the proof of this case as we already know $Y, list(\bar{x}^{\alpha}, y^{\beta})\ (\subset, <_{\Omega})\ X, list(\bar{u}^{\gamma}, v^{\delta})$.

Case

$$\dfrac{\langle \bar{u}^{\gamma}, X, v^{\delta}\rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega \cup \mathtt{r}(y^{\beta}), \subset} Q_{z^0} :: (z^0 : C') \quad \langle \bar{u}^{\gamma}, X, v^{\delta}\rangle; z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^{\alpha}), \subset} Q'_{z^0} :: (y^{\beta} : C)}{\langle \bar{u}^{\gamma}, X, v^{\delta}\rangle; \bar{x}^{\alpha} : \omega \vdash_{\Omega, \subset} (z \leftarrow Q_z; Q'_z) :: (y^{\beta} : C)}\textsc{Cut}^z,$$

where $\mathtt{r}(w) = \{z^0_j = w_j \mid j \notin \mathtt{c}(A) \,\text{and}\, j \leq n\}$. By coinductive hypothesis, we have infinitary derivations $\mathbb{D}'$ and $\mathbb{D}''$ for $\bar{x}^{\alpha} : \omega \vdash_{\Omega \cup \mathtt{r}(y^{\beta})} Q_{z^0} :: (z^0 : C')$ and $z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^{\alpha})} Q'_{z^0} :: (y^{\beta} : C)$, respectively. We can produce the last step of the derivation as

$$\dfrac{\overset{\mathbb{D}'}{\bar{x}^{\alpha} : \omega \vdash_{\Omega \cup \mathtt{r}(y^{\beta})} Q_{z^0} :: (z^0 : C')} \quad \overset{\mathbb{D}''}{z^0 : C' \vdash_{\Omega \cup \mathtt{r}(x^{\alpha})} Q'_{z^0} :: (y^{\beta} : C)}}{\bar{x}^{\alpha} : \omega \vdash_{\Omega} (z \leftarrow Q_z; Q'_z) :: (y^{\beta} : C)}\textsc{Cut}^z$$

Moreover, by the coinductive hypothesis, we know that for every

$$\bar{w}^{\gamma} : \omega' \vdash_{\Omega'} z^{\delta} \leftarrow W \leftarrow \bar{w}^{\gamma} :: (z^{\delta} : C')$$

on $\mathbb{D}'$ and $\mathbb{D}''$, and thus $\mathbb{D}$, we have

$$W, list(\bar{w}^{\gamma}, z^{\delta})\ (\subset, <_{\Omega'})\ X, list(\bar{x}^{\alpha}, y^{\beta}).$$

Cases The proof of the other cases are similar by applying the coinductive hypothesis and the infinitary system rules.

$\square$

Theorem 6.22. *A locally guarded program satisfies the FS validity condition.*

Proof. Consider a cycle $\mathbb{C}$ on a (potentially infinite) derivation produced from $\langle \bar{u}, Y, v\rangle; \bar{z}^{\alpha} : \omega \vdash_{\Omega} w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : C)$ as in Lemma 6.21,

$$\dfrac{\dfrac{\dfrac{\dfrac{\bar{x}^{\gamma} : \omega \vdash_{\Omega'} P_{\bar{x}^{\gamma}, y^{\delta}} :: (y^{\delta} : C) \quad \bar{z} : \omega \vdash X = P_{\bar{z}, w} :: (w : C) \in V}{\bar{x}^{\gamma} : \omega \vdash_{\Omega'} y^{\delta} \leftarrow X \leftarrow \bar{x}^{\gamma} :: (y^{\delta} : C)}\textsc{Def}}{\vdots}}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} P_{\bar{z}^{\alpha}, w^{\beta}} :: (w^{\beta} : C)} \quad \bar{z} : \omega \vdash X = P_{\bar{z}, w} :: (w : C) \in V}{\bar{z}^{\alpha} : \omega \vdash_{\Omega} w^{\beta} \leftarrow X \leftarrow \bar{z}^{\alpha} :: (w^{\beta} : C)}\textsc{Def}$$

By Lemma 6.21 we get

$$X, list(\bar{x}^{\gamma}, y^{\delta})\ (\subset, <_{\Omega'})\ X, list(\bar{z}^{\alpha}, w^{\beta}),$$

and thus by definition of $(\subset, <_{\Omega'})$,

$$list(\bar{x}^{\gamma}, y^{\delta}) <_{\Omega'} list(\bar{z}^{\alpha}, w^{\beta}).$$

Therefore, there is an $i \leq n$, such that either

1. $\epsilon(i) = \mu$, $x_i^{\gamma} < z_i^{\alpha}$, and $x_l^{\gamma} = z_l^{\alpha}$ for every $l < i$, having that $\bar{x} = x$ and $\bar{z} = z$ are non-empty, or
2. $\epsilon(i) = \nu$, $y_i^{\delta} < w_i^{\beta}$, and $y_l^{\delta} = w_l^{\beta}$ for every $l < i$.

In the first case, by part (b) of Lemma 6.16, a μL rule with priority $i \in \mathsf{c}(\omega)$ is applied on $\mathbb{C}$. By part (a) of the same Lemma $x = z$, and by its part (c), no νL rule with priority $c < i$ is applied on $\mathbb{C}$. Therefore, $\mathbb{C}$ is a left μ- trace.
In the second case, by part (b) of Lemma 6.17, a νR rule with priority $i \in \mathsf{c}(\omega)$ is applied on $\mathbb{C}$. By part (a) of the same Lemma $y = w$ and by its part (c), no μR rule with priority $c < i$ is applied on $\mathbb{C}$. Thus, $\mathbb{C}$ is a right ν- trace. □

6.9 Computational meta-theory

In this section we use Fortier and Santocanale's result to prove a stronger compositional progress property for (locally) guarded programs.

Theorem 6.23. *(Strong Progress) Configuration $\bar{x} : \omega \Vdash C :: (y : A)$ of (locally) guarded processes satisfies the progress property. Furthermore, after finite number of steps, either*

1. *$\mathcal{C} = (\cdot)$ is empty,*
2. *or $\mathcal{C}$ attempts to communicate to the left or right.*

Proof. There is a correspondence between the TREAT function's internal operations and the synchronous computational transitions introduced in Section 5.5. The only point of difference is the extra computation rule we introduced for the constant 1. Fortier and Santocanale's proof of termination of the function TREAT remains intact after extending TREAT's primitive operation with a reduction rule for constant 1, since this reduction step only introduces a new way of closing a process in the configuration. Under this correspondence, termination of the function TREAT on valid proofs implies the strong progress property for guarded programs. □

As a corollary to Theorem 6.23, computation of a closed guarded program $\mathcal{P} = \langle V, S\rangle$ with $\cdot \vdash S = P :: (y : 1)$ always terminates by closing the channel y (which follows by inversion on the typing derivation).

We conclude this chapter by briefly revisiting sources of unguardedness in computation. In Example 6.1 we saw that process Loop is not guarded, even though its proof is cut-free. Its computation satisfies the strong progress property as it attempts to communicate with its right side in finite number of steps. However, its communication with left and right sides of the configuration is solely by sending messages. Composing Loop with any process $y : \mathsf{nat} \vdash$ P $:: (z : 1)$ results in exchanging an infinite number of messages between them. For instance, for Block, introduced in Example 6.1, the configuration $\cdot \Vdash y \leftarrow \texttt{Loop} \mid_y z \leftarrow \texttt{Block} \leftarrow y :: (z : 1)$ does not communicate to the left or right and a never ending series of internal communications takes place. This internal loop is a result of the infinite number of unfolding messages sent by Loop without any unfolding message with higher priority being received by it. In other words, it is the result of Loop not being guarded.

6.10 Incompleteness of guard conditions

In this section we provide a straightforward example of a program with the strong progress property that our algorithm cannot identify as guarded. Intuitively, this program seems to preserve the strong progress property after being composed by other guarded programs. We show that this example does not satisfy the FS validity condition, either.

Example 6.15. *Define the signature*

$$\begin{aligned}\Sigma_5 := \mathsf{ctr} &=^1_\nu \&\{inc : \mathsf{ctr},\ \ val : \mathsf{bin}\},\\ \mathsf{bin} &=^2_\mu \oplus\{b0 : \mathsf{bin}, b1 : \mathsf{bin}, \$: 1\}\end{aligned}$$

and program $\mathcal{P}_{11} = \langle\{\texttt{Bit0Ctr}, \texttt{Bit1Ctr}, \texttt{Empty}\}, \texttt{Empty}\rangle$, *where*

$$\begin{aligned}&x : \mathsf{ctr} \vdash y \leftarrow \texttt{Bit0Ctr} \leftarrow x :: (y : \mathsf{ctr})\\ &x : \mathsf{ctr} \vdash y \leftarrow \texttt{Bit1Ctr} \leftarrow x :: (y : \mathsf{ctr})\\ &\cdot \vdash y \leftarrow \texttt{Empty} :: (y : \mathsf{ctr})\end{aligned}$$

with

$$\begin{array}{lll}
y^\beta \leftarrow \texttt{Bit0Ctr} \leftarrow x^\alpha = & [0,0,\ 0,\ 0] & \\
\quad \textbf{case}\, Ry^\beta\ (\nu_{ctr} \Rightarrow & [-1,0,0,0] & y_1^{\beta+1} < y_1^\beta, y_2^{\beta+1} = y_2^\beta \\
\quad\quad \textbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ y^{\beta+1} \leftarrow \texttt{Bit1Ctr} \leftarrow x^\alpha & [-1,\ 0,\ 0,0] & \\
\quad\quad\quad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.b0; Lx^\alpha.\nu_{ctr}; Lx^{\alpha+1}.val; y^{\beta+2} \leftarrow x^{\alpha+1})) & [-1,1,0,1] &
\end{array}$$

$$\begin{array}{lll}
y^\beta \leftarrow \texttt{Bit1Ctr} \leftarrow x^\alpha = & [0,0,\ 0,\ 0] & \\
\quad \mathbf{case}\, Ry^\beta\ (\nu_{ctr} \Rightarrow & [-1,0,0,0] & y_1^{\beta+1} < y_1^\beta, y_2^{\beta+1} = y_2^\beta \\
\quad\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ Lx^\alpha.\nu_{ctr}; Lx^{\alpha+1}.inc; y^{\beta+1} \leftarrow \texttt{Bit0Ctr} \leftarrow x^{\alpha+1} & [-1,\ 1,\ 0,0] & x_2^{\alpha+1} = x_2^\alpha \\
\quad\quad\quad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.b1; Lx^\alpha.\nu_{ctr}; Lx^{\alpha+1}.val; y^{\beta+2} \leftarrow x^{\alpha+1})) & [-1,1,0,1] &
\end{array}$$

$$\begin{array}{lll}
y^\beta \leftarrow \texttt{Empty} \leftarrow \cdot = & [0,\_,\ \_,\ 0] & \\
\quad \mathbf{case}\, Ry^\beta\ (\nu_{ctr} \Rightarrow & [-1,\_,\_,0] & y_1^{\beta+1} < y_1^\beta, y_2^{\beta+1} = y_2^\beta \\
\quad\quad \mathbf{case}\, Ry^{\beta+1}\ (inc \Rightarrow\ w^0 \leftarrow \texttt{Empty} \leftarrow \cdot; & [\infty,\ \_,\ \_,\infty] & \mathsf{ctr}, \mathtt{bin} \in \mathtt{c}(\mathsf{ctr}) \\
\qquad\qquad\qquad\qquad y^{\beta+1} \leftarrow \texttt{Bit1Ctr} \leftarrow w^0 & [-1,\ \infty,\infty,0] & \mathsf{ctr}, \mathtt{bin} \in \mathtt{c}(ctr) \\
\qquad\qquad \mid val \Rightarrow Ry^{\beta+1}.\mu_{bin}; Ry^{\beta+2}.\$; \mathbf{close}\, Ry^{\beta+2})) & [-1,\_,\_,1] &
\end{array}$$

In this example we implement a counter slightly differently from Example 6.11. We have two processes Bit0Ctr *and* Bit1Ctr *that are holding one bit ($b0$ and $b1$ respectively) and a counter* Empty *that signals the end of the chain of counter processes. This program begins with an empty counter (representing value 0). If a value is requested, then it sends \$ to the right and if an increment is requested it adds the counter* Bit1Ctr *with $b1$ value to the chain of counters. Then if another increment is asked,* Bit1Ctr *sends an increment (inc) message to its left counter (implementing the carry bit) and calls* Bit0Ctr*. If* Bit0Ctr *receives an increment from the right, it calls* Bit1Ctr *recursively.*

All (mutually) recursive calls in this program are recognized as guarded by our algorithm, except the one in which Empty *calls itself. In this recursive call, $y^\beta \leftarrow$* Empty *$\leftarrow \cdot$ calls $w^0 \leftarrow$* Empty *$\leftarrow \cdot$, where w is the fresh channel it shares with $y^{\beta+1} \leftarrow$* Bit1Ctr *$\leftarrow w^0$. The number of increment unfolding messages* Bit1Ctr *can send along channel w^0 are always less than or equal to the number of increment unfolding messages it receives along channel $y^{\beta+1}$. This implies that the number of messages $w^0 \leftarrow$* Empty *$\leftarrow \cdot$ may receive along channel w^0 is strictly less than the number of messages received by any process along channel y^β. There will be no infinite loop in the program without receiving an unfolding message from the right. Indeed Fortier and Santocanale's cut elimination for the cut corresponding to the composition* Empty $\mid$ Bit1Ctr *locally terminates. Furthermore, since no guarded program defined on the same signature can send infinitely many increment messages to the left, $\mathcal{P}_{11}$ composed with any other guarded program satisfies strong progress.*

This result is also a negative example for the FS validity condition. The path between $y^\beta \leftarrow$ Empty *$\leftarrow \cdot$ and $w^0 \leftarrow$* Empty *$\leftarrow \cdot$ in the* Empty *process is neither left traceable not right traceable since $w \neq y$. By Definition 6.14 it is therefore not a valid cycle.*

Example 6.15 shows that neither our condition nor the FS validity condition are complete. In fact, using Theorem 6.23 we can prove that no effective procedure, including our algorithm, can recognize a maximal set Ξ of programs with the strong progress property that is closed under composition.

Theorem 6.24. *It is undecidable to recognize a maximal set Ξ of session-typed programs in subsingleton logic with the strong progress property that is closed under composition.*

Proof. Pfenning and DeYoung showed that any Turing machine can be represented as a process in subsingleton logic with equirecursive fixed points [31, 73], easily embedded into our setting with isorecursive fixed points. It implies that a Turing machine on a given input halts if and only if the closed process representing it terminates. By definition of strong progress, a closed process terminates if and only if it satisfies strong progress property. Using this result, we reduce the halting problem to identifying closed programs $\mathcal{P} := \langle V, S \rangle$ with $\cdot \vdash S :: x : 1$ that satisfy strong progress. Note that a closed program satisfying strong progress is in every maximal set Ξ. □

Chapter 7

Strong progress as a predicate

7.1 Background on processes as formula

The process-as-formula paradigm has been introduced by Miller [68] in 1993. He embedded processes in the π-calculus as formulas in a linear logic with non-logical constants. He further identified computation of processes as a search for a cut-free sequent proof. He proposed conjunctive and disjunctive translations as two alternative but dual approaches for embedding processes as linear logic formulas. In both approaches, the translation is defined by induction over the structure of processes. The conjunctive translation uses multiplicative and additive conjunctions ($\otimes, \&$) and the existential quantifier ($\exists$), while the disjunctive one uses multiplicative and additive disjunctions ($\parr, \oplus$) and the universal quantifier ($\forall$). The conjunctive and disjunctive translations identify reduction steps in the π-calculus with entails and entailed-by, respectively. For example, reducing process P_1 to P_2 in multiple steps ($P_1 \mapsto^* P_2$) is identified as the entailment $[P_1] \vdash [P_2]$ in linear logic, where formula $[P_i]$ is the conjunctive translation of process P_i.

Revisions of this embedding to more expressive (finitary) extensions of linear logic [14, 55, 96] are used to prove properties about processes, e.g. proofs of progress (deadlock-freedom) for circular multiparty sessions [54] and bisimilarity for π-calculus processes [96]. Horne and Tiu [55] studied the embedding of π-calculus processes into a logic called BV, a multiplicative linear logic extended with a non-commutative self-dual logical operator. They showed that linear implication is strictly finer than any interleaving preorder in their settings. In particular, they showed that linear implication is sound with respect to weak simulation and trace inclusion. Cervesato and Scedrov [17] also described encoding of both synchronous and synchronous semantics of concurrent π-calculus processes in first-order linear logic. They prove the soundness and completeness of their encoding with regard to the notions of structural equivalence and computation.

In this chapter, we follow the approach of Miller to embed session-typed processes with an asynchronous semantics as formulas in the infinitary first-order multiplicative linear logic with fixed points ($FIMALL^{\infty}_{\mu,\nu}$) that we introduced in Chapter 4. Our embedding is closely related to Miller's conjunctive translation, but we present it as a predicate defined over configurations using mutual induction and coinduction. Our principal motivation for introducing this embedding is to prove strong progress for binary session types. To achieve our goal, we formalize the strong progress property as a predicate indexed by session types. We show that the embedding of a configuration entails the strong progress predicate: we build a derivation for the entailment in $FIMALL^{\infty}_{\mu,\nu}$ and prove that the derivation is a valid proof if the underlying configuration is guarded. Finally, we build a cut-free valid proof for this entailment and show that it ensures strong progress of the underlying configuration when the configuration is executed with any synchronous scheduler.

A synchronous scheduler synchronizes sends and receives along internal channels of the configuration; a message can be spawned only if there is a process in the configuration ready to receive it. However, a process may spawn a message along an external channel of the configuration without waiting for a receiver. As a result, even after restricting the scheduler to be synchronous, we can prove the strong progress property stated for asynchronous semantics: a configuration either terminates in an empty configuration or one attempting to *receive along an external channel* (see Section 5.7). We will leave it to future work to build a derivation that ensures strong progress of the underlying configuration when executed with an arbitrary scheduler.

In essence, in this chapter, we use $FIMALL^{\infty}_{\mu,\nu}$ as an infinitary metalogic to carry out proof of strong progress in it. The formalization of strong progress as a predicate defined with nested least and greatest fixed points and the structure of the strong progress proof in a substructural metalogic with circular proofs provides a better understanding of the nature of the strong progress property.

In Section 7.2, we present infinitary inference rules for session-typed processes similar to the one introduced in Section 6.6. Similar to its counterpart, programs derived in this system are all *well-typed*, but not necessarily enjoy strong progress. We impose a *guard* condition on processes and prove strong progress using a processes-as-formulas approach for guarded processes. The guard condition introduced in this chapter is not a local one. However, it is straightforward to adapt the proof in Chapter 6 to show that the local guard condition we described in Section 6.7 is a stricter condition than the one we introduce here; the results of this chapter also hold for locally guarded processes.

Several notations introduced for $FIMALL^{\infty}_{\mu,\nu}$ calculus overlap with the notations we use in the context of session-typed processes. For example in both, a signature stores definitions and the relative priority of fixed points. The overloaded notation is inevitable since our metalogic is a generalization of the infinitary subsingleton logic with fixed points based on which binary session typed processes are defined. We use it to our advantage in the last section to prove our main result using a bisimulation. Whenever possible, we distinguish between the notations in $FIMALL^{\infty}_{\mu,\nu}$ and session-type processes by using different fonts. For example, we refer to the first-order signature that contains predicates as Σ, and to the signature in session-typed processes that contains (not dependent) session types as $\mathbf{\Sigma}$. In particular, for the rest of this chapter, we fix a signature $\mathbf{\Sigma}$ with $\mathbf{n}$ being the maximum priority in it. We use the type-setting $\mathtt{A}$ for session types, and P for process terms to emphasize that they are different from formulas in $FIMALL^{\infty}_{\mu,\nu}$.

7.2 Typing rules for session-typed processes

This section presents an infinitary type system for session-typed processes with the least and greatest fixed points and a guard condition on the typing derivations. The calculus and the guard condition imposed upon it are refinements of their counterparts in Chapter 6. Here a channel evolves to its next generation after any sort of communication, not only by sending or receiving a fixed point unfolding message. This change is needed to establish a bisimulation critical to the proof of our main theorem (see Section 7.5). Moreover, we relax the condition on the cut rule since we are not looking for a local guard condition here. We do not annotate process terms with generations in the calculus presented in this chapter for brevity.

The process typing judgments are of the form

$$\bar{x}^{\alpha} : \omega \vdash_{\Omega} \mathsf{P} :: (y^{\beta} : \mathtt{A}),$$

where P is a process, and x^{α} (the α-th generation of channel x) and y^{β} (the β-th generation of channel y) are its left and right channels of types ω and A, respectively. We build Ω to collect the relation between the generations of left and right channels indexed by their priority of types. We only consider judgments in which all variables $x^{\alpha'}$ ($y^{\beta'}$) occurring in Ω are such that $\alpha' < \alpha$ ($\beta' < \beta$), and impose a freshness condition on the channel introduced by the cut rule. This presupposition guarantees that if we construct a derivation bottom-up, any future generations for x and y are fresh and not yet constrained by Ω. All our rules, again read bottom-up, will preserve this property.

Programs derived in this system are all *well-typed*, but not necessarily *guarded*. We use a list notation to define a guard condition on processes. For a given signature Σ, *snapshot* of a channel x^{α} is a list $\mathsf{snap}(x^{\alpha}) = [\mathbf{x}_i^{\alpha}]_{i \leq n}$, where n is the maximum priority in Σ. Having the relation between annotated channel in Ω, we can define a partial order on snapshots of channels [1]. The left μ-trace and right ν-trace are defined similar to previous definitions.

[1]This notation is the same as the list $[x^{\alpha}]i \leq n$ introduced in Section 6.8 and similar to snap used over generational variables in Section 4.3. We avoid using the $[x^{\alpha}]i \leq n$ notation here to avoid confusion with the notation that we will use in Figure 7.3

Definition 7.1. An infinite branch of a derivation is a *left μ-trace* if for infinitely many channels $x1^{\alpha_1}, x2^{\alpha_2}, \cdots$ appearing as antecedents of judgments in the branch as

$$\begin{array}{c} \vdots \\ x3^{\alpha_3} : \mathtt{A}_3 \vdash_{\Omega_2} \mathtt{Q}_3 :: (z^{\eta} : \mathtt{C}_3) \\ \hline \vdots \\ x2^{\alpha_2} : \mathtt{A}_2 \vdash_{\Omega_1} \mathtt{Q}_2 :: (y^{\delta} : \mathtt{C}_2) \\ \hline \vdots \\ \hline x1^{\alpha_1} : \mathtt{A}_1 \vdash_{\Omega_0} \mathtt{Q}_1 :: (w^{\beta} : \mathtt{C}_1) \\ \vdots \end{array}$$

we can form an infinite chain of inequalities

$$\mathsf{snap}(x1^{\alpha_1}) >_{\Omega_1} \mathsf{snap}(x2^{\alpha_2}) >_{\Omega_2} \cdots .$$

Dually, an infinite branch of a derivation is a *right ν-trace* if for infinitely many channels $y1^{\beta_1}, y2^{\beta_2}, \cdots$ appearing as the succedents of judgments in the branch as

$$\begin{array}{c} \vdots \\ \bar{w}^{\delta} : \omega_2 \vdash_{\Omega_1} \mathtt{Q}_2 :: (y2^{\beta_2} : \mathtt{C}_2) \\ \hline \vdots \\ \hline \bar{x}^{\alpha} : \omega_1 \vdash_{\Omega} \mathtt{Q}_1 :: (y1^{\beta_1} : \mathtt{C}_1) \\ \vdots \end{array}$$

we can form an infinite chain of inequalities

$$\mathsf{snap}(y1^{\beta_1}) >_{\Omega_1} \mathsf{snap}(y2^{\beta_2}) >_{\Omega_2} \cdots .$$

Definition 7.2 (Guard condition for processes)**.** A program defined over signature $\mathbf{\Sigma}$ and the set of process definitions V is *guarded* if any infinite branch in the derivation of $\bar{x}^{\alpha} : \omega \vdash y \leftarrow \mathsf{X} \leftarrow x :: y^{\beta} : \mathtt{B}$ for every $\bar{x} : \omega \vdash \mathsf{X} = \mathsf{P}_{\bar{x},y} :: y : \mathtt{B} \in V$ is either a left μ-trace or a right ν-trace.

It is straightforward to adapt the proof in the previous chapter to show that if a program is locally guarded, it is also guarded by Definition 7.2.

$$\frac{}{x^\alpha : \mathtt{A} \vdash_\Omega y \leftarrow x :: (y^\beta : \mathtt{A})}\,\textsc{Id}$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P}_w :: (w^\eta : \mathtt{A}) \quad w^\eta : \mathtt{A} \vdash_\Omega \mathsf{Q}_w :: (y^\beta : \mathtt{C})}{\bar{x}^\alpha : \omega \vdash_\Omega ((w : \mathtt{A}) \leftarrow \mathsf{P}_w; \mathsf{Q}_w) :: (y^\beta : \mathtt{C})}\,\textsc{Cut}^w$$

$$\frac{}{\cdot \vdash_\Omega \mathbf{close}\, Ry :: (y^\beta : 1)}\,1R$$

$$\frac{\cdot \vdash_\Omega \mathsf{Q} :: (y^\beta : \mathtt{A})}{x^\alpha : 1 \vdash_\Omega \mathbf{wait}\, Lx; \mathsf{Q} :: (y^\beta : \mathtt{A})}\,1L$$

$$\frac{\bar{x}^\alpha : \omega \vdash_{\Omega\cup\{y^\beta = y^{\beta+1}\}} \mathsf{P} :: (y^{\beta+1} : \mathtt{A}_k) \quad (k \in L)}{\bar{x}^\alpha : \omega \vdash_\Omega Ry.k; \mathsf{P} :: (y^\beta : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L})}\,\oplus R$$

$$\frac{\forall \ell \in L \quad x^{\alpha+1} : \mathtt{A}_\ell \vdash_{\Omega\cup\{x^\alpha = x^{\alpha+1}\}} \mathsf{P}_\ell :: (y^\beta : \mathtt{C})}{x^\alpha : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L} \vdash_\Omega \mathbf{case}\, Lx\ (\ell \Rightarrow \mathsf{P}_\ell) :: (y^\beta : \mathtt{C})}\,\oplus L$$

$$\frac{\forall \ell \in L \quad \bar{x}^\alpha : \omega \vdash_{\Omega\cup\{y^\beta = y^{\beta+1}\}} \mathsf{P}_\ell :: (y^{\beta+1} : \mathtt{A}_\ell)}{\bar{x}^\alpha : \omega \vdash_\Omega \mathbf{case}\, Ry\ (\ell \Rightarrow \mathsf{P}_\ell) :: (y^\beta : \&\{\ell : \mathtt{A}_\ell\}_{\ell\in L})}\,\&R$$

$$\frac{k \in L \quad x^{\alpha+1} : \mathtt{A}_k \vdash_{\Omega\cup\{x^\alpha = x^{\alpha+1}\}} \mathsf{P} :: (y^\beta : \mathtt{C})}{x^\alpha : \&\{\ell : \mathtt{A}_l\}_{\ell\in L} \vdash_\Omega Lx.k; \mathsf{P} :: (y^\beta : \mathtt{C})}\,\&L$$

$$\frac{\begin{array}{l}\Omega' = \Omega \cup \{(y^\beta)_j = (y^{\beta+1})_j \mid j \neq i\} \\ \quad \bar{x}^\alpha : \omega \vdash_{\Omega'} \mathsf{P}_y :: (y^{\beta+1} : \mathtt{A}) \qquad t =^i_\mu \mathtt{A}\end{array}}{\bar{x}^\alpha : \omega \vdash_\Omega Ry.\mu_t; \mathsf{P}_{y^\beta} :: (y^\beta : t)}\,\mu R$$

$$\frac{\begin{array}{l}\Omega' = \Omega \cup \{x_i^{\alpha+1} < x_i^\alpha\} \cup \{x_j^{\alpha+1} = x_j^\alpha \mid j \neq i\} \\ \quad x^{\alpha+1} : \mathtt{A} \vdash_{\Omega'} \mathsf{Q}_{x^{\alpha+1}} :: (y^\beta : \mathtt{C}) \qquad t =^i_\mu \mathtt{A}\end{array}}{x^\alpha : t \vdash_\Omega \mathbf{case}\, Lx\ (\mu_t \Rightarrow \mathsf{Q}_{x^\alpha}) :: (y^\beta : \mathtt{C})}\,\mu L$$

$$\frac{\begin{array}{l}\Omega' = \Omega \cup \{y_i^{\beta+1} < y_i^\beta\} \cup \{y_j^{\beta+1} = y_j^\beta \mid i \neq j\} \\ \quad \bar{x}^\alpha : \omega \vdash_{\Omega'} \mathsf{P} :: (y^{\beta+1} : \mathtt{A}) \qquad t =^i_\nu \mathtt{A}\end{array}}{\bar{x}^\alpha : \omega \vdash_\Omega \mathbf{case}\, Ry\ (\nu_t \Rightarrow \mathsf{P}) :: (y^\beta : t)}\,\nu R$$

$$\frac{\begin{array}{l}\Omega' = \Omega \cup \{(x^{\alpha+1})_j = (x^\alpha)_j \mid j \neq i\} \\ \quad x^{\alpha+1} : \mathtt{A} \vdash_{\Omega'} \mathsf{Q} :: (y^\beta : \mathtt{C}) \qquad t =^i_\nu \mathtt{A}\end{array}}{x^\alpha : t \vdash_\Omega Lx.\nu_t; \mathsf{Q} :: (y^\beta : \mathtt{C})}\,\nu L$$

$$\frac{\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P}_{\bar{x},y} :: (y^\beta : \mathtt{C}) \quad \bar{u} : \omega \vdash \mathsf{X} = \mathsf{P}_{\bar{u},w} :: (w : \mathtt{C}) \in V}{\bar{x}^\alpha : \omega \vdash_\Omega y \leftarrow \mathsf{X} \leftarrow \bar{x} :: (y^\beta : \mathtt{C})}\,\textsc{Def}(X)$$

FIGURE 7.1: Infinitary typing rules for processes with an ordering on channels.

7.3 Asynchronous Semantics

In this section, we follow the approach of processes-as-formulas to provide an asynchronous semantics for session-typed processes.

Recall the definition of configuration $\mathcal{C}$ as a list of processes that communicate with each other along their private channels:

$$\mathcal{C} ::= \cdot \mid \mathbf{msg}(M) \mid \mathbf{proc}(\mathsf{P}) \mid (\mathcal{C}_1 \mid_{x:\mathtt{A}} \mathcal{C}_2),$$

where $\mid$ is an associative, noncommutative operator and $(\cdot)$ is the unit. The type checking judgment and rules for configurations can be adapted to include generation of channels. The type checking rules for $\bar{x}^\alpha : \omega \Vdash \mathcal{C} :: (y^\beta : \mathtt{B})$ are:

$$\frac{}{x^\alpha : \mathtt{A} \Vdash \cdot :: (x^\alpha : \mathtt{A})}\text{EMP}$$

$$\frac{\bar{x}^\alpha : \omega \Vdash \mathcal{C}_1 :: (z^0 : \mathtt{A}) \quad z^0 : \mathtt{A} \Vdash \mathcal{C}_2 :: (y^\beta : \mathtt{B})}{\bar{x}^\alpha : \omega \Vdash \mathcal{C}_1|_{z:\mathtt{A}}\, \mathcal{C}_2 :: (y^\beta : \mathtt{B})}\text{COMP}$$

$$\frac{\bar{x}^\alpha : \omega \vdash \mathsf{P} :: (y^\beta : \mathtt{B})}{\bar{x}^\alpha : \omega \Vdash \mathsf{P} :: (y^\beta : \mathtt{B})}\text{PROC}$$

$$\frac{\bar{x}^\alpha : \omega \vdash M :: (y^\beta : B)}{\bar{x}^\alpha : \omega \Vdash \mathbf{msg}(M) :: (y^\beta : B)}\text{MSG}$$

By assumption, the original configuration that the computation starts from does not contain any messages; it only consists of processes, and messages appear in the configuration throughout the computation.

We read predicate $\mathsf{Msg}(x^\alpha.b(y^\beta))$ as message b is sent along x^α and x^α is substituted by its continuation y^β. It can be interpreted as a translation for $\mathsf{msg}(Rx^\alpha.b; x^\alpha \leftarrow y^\beta)$ when x^α is of a positive type and $\mathsf{msg}(Lx^\alpha.b; y^\beta \leftarrow x^\alpha)$ when x^α is of a negative type.

For a configuration of processes $(\bar{x}^\alpha : \omega) \Vdash \mathcal{C} :: (y^\beta : \mathtt{B})$, we define the recursive predicate $\mathsf{Cfg}_{x^\alpha:\omega, y^\beta:\mathtt{B}}(\mathcal{C})$ as its translation (Figure 7.2). Similar to Miller's conjunctive translation we define spawning by multiplicative conjunction ($\otimes$), choosing between branches by additive conjunction ($\&$), and bounding channels by existential quantifier ($\exists$). For example, the translation for process $\bar{x}^\alpha : \omega \vdash (z : \mathtt{C}) \leftarrow \mathsf{Q}_1; \mathsf{Q}_2 :: (y^\eta{:}\mathtt{B})$ that spawns $\bar{x}^\alpha : \mathtt{A} \vdash \mathsf{Q}_1 :: z^\eta{:}\mathtt{C}$ and continues as $z^\eta{:}\mathtt{C} \vdash \mathsf{Q}_2 :: y^\beta{:}\mathtt{B}$ is as follows:

$$\exists z.\exists \eta.\mathsf{Cfg}_{\bar{x}^\alpha:\mathtt{A}, z^\eta:\mathtt{C}}(\mathsf{Q}_1) \otimes \mathsf{Cfg}_{z^\eta:\mathtt{C}, y^\beta:\mathtt{B}}(\mathsf{Q}_2)$$

The translation of a forwarding process $x^\alpha{:}\mathtt{A} \vdash (y \leftarrow x) :: y^\beta{:}\mathtt{A}$, is simply to make the channels equal to each other $x^\alpha = y^\beta$. Here, α and β range over natural numbers, and channels are names. To find the most general unifier(mgu) for x^α and y^β in the logic, we treat a channel and its generation as an abstract variable indexed by a natural number. In this case, the mgu of x^α and y^β can be either of them.

Miller's work is in the synchronous semantics of π-calculus. In a synchronous setting, messages are not spawned as a special form of processes. Instead, senders and receivers wait until both are ready to perform the message transfer. To model this behavior, Miller added two non-logical constants `send` and `get` to the language of linear logic for receiving and sending a message, respectively. He described the computation when a pair of matching `send` and `get` appear in the configuration using a first-order Horn clause:

$$\forall x, y, P, Q.\, ((\mathtt{get}\, x\, w;\ P_w \otimes \mathtt{send}\, x\, y; Q) \multimap (P_y \otimes Q))$$

$\mathtt{send}\, x\, y; Q$ is the translation for a process that is ready to send message y along channel x and continues as Q. And $\mathtt{get}\, x\, w; P_w$ is the translation for a process that is ready to receive any messages w offered along x and continue according to the content of w as P_w. This translation is compatible with the synchronous nature of Miller's setting.

In an asynchronous semantics, the senders output a message and proceed with their continuation. In Section 5.5.3 we modeled outputted messages as specific processes containing the value of the message followed by a forwarding and extended the configuration grammar to include them as well. In this chapter, we translate messages as a specified predicate $\mathsf{Msg}(x^\alpha.b(y^\beta))$ in our logic.

Similar to other cases of spawning a process, we use multiplicative conjunction ($\otimes$) for spawning a message. Recall from Section 5.5.3 that in an asynchronous semantics a fresh channel is allocated whenever a new message is spawned. The forward then links the fresh channel and the previous one. In this chapter we set the new allocated channel to be the next generation of the previous one. For example, the process $x^\alpha : \&\{\ell{:}\mathtt{A}_\ell\}_{\ell \in L} \vdash Lx^\alpha.k; \mathsf{P} :: y^\beta{:}\mathtt{B}$ spawns a message $\mathbf{msg}(Lx^\alpha.k; x^{\alpha+1} \leftarrow x^\alpha)$ and continues as $x^{\alpha+1} : \mathtt{A}_k \vdash \mathsf{P} :: y^\beta{:}\mathtt{B}$. We embed process $x^\alpha.k; \mathsf{P}$ as

$$\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k, y^\beta:\mathtt{B}}(\mathsf{P}).$$

To describe receiving a message, we use the dual operator $\multimap$. Even if we always introduce the continuation of a spawned message to be the next generation of the previous one, we still need to consider a general form of the message predicate $\mathsf{Msg}(x^\alpha.b(w^\eta))$ since forwarding may substitute the new generation of a channel ($x^{\alpha+1}$) with another channel (w^η). As a result, the continuation channel of the receiving process depends on the content of message it receives. For example, process $z^\delta : \omega \vdash \mathbf{case}\, Rx^\alpha(\ell \Rightarrow \mathsf{Q}_\ell) :: x^\alpha{:}\&\{\ell : \mathtt{A}_\ell\}_{\ell \in L}$ waits to receive a message $Lx^\alpha.\ell; w^\eta \leftarrow x^\alpha$ for any channel w^η and label $\ell \in L$ to continue as $z^\delta : \omega \vdash \mathsf{Q}_\ell ::$

$x^\alpha{:}\mathtt{A}_\ell$. Following Miller's conjunctive approach, we translate this process as

$$\forall w^\eta.\ \&\{\ \mathsf{Msg}(x^\alpha.\ell(w^\eta)) \multimap \mathsf{Cfg}_{z^\delta:\omega, w^\eta:\mathtt{A}_\ell}(\mathsf{Q}_\ell)\}_{\ell\in L}.$$

The following derivation provides an example to show the relation between asynchronous transitions of configurations and entailment of their translations[2]. As apparent in this example derivation, the use of multiplicative conjunction ($\otimes$) and linear implication ($\multimap$) for modeling sending and receiving messages is not surprising; they model producing and consuming a resource in linear logic, respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \vdash \mathsf{Msg}(x^\alpha.k(x^{\alpha+1}))}\textsc{Id} \quad \dfrac{}{\mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_k}(\mathsf{Q}_k) \vdash \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_\ell}(\mathsf{Q}_k)}\textsc{Id}}{\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})), \mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \multimap \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_k}(\mathsf{Q}_k) \vdash \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_\ell}(\mathsf{Q}_k)}{\multimap}L}{\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})), \&\{\ \mathsf{Msg}(x^\alpha.\ell(x^{\alpha+1})) \multimap \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_\ell}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_\ell}(\mathsf{Q}_k)}\&L}{\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})), \forall w^\eta.\ \&\{\ \mathsf{Msg}(x^\alpha.\ell(w^\eta)) \multimap \mathsf{Cfg}_{z^\delta:\omega, w^\eta:\mathtt{A}_\ell}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash \mathsf{Cfg}_{z^\delta:\omega, x^{\alpha+1}:\mathtt{A}_k}(\mathsf{Q}_k)}\forall L$$

Using the specified predicate for the messages, we can avoid introducing Miller's non-logical constants `send` and `get` to the language. There will be no need for a Horn clause to capture the connection between sending a label and receiving it either; the duality between multiplicative conjunction and linear implication handles this connection. However, our formulation results in asynchronous semantics: spawning a message is not necessarily synchronized with receiving it.

As a special case, when a configuration is of the form $\cdot \vdash \mathbf{close}Ry :: y^\alpha{:}1$, we spawn a message $\mathsf{Msg}(y^\alpha.\mathbf{closed}(\_))$ with no continuation, and terminate the translation:

$$\mathsf{Msg}(y^\alpha.\mathbf{closed}(\_)) \otimes 1$$

Similarly, we embed the process of the form $y^\alpha{:}1 \vdash \mathbf{wait}Ly; \mathsf{Q} :: (z^\delta : \mathtt{B})$ that waits on a closing message as:

$$\mathsf{Msg}(y^\alpha.\mathbf{closed}(\_)) \multimap \mathsf{Cfg}_{\cdot, z^\delta:\mathtt{B}}(\mathsf{Q})$$

One essential difference between our logic and Miller's is that our underlying logic has the apparatus to internalize inductive and coinductive predicates in the language. The predicate Cfg is defined inside the language using mutual induction and coinduction. In Figure 7.2 we present a definition of Cfg based on pattern matching. The first two cases in the definition of Cfg reflect the rules for composition of configurations and an empty configuration. In the second line where two configurations are composed, a fresh channel z^η is created. Channel z^η

[2]It is out of this thesis's scope to prove formally that configuration transitions are identified with entailment. In particular, because the transition of a program by definition has a finite nature, one may need to restrict entailment only to sequents with finite proofs to get such a result.

is an internal channel in the composition of configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ and is used by them to communicate with each other. Channels $\bar{x}^\alpha$ and y^β are the externals channels of this composition.

The rest of the cases in Figure 7.2 refer to the configuration consists of a single process $(\bar{x}^\alpha : \omega) \vdash \mathsf{P} :: (y^\beta : \mathtt{B})$. For identity (row 3) we put the generational channels to be equal to each other. Cut (row 4) spawns a new process Q_1 offering along a fresh variable z^η and continues as Q_2 which is using the resource offered along z^η. Processes Q_1 and Q_2 communicate along their private channel z^η.

The definition of predicate Cfg in rows 5-14 of Figure 7.2 captures the operational meaning of session types presented in Tables 5.1 and 5.2. For the cases in which the process sends a message along a channel (rows 5,8,9,12,13), we first declare the message and then proceed the computation with the rest of the process. In the cases where the process needs to receive a message to continue (rows 6,7,10,11,14), the predicate is defined as a conjunction of the possible continuations, universally quantified for the possible continuations. The definition may proceed with each continuation channel and label provided that the label is declared via a message predicate Msg.

In the last case a process variable is unfolded while instantiating the left and right channels $\bar{u}$ and w in the process definition with proper channel names $\bar{x}$ and y, respectively. All cases of Cfg except the last one are defined recursively on a process with a smaller size; we define the predicate in these cases as a least fixed point. In the last case a process variable Y is replaced by its definition Q of a possibly larger size; accordingly the predicate is defined as a greatest fixed point in this case. Since the behaviour of a recursive process cannot be defined by induction on its size, we put the priority of the ν-term in the last case to be higher than μ-terms in the other cases to express that the behaviour is defined coinductively. The first $\mathbf{n}$ priorities in the signature are reserved for the recursive cases in the definition of the strong progress predicate which we will introduce in Section 7.4.

The process terms and session types are not part of the Cfg predicate; they are parameters. We build the equivalent formula for Cfg using existential quantifiers for each pattern and $\oplus$ to unify the cases into a single formula. Here, we show a part of the equivalent formula corresponding to the first 7 lines of Figure 7.2, where c_i and d_i stand for variables in $FIMALL^{\infty}_{\mu,\nu}$:

$$
\begin{aligned}
\mathsf{Cfg}_{\bar{x}^\alpha:d_1,y^\beta:d_2}(d_3) = \quad & (\exists c_1, c_2, z, c.\,(d_3 = c_1|_{z:c}c_2) \otimes \exists z.\exists \eta.\mathsf{Cfg}_{\bar{x}^\alpha:d_1,z^\eta:c}(c_1) \otimes \mathsf{Cfg}_{z^\eta:c,y^\beta:d_2}(c_2)) \\
& \oplus ((d_3 = \cdot) \otimes 1) \\
& \oplus (d_3 = y \leftarrow x) \otimes x^\alpha = y^\beta \\
& \oplus (\exists c_1, c_2, z, c.\,(d_3 = (z : c) \leftarrow c_1; c_2) \otimes \exists z.\exists \eta.\mathsf{Cfg}_{\bar{x}^\alpha:d_1,z^\eta:c}(c_1) \otimes \mathsf{Cfg}_{z^\eta:c,y^\beta:d_2}(c_2)) \\
& \oplus d_2 = 1 \otimes (d_3 = \mathbf{close} Ry) \otimes \mathsf{Msg}(y^\beta.\mathsf{closed}(\_)) \otimes 1 \\
& \oplus \exists c_1.d_1 = 1 \otimes d_3 = \mathbf{wait} Lx; c_1 \otimes \mathsf{Msg}(y^\beta.\mathsf{closed}(\_)) \multimap \mathsf{Cfg}_{\cdot,y^\beta:d_2}(c_1) \\
& \oplus \exists c_1, L, \vec{A}_{\ell\in L}.d_2 = \&\{\ell{:}A_\ell\}_{\ell\in L} \otimes \\
& \qquad d_3 = \mathbf{case} Ry(\ell \Rightarrow c_\ell)_{\ell\in L} \otimes \forall w^\eta.\&\{\ell : \mathsf{Msg}(y^\beta.\ell(w^\eta)) \multimap \mathsf{Cfg}_{\bar{x}^\alpha:d_1,w^\eta:A_\ell}(c_\ell)\} \\
& \oplus \cdots
\end{aligned}
$$

$$\begin{array}{lllr}
1. & \mathsf{Cfg}_{x^\alpha:\mathtt{A},x^\alpha:\mathtt{A}}(\cdot) & =^{\mathbf{n}+2}_{\mu} & 1 \\
 & & & \textit{empty configuration} \\
2. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathcal{C}_1|_{z:\mathtt{C}}\mathcal{C}_2) & =^{\mathbf{n}+2}_{\mu} & \exists z.\exists\eta.\mathsf{Cfg}_{\bar{x}^\alpha:\omega,z^\eta:\mathtt{C}}(\mathcal{C}_1)\otimes\mathsf{Cfg}_{z^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathcal{C}_2) \\
 & & & \textit{composition of configurations} \\
3. & \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{A}}(y\leftarrow x) & =^{\mathbf{n}+2}_{\mu} & (x^\alpha = y^\beta) \\
 & & & \textit{forward} \\
4. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}((z:\mathtt{C})\leftarrow \mathsf{Q}_1\ ;\ \mathsf{Q}_2) & =^{\mathbf{n}+2}_{\mu} & \exists z.\exists\eta.\mathsf{Cfg}_{\bar{x}^\alpha:A,z^\eta:\mathtt{C}}(\mathsf{Q}_1)\otimes\mathsf{Cfg}_{z^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}_2) \\
 & & & \textit{spawn} \\
5. & \mathsf{Cfg}_{\cdot,y^\beta:1}(\mathbf{close}Ry) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(y^\beta.\mathbf{closed}(_))\otimes 1 \\
6. & \mathsf{Cfg}_{x^\alpha:1,y^\beta:\mathtt{A}}(\mathbf{wait}Lx;\mathsf{Q}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(x^\alpha.\mathbf{closed}(_))\multimap\mathsf{Cfg}_{\cdot,y^\beta:\mathtt{A}}(\mathsf{Q}) \\
 & & & \textit{session type: } 1 \\
7. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\&\{\ell:\mathtt{B}_\ell\}_{\ell\in L}}(\mathbf{case}Ry(\ell\Rightarrow\mathsf{Q}_\ell)_{\ell\in L}) & =^{\mathbf{n}+2}_{\mu} & \forall w^\eta.\,\&\{\ell:\mathsf{Msg}(y^\beta.\ell(w^\eta))\multimap\mathsf{Cfg}_{\bar{x}^\alpha:\omega,w^\eta:\mathtt{B}_\ell}(\mathsf{Q}_\ell)\}_{\ell\in L} \\
8. & \mathsf{Cfg}_{x^\alpha:\&\{\ell:\mathtt{A}_\ell\}_{\ell\in L},y^\beta:\mathtt{B}}(Lx.k;\mathsf{Q}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(x^\alpha.k(x^{\alpha+1}))\otimes\mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^\beta:\mathtt{B}}(\mathsf{Q}) \\
 & & & \textit{session type: } \&\{\ell:\mathtt{A}_\ell\}_{\ell\in L} \\
9. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\oplus\{\ell:\mathtt{B}_\ell\}_{\ell\in L}}(Rw.k;\mathsf{Q}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(y^\beta.k(y^{\beta+1}))\otimes\mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^{\beta+1}:\mathtt{B}_k}(\mathsf{Q}) \\
10. & \mathsf{Cfg}_{x^\alpha:\oplus\{\ell:\mathtt{A}_\ell\}_{\ell\in L},y^\beta:\mathtt{B}}(\mathbf{case}Lx(\ell\Rightarrow\mathsf{Q}_\ell)_{\ell\in L}) & =^{\mathbf{n}+2}_{\mu} & \forall w^\eta.\&\{\ell:\mathsf{Msg}(x^\alpha.\ell(w^\eta))\multimap\mathsf{Cfg}_{w^\eta:\mathtt{A}_\ell,y^\beta:\mathtt{B}}(\mathsf{Q}_\ell)\}_{\ell\in L} \\
 & & & \textit{session type: } \oplus\{\ell:\mathtt{A}_\ell\}_{\ell\in L} \\
11. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{t}}(\mathbf{case}Ry(\nu_{\mathtt{t}}\Rightarrow\mathsf{Q})) & =^{\mathbf{n}+2}_{\mu} & \forall w^\eta.\mathsf{Msg}(y^\beta.\nu_t(w^\eta))\multimap\mathsf{Cfg}_{\bar{x}^\alpha:\omega,w^\eta:\mathtt{C}}(\mathsf{Q}) \\
12. & \mathsf{Cfg}_{x^\alpha:\mathtt{t},y^\beta:\mathtt{B}}(Lx.\nu_{\mathtt{t}};\mathsf{Q}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(x^\alpha.\nu_t(x^{\alpha+1}))\otimes\mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C},y^\beta:B}(\mathsf{Q}) \\
 & & & \textit{session type: } \mathtt{t} =^i_\nu \mathtt{C}\in\mathbf{\Sigma} \\
13. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{t}}(Ry.\mu_{\mathtt{t}};\mathsf{Q}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Msg}(y^\beta.\mu_t(y^{\beta+1}))\otimes\mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^{\beta+1}:\mathtt{C}}(\mathsf{Q}) \\
14. & \mathsf{Cfg}_{x^\alpha:\mathtt{t},y^\beta:\mathtt{B}}(\mathbf{case}Lx(\mu_{\mathtt{t}}\Rightarrow\mathsf{Q})) & =^{\mathbf{n}+2}_{\mu} & \forall w^\eta.\mathsf{Msg}(x^\alpha.\mu_t(w^\eta))\multimap\mathsf{Cfg}_{w^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \\
 & & & \textit{session type: } \mathtt{t} =^i_\mu \mathtt{C}\in\mathbf{\Sigma} \\
15. & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathsf{Y}) & =^{n+1}_{\nu} & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathsf{Q}[y/w,\overline{x}/\bar{u}]) \\
 & & & \bar{u}:\omega\vdash\mathsf{Y}=\mathsf{Q}::(w:\mathtt{B})\in V
\end{array}$$

FIGURE 7.2: Definition of predicate Cfg.

The last case (row 15 in Figure 7.2) where the predicate is a greatest fixed point is defined using an abbreviation. We can unfold this abbreviation using finitely many intermediate predicates Call_Y (for each $\mathsf{Y}\in V$) as:

$$\begin{array}{llll}
\mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathsf{Y}) & =^{\mathbf{n}+2}_{\mu} & \mathsf{Call}_Y(\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}) & \\
\mathsf{Call}_Y(\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}) & =^{n+1}_{\nu} & \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathsf{Q}[y/w,\overline{x}/\bar{u}]) & \bar{u}:\omega\vdash\mathsf{Y}=\mathsf{Q}::(w:\mathtt{B})\in V.
\end{array}$$

For a signature consisting of only two fixed point definitions $\mathsf{t}=_\mu \mathsf{A}$, $\mathsf{s}=_\mu \mathsf{B}$ and a program with two process variables $\bar{u}:\omega\vdash\mathsf{Y}=\mathsf{Q}::(w:\mathtt{B})$, and $\bar{u}:\omega'\vdash\mathsf{X}=\mathsf{P}::(w:\mathtt{C})$, the part of equivalent formula corresponding to lines 13 and 15 in Figure 7.2 is defined as:

$$\begin{array}{ll}
\mathsf{Cfg}_{\bar{x}^\alpha:d_1,y^\beta:d_2}(d_3) = & \cdots \\
 & \oplus\,\exists c_1.d_2=\mathsf{t}\otimes d_3 = Ry.\mu_{\mathsf{t}};c_1\otimes\mathsf{Msg}(y^\beta.\mu_{\mathsf{t}}(y^{\beta+1}))\otimes\mathsf{Cfg}_{\bar{x}^\alpha:d_1,y^{\beta+1}:\mathtt{A}}(c_1) \\
 & \oplus\,\exists c_1.d_2=\mathsf{s}\otimes d_3 = Ry.\mu_{\mathsf{s}};c_1\otimes\mathsf{Msg}(y^\beta.\mu_{\mathsf{s}}(y^{\beta+1}))\otimes\mathsf{Cfg}_{\bar{x}^\alpha:d_1,y^{\beta+1}:\mathtt{B}}(c_1) \\
 & \oplus\, d_3=\mathsf{X}\otimes\mathsf{Call}_X(\bar{x}^\alpha{:}d_1,y^\beta{:}d_2) \\
 & \oplus\, d_3=\mathsf{Y}\otimes\mathsf{Call}_Y(\bar{x}^\alpha{:}d_1,y^\beta{:}d_2)
\end{array}$$

7.4 A predicate for strong progress

Strong progress in an asynchronous setting states that a program eventually terminates either in an empty configuration or one attempting to receive along an external channel. In Section 5.7, we showed that recursion destroys strong progress similar to the way adding circularity to a calculus breaks down cut elimination. In this section, we introduce a predicate that formalizes the concept of strong progress for a configuration of processes in the language of $FIMALL^{\infty}_{\mu,\nu}$.

We first focus on a particular case in which the configuration is closed, i.e. it does not use any services on the left. We need to show the computational behavior of configuration $\cdot \Vdash \mathcal{C} :: (y^{\beta} : \mathtt{B})$ as defined in Figure 7.2 ensures that its external channel $y^{\beta}{:}\mathtt{B}$ will be eventually closed or blocked by waiting to receive a message. In the latter case, as soon as the message becomes available, y^{β} evolves to the continuation provided by the message (w^{η}), and the continuation has to maintain the same property. In Figure 7.3 we define a predicate $[y^{\beta} : \mathtt{B}]$ to formalize this property.

Similar to the definition of Cfg we use pattern matching to define $[y^{\alpha} : \mathtt{B}]$ (Figure 7.3). We introduce a case for each session type such that $[y^{\alpha} : \mathtt{B}]$ in each case is defined using the main connective of its underlying session type B. The first line in the definition of $[y^{\alpha} : \mathtt{B}]$ corresponds to the case in which y^{β} terminates; in this case we declare that y^{β} is closed ($\mathsf{Msg}(y^{\beta}.\mathbf{closed}(\_))$) and terminate ($\mathbf{1}$). For positive types ($\oplus$ and positive fixed points), the provider declares a message of a correct type along y^{β} while the continuation channel $y^{\beta+1}$ has to enjoy the strong progress property. For negative types ($\&$ and negative fixed points), the channel waits on a message along y^{β}. Upon receipt of such a message, the continuation channel w^{η} has to maintain the strong progress property.

The recursive definitions in all cases are defined based on the underlying session type structure. The structure of the underlying session type is itself defined using a mutual inductive and coinductive: a session type is either built using a composition of types with smaller sizes, or defined inductively as a positive fixed point, or defined coinductively as a negative fixed point. When the underlying session type is a positive fixed point $\mathtt{t}$, the predicate $[y^{\beta} : \mathtt{t}]$ is inductively defined on the structure of $\mathtt{t}$ and inherits its priority from $\mathtt{t}$. When the session type is a negative fixed point $\mathtt{t} =^{i}_{\nu} \mathtt{A}$, the predicate is defined as a greatest fixed point based on the structure of $\mathtt{t}$ and again inherits its priority from $\mathtt{t}$. In the latter case, predicate $[y^{\beta} : \mathtt{t}]$ ensures a property that is desired by strong progress: the channel y^{β} is blocked until it receives a message. In this case the property is defined coinductively since after a message along $y^{\beta}{:}\mathtt{t}$ is received, the predicate holds for the continuation.

Moreover, when the message is received, the continuation w^{η} continues to maintain the desired property. If the underlying session type is a composition of smaller ones (rows 2 and 4), the predicate is defined inductively on the size of session types. The priority $\mathbf{n}+\mathbf{2}$ refers to an induction based on size. We will see the priority assigned to each case ensures that if a program

$$\begin{array}{lll}
[y^\beta : 1] & =_\mu^{\mathbf{n}+2} & \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \otimes 1 \\
[y^\beta : \&\{\ell : \mathtt{A}_\ell\}_{\ell \in L}] & =_\mu^{\mathbf{n}+2} & \forall w^\eta.\&\{\ell : (\mathsf{Msg}(y^\beta.\ell(w^\eta)) \multimap [w^\eta : \mathtt{A}_\ell])\}_{\ell \in L} \\
[y^\beta : \mathtt{t}] & =_\nu^i & \forall w^\eta.\mathsf{Msg}(y^\beta.\nu_t(w^\eta)) \multimap [w^\eta : \mathtt{A}] \quad \mathtt{t} =_\nu^i \mathtt{A} \in \mathbf{\Sigma} \\
[y^\beta : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell \in L}] & =_\mu^{\mathbf{n}+2} & \oplus\{\ell : (\mathsf{Msg}(y^\beta.\ell(y^{\beta+1})) \otimes [y^{\beta+1} : \mathtt{A}_\ell])\}_{\ell \in L} \\
[y^\beta : \mathtt{t}] & =_\mu^i & (\mathsf{Msg}(y^\beta.\mu_t(y^\beta + 1)) \otimes [y^{\beta+1} : \mathtt{A}]) \quad \mathtt{t} =_\mu^i \mathtt{A} \in \mathbf{\Sigma}
\end{array}$$

FIGURE 7.3: Definition of predicate $[y^\beta : \mathtt{A}]$.

is guarded, then there is a valid derivation in the system of Figure 4.1 proving the predicate that formalizes strong progress.

Next, we briefly explain how to convert the definition of Figure 7.3 to a formula in the language of $FIMALL^\infty_{\mu,\nu}$. For $\mathtt{t} =_\nu^i \mathtt{A} \in \mathbf{\Sigma}$, the definition $[y^\beta : \mathtt{t}]$ is an abbreviation. We can unfold the abbreviation, using finitely many intermediate predicates $\mathsf{Unfold}_\mathtt{t}$ (for each $t =_\nu \mathtt{A} \in \mathbf{\Sigma}$):

$$\begin{array}{lll}
[y^\beta : \mathtt{t}] & =_\mu^{n+2} & \mathsf{Unfold}_\mathtt{t}(y^\beta) \\
\mathsf{Unfold}_\mathtt{t}(y^\beta) & =_\nu^i & \forall w^\eta.\mathsf{Msg}(y^\beta.\nu_t(w^\eta)) \multimap [w^\eta : \mathtt{A}] \qquad \mathtt{t} =_\nu^i \mathtt{A} \in \mathbf{\Sigma}
\end{array}$$

It means that we do not have a single closed encoding of session-typed programs and strong progress, but we have a different encoding for every signature $\mathbf{\Sigma}$. Having this abbreviation helps us in the proof of the strong progress theorem: we can assign a matching generational variable $\mathbf{y}^\beta$ to every predicate $[y^\beta : \mathtt{t}]$ occurring in the derivation. The structure of the derivation or its validity won't be affected by the abbreviation, but it ensures that the generation of variables steps at the same pace as the generation of channels (see the proof of Lemma 7.3).

Using the predicate defined in Figure 7.3, strong progress for closed configuration $\cdot \vdash \mathcal{C} :: (y^\beta{:}B)$ is formalized as

$$\mathsf{Cfg}_{\cdot,y^\beta:B}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}].$$

The generalization of this judgment to an open configuration $x^\alpha : \mathtt{A} \Vdash \mathcal{C} :: (y^\beta : \mathtt{B})$ is straightforward:

$$[x^\alpha : \mathtt{A}], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$$

Configuration $\mathcal{C}$ satisfies strong progress if assuming strong progress for $x^\alpha : \mathtt{A}$ we can prove strong progress of $y^\beta : \mathtt{B}$.

7.5 A direct proof for strong progress

In this section, we give a direct proof for the strong progress property of guarded programs with *asynchronous communication*. The major steps of the proof are as follows:

1. For a configuration of processes $\bar{x}^\alpha : \omega \vdash \mathcal{C} :: (y^\beta : \mathtt{B})$, we show that there is a (possibly infinite) derivation for $[\bar{x}^\alpha : \omega], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:B}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$ (Lemma 7.3).

2. We show that when defined over a guarded program the derivation introduced in Step 1. is a valid proof. The idea is to introduce a validity-preserving bisimulation between the annotated derivation given in Lemma 7.3 and the typing derivation of processes.

3. Finally, we show that a valid cut-free proof for $[\bar{x}^\alpha : \omega], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:B}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$, ensures strong progress of a configuration of guarded processes $\bar{x}^\alpha : \omega \Vdash \mathcal{C} :: y^\beta : \mathtt{B}$ when executed with a synchronous scheduler (Theorem 7.8).

The detailed proof for each step is given below.

Lemma 7.3. *For a configuration of processes $\bar{x}^\alpha : \omega \vdash \mathcal{C} :: (y^\beta : \mathtt{B})$, there is a (possibly infinite) derivation for $[\bar{x}^\alpha : \omega], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:B}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$.*

Proof. For an open configuration $x^\alpha : \mathtt{A} \vdash \mathcal{C} :: y^\beta$:B, it is enough to build a circular derivation for

$$\star\, [x^\alpha : \mathtt{A}], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}].$$

If the configuration is closed, it is enough to build a derivation $\star\ \mathsf{Cfg}_{\cdot,y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta$:B] for a closed configuration $\cdot \vdash \mathcal{C} :: y^\beta : \mathtt{B}$. For the sake of brevity we write $\star\, \overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$ as a generalization for both open and closed configurations, where $\overline{[\bar{x}^\alpha : \omega]}$ is either empty or $[\bar{x}^\alpha : \omega]$, and it is empty if and only if $\bar{x} : \omega$ is empty. We provide a circular derivation for each possible pattern of $\mathcal{C}$. Here are the circular derivations when $\mathcal{C}$ is an empty configuration and a composition of configurations, respectively:

$$\dfrac{\dfrac{\dfrac{}{[x^\alpha : \mathtt{A}] \vdash [x^\alpha : \mathtt{A}]}\,\text{ID}}{[x^\alpha : \mathtt{A}], 1 \vdash [x^\alpha : \mathtt{A}]}\,1L}{\star\ [x^\alpha : \mathtt{A}], \mathsf{Cfg}_{x^\alpha:\mathtt{A},x^\alpha:\mathtt{A}}(\cdot) \vdash [x^\beta : \mathtt{A}]}\,\mu L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{\star}{\overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega,z^\zeta:\mathtt{C}}(\mathcal{C}_1) \vdash [z^\zeta : \mathtt{C}]} \quad \overset{\star}{[z^\zeta : \mathtt{C}], \mathsf{Cfg}_{z^\zeta:\mathtt{C},y^\beta:\mathtt{B}}(\mathcal{C}_2) \vdash [y^\beta : \mathtt{B}]}}{\overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega,z^\zeta:\mathtt{C}}(\mathcal{C}_1), \mathsf{Cfg}_{z^\zeta:\mathtt{C},y^\beta:\mathtt{B}}(\mathcal{C}_2)) \vdash [y^\beta : \mathtt{B}]}\,\textsc{Cut}}{\overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega,z^\zeta:\mathtt{C}}(\mathcal{C}_1) \otimes \mathsf{Cfg}_{z^\zeta:\mathtt{C},y^\beta:\mathtt{B}}(\mathcal{C}_2)) \vdash [y^\beta : \mathtt{B}]}\,\otimes L}{\overline{[\bar{x}^\alpha : \omega]}, \exists z.\exists\zeta.(\mathsf{Cfg}_{\bar{x}^\alpha:\omega,z^\zeta:\mathtt{C}}(\mathcal{C}_1) \otimes \mathsf{Cfg}_{z^\zeta:\mathtt{C},y^\beta:\mathtt{B}}(\mathcal{C}_2)) \vdash [y^\beta : \mathtt{B}]}\,\exists L}{\star\ \overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega,y^\beta:\mathtt{B}}(\mathcal{C}_1 \mid_{z:\mathtt{C}} \mathcal{C}_2) \vdash [y^\beta : \mathtt{B}]}\,\mu L$$

The derivation built above is based on the definition of the strong progress formula by pattern matching. We briefly explain how to unfold this derivation to an infinite definition in the system of Figure 4.6 without pattern matching. Consider the first two lines in expanded definition of predicate Cfg:

$$\begin{aligned}\mathsf{Cfg}_{\bar{x}^\alpha:d_1,y^\beta:d_2}(d_3) = \quad & (\exists c_1, c_2, z, c.\, (d_3 = c_1|_{z:c}c_2) \otimes \exists z.\exists\eta.\mathsf{Cfg}_{\bar{x}^\alpha:d_1,z^\eta:c}(c_1) \otimes \mathsf{Cfg}_{z^\eta:c,y^\beta:d_2}(c_2)) \\ & \oplus((d_3 = \cdot) \otimes 1) \oplus \cdots\end{aligned}$$

To prove the judgment $\star[x^\alpha : \mathtt{A}], \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$ without pattern matching, we first need to unfold the definition of $\mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}$ with a μL rule. Next, we apply an $\oplus L$ rule on the resulting formula which is the right side of the above definition. After this step, we are in a quite similar situation to the pattern matching argument: we have to prove several branches each corresponding to a pattern of $\mathcal{C}$. In some cases, we may need to apply extra equality and $\oplus$ rules in the derivation without pattern matching. However, for all cases the fixed point rules applied in a cycle are the same in the derivations built with and without pattern matching.

For the cases in which the configuration consists of a single process we give an annotated derivation with generational variables and track the relationship between their generations. The annotations of infinitary derivations in $FIMALL^{\infty}_{\mu,\nu}$ are introduced in Chapter 4.

Without loss of generality, we annotate predicates of the form $[z^\eta : \mathtt{C}]$ with a matching generational variable $\mathbf{z}^\eta$ at the start (the bottom) of each cycle. We leave it to the reader to check that this assumption holds as an invariant at the end (the top) of each cycle in the derivation.

Case 1. $(y \leftarrow x)$

$$\dfrac{\dfrac{\dfrac{}{\mathbf{x}^\alpha : [y^\beta : \mathtt{A}] \vdash_{\Lambda_1} \mathbf{y}^\beta : [y^\beta : \mathtt{A}]}\,\mathrm{ID}}{\mathbf{x}^\alpha : [x^\alpha : \mathtt{A}], \mathbf{c}^{\eta+1} : (x^\alpha = y^\beta) \vdash_{\Lambda_1} \mathbf{y}^\beta : [y^\beta : \mathtt{A}]}\,{=}L}{\star\, \mathbf{x}^\alpha : [x^\alpha : \mathtt{A}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{A}, y^\beta:\mathtt{A}}(y \leftarrow x) \vdash_\Lambda \mathbf{y}^\beta : [y^\beta : \mathtt{A}]}\,\mu L$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_i = \mathbf{c}^\eta_i \mid i \neq n+2\}.$

Case 2. $(\mathtt{close}\, Ry)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{w}^\delta : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \vdash_{\Lambda_4} \mathbf{z}^\gamma : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_))}\,\mathrm{ID} \qquad \dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\Lambda_5} \mathbf{y}^{\beta+2} : 1}\,1R}{\mathbf{c}^{\eta+1} : 1 \vdash_{\Lambda_5} \mathbf{y}^{\beta+2} : 1}\,1L}{\mathbf{c}^{\eta+1} : 1 \vdash_{\Lambda_3} \mathbf{y}^{\beta+1} : [\cdot : \cdot]}\,\mu R}{\mathbf{w}^\delta : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)), \mathbf{c}^{\eta+1} : 1 \vdash_{\Lambda_3} \mathbf{y}^{\beta+1} : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \otimes [\cdot : \cdot]}\,\otimes R}{\mathbf{c}^{\eta+1} : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \otimes 1 \vdash_{\Lambda_2} \mathbf{y}^{\beta+1} : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \otimes [\cdot : \cdot]}\,\otimes L}{\mathbf{c}^{\eta+1} : \mathsf{Msg}(y^\beta.\mathbf{closed}(\_)) \otimes 1 \vdash_{\Lambda_1} \mathbf{y}^\beta : [y^\beta : 1]}\,\mu R}{\star\, \mathbf{c}^\eta : \mathsf{Cfg}_{\cdot, y^\beta:1}(\mathtt{close}\, Ry) \vdash_\Lambda \mathbf{y}^\beta : [y^\beta : 1]}\,\mu L$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^\eta_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_i = \mathbf{c}^\eta_i \mid i \neq n+2\}$, $\Lambda_2 = \Lambda_1 \cup \{\mathbf{y}^{\beta+1}_{n+2} < \mathbf{y}^\beta_{n+2}\} \cup \{\mathbf{y}^{\beta+1}_i = \mathbf{y}^\beta_i \mid i \neq n+2\}$ $\Lambda_3 = \Lambda_2 \cup \{\mathbf{c}^{\eta+1} = \mathbf{w}^\delta\}$, $\Lambda_4 = \Lambda_3 \cup \{\mathbf{y}^{\beta+1} = \mathbf{z}^\gamma\}$, $\Lambda_5 = \Lambda_3 \cup \{\mathbf{y}^{\beta+2}_{n+2} < \mathbf{y}^{\beta+1}_{n+2}\} \cup \{\mathbf{y}^{\beta+2}_i = \mathbf{y}^{\beta+1}_i \mid i \neq n+2\}$

Case 3. $(\mathtt{wait}\, Lx; \mathsf{Q})$

$$
\dfrac{
\dfrac{
\dfrac{}{\mathtt{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)) \vdash_{\Lambda_3} \mathbf{z}^{\kappa} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_))}\textsc{Id}
\quad
\dfrac{\dfrac{\dfrac{\star}{\mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_4} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}}{\mathbf{x}^{\alpha+2} : 1, \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_4} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}1L}{\mathbf{x}^{\alpha+1} : [\cdot : \cdot], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
}{
\dfrac{\mathtt{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)), \mathbf{x}^{\alpha+1} : [\cdot : \cdot], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)) \multimap \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)) \otimes [\cdot : \cdot], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)) \multimap \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\mathbf{x}^{\alpha} : [x^{\alpha} : 1], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.\mathbf{closed}(\_)) \multimap \mathsf{Cfg}_{\cdot,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{\star\, \mathbf{x}^{\alpha} : [x^{\alpha} : 1], \mathbf{c}^{\eta} : \mathsf{Cfg}_{x^{\alpha}:1,y^{\beta}:\mathtt{B}}(\mathtt{wait}\, Lx; \mathsf{Q}) \vdash_{\Lambda} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
}\mu L
}\otimes L
}\multimap L
}{}
$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_{i} = \mathbf{c}^{\eta}_{i} \mid i \neq n+2\}$, $\Lambda_2 = \Lambda_1 \cup \{\mathbf{x}^{\alpha+1}_{n+2} < \mathbf{x}^{\alpha}_{n+2}\} \cup \{\mathbf{x}^{\alpha+1}_{i} = \mathbf{x}^{\alpha}_{i} \mid i \neq n+2\}$, $\Lambda_3 = \Lambda_2 \cup \{\mathbf{x}^{\alpha+1} = \mathbf{w}^{\omega}\}$, $\Lambda_4 = \Lambda_3 \cup \{\mathbf{x}^{\alpha+2}_{n+2} < \mathbf{x}^{\alpha+1}_{n+2}\} \cup \{\mathbf{x}^{\alpha+2}_{i} = \mathbf{x}^{\alpha+1}_{i} \mid i \neq n+2\}$

Case 4. $((z : \mathtt{C}) \leftarrow \mathsf{Q}_1; \mathsf{Q}_2)$

$$
\dfrac{
\dfrac{\star}{\overline{\mathbf{x}^{\alpha} : [\bar{x}^{\alpha} : \omega]}, \mathbf{w}^{\omega} : \mathsf{Cfg}_{\bar{x}^{\alpha}:\omega,z^{\zeta}:\mathtt{C}}(\mathsf{Q}_1) \vdash_{\Lambda_2} \mathbf{z}^{\zeta} : [z^{\zeta} : \mathtt{C}]}
\quad
\dfrac{\star}{\mathbf{z}^{\zeta} : [z^{\zeta} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{z^{\zeta}:\mathtt{C},y^{\beta}:\mathtt{B}}(\mathsf{Q}_2) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}
}{
\dfrac{\overline{\mathbf{x}^{\alpha} : [\bar{x}^{\alpha} : \omega]}, \mathbf{w}^{\omega} : \mathsf{Cfg}_{\bar{x}^{\alpha}:\omega,z^{\zeta}:\mathtt{C}}(\mathsf{Q}_1), \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{z^{\zeta}:\mathtt{C},y^{\beta}:\mathtt{B}}(\mathsf{Q}_2)) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\overline{\mathbf{x}^{\alpha} : [\bar{x}^{\alpha} : \omega]}, \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\bar{x}^{\alpha}:\omega,z^{\zeta}:\mathtt{C}}(\mathsf{Q}_1) \otimes \mathsf{Cfg}_{z^{\zeta}:\mathtt{C},y^{\beta}:\mathtt{B}}(\mathsf{Q}_2)) \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\overline{\mathbf{x}^{\alpha} : [\bar{x}^{\alpha} : \omega]}, \mathbf{c}^{\eta+1} : \exists z.\exists\zeta.(\mathsf{Cfg}_{\bar{x}^{\alpha}:\omega,z^{\zeta}:\mathtt{C}}(\mathsf{Q}_1) \otimes \mathsf{Cfg}_{z^{\zeta}:\mathtt{C},y^{\beta}:\mathtt{B}}(\mathsf{Q}_2)) \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{\star\, \overline{\mathbf{x}^{\alpha} : [\bar{x}^{\alpha} : \omega]}, \mathbf{c}^{\eta} : \mathsf{Cfg}_{\bar{x}^{\alpha}:\omega,y^{\beta}:\mathtt{B}}((z : \mathtt{C}) \leftarrow \mathsf{Q}_1; \mathsf{Q}_2) \vdash_{\Lambda} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
}\exists L
}\otimes L
}\textsc{Cut}
$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_{i} = \mathbf{c}^{\eta}_{i} \mid i \neq n+2\}$ *and* $\Lambda_2 = \Lambda_1 \cup \{\mathbf{w}^{\omega} = \mathbf{c}^{\eta+1}\}$.

Case 5.$(Lx.k; \mathsf{Q})$

$$
\dfrac{
\dfrac{}{\mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \vdash_{\Lambda_3} \mathbf{z}^{\kappa} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1}))}\textsc{Id}
\quad
\dfrac{\star}{\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{A}_k], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}
}{
\dfrac{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \multimap [x^{\alpha+1} : \mathtt{A}_k], \mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})), \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \multimap [x^{\alpha+1} : \mathtt{A}_k]\}, \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\mathbf{x}^{\alpha+1} : \forall w^{\eta}.\&\{\ell : \mathsf{Msg}(\ell, x^{\alpha+1} : \mathtt{A}_{\ell}(w^{\eta})) \multimap [w^{\eta} : A_{\ell}]\}_{\ell \in L}, \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{
\dfrac{\mathbf{x}^{\alpha} : [x^{\alpha} : \&\{\ell : \mathtt{A}_{\ell}\}_{\ell \in L}], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k,y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{\star\, \mathbf{x}^{\alpha} : [x^{\alpha} : \&\{\ell : \mathtt{A}_{\ell}\}_{\ell \in L}], \mathbf{c}^{\eta} : \mathsf{Cfg}_{x^{\alpha}:\&\{\ell:\mathtt{A}_{\ell}\}_{\ell \in L},y^{\beta}:\mathtt{B}}(Lx.k; \mathsf{Q}) \vdash_{\Lambda} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
}\mu L
}\forall L/\& L
}\otimes L
}\multimap L
$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_{i} = \mathbf{c}^{\eta}_{i} \mid i \neq n+2\}$, $\Lambda_2 = \Lambda_1 \cup \{\mathbf{x}^{\alpha+1}_{n+2} < \mathbf{x}^{\alpha}_{n+2}\} \cup \{\mathbf{x}^{\alpha+1}_{i} = \mathbf{x}^{\alpha}_{i} \mid i \neq n+2\}$, $\Lambda_3 = \Lambda_2 \cup \{\mathbf{c}^{\eta+1} = \mathbf{w}^{\omega}\}$.

Case 6. $(Ry.k; \mathsf{Q})$ Dual to **Case 5**.

Case 7. $(\mathbf{case}Lx(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L})$

$$
\cfrac{\cfrac{\forall k \in L \quad \cfrac{\cfrac{\cfrac{\cfrac{}{\mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \vdash_{\Lambda_3} \mathbf{z}^{\zeta} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1}))}\textsc{Id} \quad \overset{\star}{\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{A}_k], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k, y^{\beta}:\mathtt{B}}(\mathsf{Q}_k) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}}{\mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})), \mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{A}_k], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k, y^{\beta}:\mathtt{B}}(\mathsf{Q}_k) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\multimap L}{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{A}_k], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_k, y^{\beta}:\mathtt{B}}(\mathsf{Q}_k) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\otimes L}{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.k(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{A}_k], \mathbf{c}^{\eta+1} : \&\{\ell : \mathsf{Msg}(x^{\alpha}.\ell(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}_\ell, y^{\beta}:\mathtt{B}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\forall L/\& L}{\cfrac{\mathbf{x}^{\alpha+1} : \oplus\{\ell : \mathsf{Msg}(x^{\alpha}.\ell(x^{\alpha+1})) \otimes [x^{\alpha+1} : A_\ell]\}_{\ell\in L}, \mathbf{c}^{\eta+1} : \forall w^{\eta}.\&\{\ell : \mathsf{Msg}(x^{\alpha}.\ell(w^{\eta})) \multimap \mathsf{Cfg}_{e^{\eta}:\mathtt{A}_\ell, y^{\beta}:\mathtt{B}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}{\mathbf{x}^{\alpha} : [x^{\alpha} : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \mathbf{c}^{\eta+1} : \forall w^{\eta}.\&\{\ell : \mathsf{Msg}(x^{\alpha}.\ell(w^{\eta})) \multimap \mathsf{Cfg}_{w^{\eta}:\mathtt{A}_\ell, y^{\beta}:\mathtt{B}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L}\oplus L}{\star\ \mathbf{x}^{\alpha} : [x^{\alpha} : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \mathbf{c}^{\eta} : \mathsf{Cfg}_{x^{\alpha}:\oplus\{\ell:A_\ell\}_{\ell\in L}, y^{\beta}:\mathtt{B}}(\mathbf{case}Lx(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L}) \vdash_{\Lambda} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_{i} = \mathbf{c}^{\eta}_{i} \mid i \neq n+2\}$, $\Lambda_2 = \Lambda_1 \cup \{\mathbf{x}^{\alpha+1}_{n+2} < \mathbf{x}^{\alpha}_{n+2}\} \cup \{\mathbf{x}^{\alpha+1}_{i} = \mathbf{x}^{\alpha}_{i} \mid i \neq n+2\}$, $\Lambda_3 = \Lambda_2 \cup \{\mathbf{x}^{\alpha+1} = \mathbf{w}^{\omega}\}$.

Case 8. $(\mathbf{case}Ry(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L})$ Dual to **Case 7**.

Case 9. $(Lx.\nu_{\mathtt{t}}; \mathsf{Q})$ where $\mathtt{t} =^{k}_{\nu} \mathtt{C}$

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})) \vdash_{\Lambda_3} \mathbf{z}^{\zeta} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1}))}\mathrm{ID} \quad \overset{\star}{\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}}{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})) \multimap [x^{\alpha+1} : \mathtt{C}], \mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})), \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\multimap L}{\mathbf{x}^{\alpha+1} : \forall w^{\eta}.\mathsf{Msg}(x^{\alpha}.\nu_t(w^{\eta})) \multimap [w^{\eta} : \mathtt{C}], \mathbf{w}^{\omega} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})), \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\forall L}{\mathbf{x}^{\alpha+1} : \forall w^{\eta}.\mathsf{Msg}(x^{\alpha}.\nu_t(w^{\eta})) \multimap [w^{\eta} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_2} \mathbf{y}^{\beta} : [y^{\beta} : B]}\otimes L}{\mathbf{x}^{\alpha} : [x^{\alpha} : \mathtt{t}], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^{\alpha}.\nu_t(x^{\alpha+1})) \otimes \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^{\beta}:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_1} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\nu L}{\star\ \mathbf{x}^{\alpha} : [x^{\alpha} : \mathtt{t}], \mathbf{c}^{\eta} : \mathsf{Cfg}_{x^{\alpha}:\mathtt{t}, y^{\beta}:B}(Lx.\nu_{\mathtt{t}}; \mathsf{Q}) \vdash_{\Lambda} \mathbf{y}^{\beta} : [y^{\beta} : \mathtt{B}]}\mu L
$$

$\Lambda_2 = \Lambda_1 \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^{\eta}_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_{i} = \mathbf{c}^{\eta}_{i} \mid i \neq n+2\}$, $\Lambda_1 = \Lambda \cup \{\mathbf{x}^{\alpha+1}_{i} = \mathbf{x}^{\alpha}_{i} \mid i \neq k\}$, $\Lambda_3 = \Lambda_2 \cup \{\mathbf{c}^{\eta+1} = \mathbf{w}^{\omega}\}$.

In the proof of this case, we use the abbreviated definition of $[x^{\alpha} : \mathtt{t}]$. If we use the following expanded definition instead

$$
\begin{array}{lll}
[y^{\beta} : \mathtt{t}] & =^{n+2}_{\mu} & \mathsf{Unfold}_{\mathtt{t}}(y^{\beta}) \\
\mathsf{Unfold}_{\mathtt{t}}(y^{\beta}) & =^{i}_{\nu} & \forall w^{\eta}.(\mathsf{Msg}(y^{\beta}.\nu_t(w^{\eta})) \multimap [w^{\eta} : \mathtt{A}]),
\end{array}
$$

we need to apply an extra μL rule with priority $n+2$ on the predicate $[x^\alpha : \mathtt{t}]$. Immediately after this μL rule, we apply the νL rule with priority $k < n+1$. This implies that the extra μL rule does not play a role in validity of the derivation. As a result, the validity of the derivation given here implies validity of the derivation using the non-abbreviated definition. Furthermore, if we use the non-abbreviated definition we will have $\mathbf{x}^{\alpha+2} : [x^{\alpha+1} : \mathtt{C}]$ at the end (the top) of the cycle. Thus, we decided to use the abbreviated definition to make sure that the generation of position variables steps at the same pace as the generation of channels. This decision will help us to describe the bisimulation between the derivation built here and the typing derivation of processes in a more elegant way.

Case 10. $(Ry.\mu_{\mathtt{t}};\mathsf{Q})$ Dual to **Case 9.**

Case 11. $(\mathbf{case}Lx(\mu_{\mathtt{t}} \Rightarrow \mathsf{Q}))$ where $\mathtt{t} =^k_\mu \mathtt{C}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{w}^\omega : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1})) \vdash_{\Lambda_3} \mathbf{z}^\zeta : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1}))}\,\text{ID} \qquad \dfrac{\star}{\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}}{\mathbf{w}^\omega : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1})), \mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,{\multimap L}}{\mathbf{w}^\omega : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1})), \mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \forall w^\eta.\mathsf{Msg}(x^\alpha.\mu_t(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_3} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,\forall LL}{\mathbf{x}^{\alpha+1} : \mathsf{Msg}(x^\alpha.\mu_t(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \forall w^\eta.\mathsf{Msg}(x^\alpha.\mu_t(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_2} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,{\otimes L}}{\mathbf{x}^\alpha : [x^\alpha : \mathtt{t}], \mathbf{c}^{\eta+1} : \forall w^\eta.\mathsf{Msg}(x^\alpha.\mu_t(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{C},y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda_1} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,\mu L}{\star\, \mathbf{x}^\alpha : [x^\alpha : \mathtt{t}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{t},y^\beta:\mathtt{B}}(\mathbf{case}Lx(\mu_{\mathtt{t}} \Rightarrow \mathsf{Q})) \vdash_\Lambda \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,\mu L$$

$\Lambda_1 = \Lambda \cup \{\mathbf{c}^{\eta+1}_{n+2} < \mathbf{c}^\eta_{n+2}\} \cup \{\mathbf{c}^{\eta+1}_i = \mathbf{c}^\eta_i \mid i \neq n{+}2\}$, $\Lambda_2 = \Lambda_1 \cup \{\mathbf{x}^{\alpha+1}_k < \mathbf{x}^\alpha_k\} \cup \{\mathbf{x}^{\alpha+1}_i = \mathbf{x}^\alpha_i \mid i \neq k\}$, $\Lambda_3 = \Lambda_2 \cup \{\mathbf{x}^{\alpha+1} = \mathbf{w}^\omega\}$.

Case 12 $(\mathbf{case}Ry(\nu_{\mathtt{t}} \Rightarrow \mathsf{Q}))$ Dual to **Case 11**.

Case 13. $(y \leftarrow \mathsf{Y} \leftarrow x)$

$$\dfrac{\dfrac{\star}{\mathbf{x}^\alpha : [x^\alpha : \mathtt{A}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}(Q) \vdash_{\Lambda \cup \{c^{\eta+1}_i = c^\eta_i \mid i \neq n+1\}} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}}{\star\, \mathbf{x}^\alpha : [x^\alpha : \mathtt{A}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{A},y^\beta:\mathtt{B}}(y \leftarrow \mathsf{Y} \leftarrow x) \vdash_\Lambda \mathbf{y}^\beta : [y^\beta : \mathtt{B}]}\,\nu L$$

In cases 1-13 a predicate annotated with a position variable $\mathbf{c}$ in a branch can be interpreted as the (potential) computational continuation of the predicate $\mathsf{Cfg}()$ in the conclusion (at the bottom) of the block. Also, the only predicates that occur as a cut formula are of the form $[x^\alpha : \mathtt{A}]$.

Furthermore, in all the cases a rule is applied on $[x^\alpha : \mathtt{A}]$ only if there is a process in the antecedents willing to receive or send a message along channels x or y via a Msg predicate: there is a predicate in the antecedents of the form $\mathsf{Msg}(x^\alpha.b(x^{\alpha+1})) \otimes \mathsf{Cfg}_{\_\_}(P)$ or $\forall w^\eta.\&\{\mathsf{Msg}(x^\alpha.b_\ell(w^\eta)) \multimap \mathsf{Cfg}_{\_\_}(P_\ell)\}_{\ell \in L}$ (or $\forall w^\eta.\mathsf{Msg}(x^\alpha.b(w^\eta)) \multimap \mathsf{Cfg}_{\_\_}(P)$).

Observe that in each circular branch a position variable $\mathbf{v}^\delta$ annotating a predicate $[v^\delta : \mathtt{D}]$ is only related by $<$ with it a future or previous generation of itself $\mathbf{v}^\gamma$. These observations are important in particular in the proof of Theorem 7.8. □

We need to show that when defined over a guarded program the derivation introduced above is a valid proof. Since a configuration is always finite, it is enough to prove validity of the annotated derivation built using Cases 1-13 for a single process P.

We use a validity-preserving bisimulation between the annotated derivation and the typing derivation of process P. The notation $\Omega \vDash a \leq b$ stands for "the relation $a \leq b$ can be deduced from the reflexive transitive closure of set Ω".

Definition 7.4. Define relation $\mathcal{R}$ between process typing judgments

$$y^\beta : B \vdash_\Omega \mathtt{P} :: (x^\alpha : A)$$

defined over $\mathbf{\Sigma}$ and annotated sequents in $\mathit{FIMALL}^\infty_{\mu,\nu}$:

$$\begin{array}{lll} x^\alpha : \mathtt{A} \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) & \mathcal{R} & \mathbf{x}^\alpha : [x^\alpha : \mathtt{A}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{A}, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}], \\ \cdot \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) & \mathcal{R} & \mathbf{c}^\eta : \mathsf{Cfg}_{\cdot, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}], \end{array}$$

where[3] for $i \leq n$,

- $\Omega \vDash x_i^\alpha \leq w_i^\zeta$ iff $\Omega' \vDash \mathbf{x}_i^\alpha \leq \mathbf{w}_i^\zeta$, and
- $\Omega \vDash y_i^\beta \leq w_i^\zeta$ iff $\Omega' \vDash \mathbf{y}_i^\beta \leq \mathbf{w}_i^\zeta$.

We define two stepping rules over the typing derivation of a process and the derivation built in the proof of Lemma 7.3.

Definition 7.5. We define two stepping rules $\hookrightarrow$ and $\Rightarrow$ over processes and annotated sequents, respectively:

- $\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) \hookrightarrow \bar{z}^\delta : \omega' \vdash_\Lambda \mathsf{Q} :: (w^\gamma : \mathtt{D})$ iff there is a rule in the infinitary system of Figure 7.1 of the form

$$\frac{\cdots \quad \bar{z}^\delta : \omega' \vdash_\Lambda \mathsf{Q} :: (w^\gamma : \mathtt{D}) \quad \cdots}{\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B})}$$

- $$\begin{array}{ll} (i)\overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^{\eta_1} : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_\Omega \mathbf{y}^\beta : [y^\beta : \mathtt{B}] & \Rightarrow \\ (ii)\ \ \overline{\mathbf{z}^\delta : [\bar{z}^\delta : \omega']}, \mathbf{d}^{\eta_2} : \mathsf{Cfg}_{\bar{z}^\delta:\omega', w^\gamma:\mathtt{D}}(\mathsf{Q}) \vdash_\Lambda \mathbf{w}^\gamma : [w^\gamma : \mathtt{D}] & \end{array}$$

  iff (i) is the conclusion of one of the blocks 1-13 in the proof of Lemma 7.3 and (ii) is a $\star$ assumption of it.

[3] n is the maximum priority of fixed points in $\mathbf{\Sigma}$.

Next we prove that relation $\mathcal{R}$ is a validity preserving bisimulation with regard to $\hookrightarrow$ and $\Rightarrow$ transitions.

Lemma 7.6. *$\mathcal{R}$ forms a bisimulation between the derivation given for*

$$\star\,\overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$$

and the typing derivation of process

$$\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}).$$

Proof. The proof of this bisimulation is straightforward. It follows from the way we built proof blocks for each case in Lemma 7.3, and the typing rules for annotated processes in Figure 7.1:

$\rightarrow$**:**

$$\begin{tikzcd}
\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) \arrow[r, dash, "\mathcal{R}"] \arrow[d, hook] & \overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \arrow[d, Rightarrow, dashed] \\
\bar{z}^\gamma : \omega' \vdash_\Lambda \mathsf{Q} :: (w^\delta : \mathtt{D}) \arrow[r, dash, dashed, "\mathcal{R}"'] & \overline{\mathbf{z}^\gamma : [\bar{z}^\gamma : \omega']}, \mathbf{d}^\theta : \mathsf{Cfg}_{\bar{z}^\gamma:\omega', w^\delta:\mathtt{D}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{w}^\delta : [w^\delta : \mathtt{D}]
\end{tikzcd}$$

The proof is by considering different cases for $\hookrightarrow$:

Case 1. ($\mathbf{wait}Lx; \mathsf{Q}$)

1.Premise	$x^\alpha : 1 \vdash_\Omega \mathbf{wait}Lx; \mathsf{Q} :: y^\beta : \mathtt{B} \hookrightarrow \cdot \vdash_\Omega \mathsf{Q} :: y^\beta : \mathtt{B}$
2.Case 3 of Lemma 7.3	$\mathbf{x}^\alpha : [x^\alpha : 1], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:1, y^\beta:B}(\mathbf{wait}Lx; \mathsf{Q}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \Rightarrow$ $\mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot, y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$
3.Definition of Λ'	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$.
4.By assumption and line 3	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega \vDash y_i^\beta \leq z_i^\gamma$
5.By line 4	$\cdot \vdash_\Omega \mathsf{Q} :: y^\beta : \mathtt{B} \; \mathcal{R} \; \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot, y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$

Case 2. ($Lx.\nu_{\mathtt{t}}; \mathsf{Q}$)

1.Premise	$x^\alpha : \mathtt{t} \vdash_\Omega \mathtt{Lx}.\nu_{\mathtt{t}}; \mathsf{Q} :: y^\beta : \mathtt{B} \hookrightarrow x^{\alpha+1} : \mathtt{C} \vdash_\Lambda Q :: y^\beta : \mathtt{B}$
2.Case 9 of Lemma 7.3	$\mathbf{x}^\alpha : [x^\alpha : \mathtt{t}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{t}, y^\beta:\mathtt{B}}(Lx.\nu_{\mathtt{t}}; \mathsf{Q}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \Rightarrow$ $\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$
3.Definition of Λ'	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$.
4.Definition of Λ	for $i \leq n, \Lambda \vDash y_i^\beta \leq z_i^\gamma$ iff $\Omega \vDash y_i^\beta \leq z_i^\gamma$.
5.By assumption	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Lambda \vDash y_i^\beta \leq z_i^\gamma$
6.Definition of Λ'	for $i \leq n$ and $z^\gamma \neq x^{\alpha+1}, \Lambda' \vDash \mathbf{x}_i^{\alpha+1} \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{x}_i^\alpha \leq \mathbf{z}_i^\gamma$.
7.Definition of Λ	for $i \leq n,$ and $z^\gamma \neq x^{\alpha+1}, \Lambda \vDash x_i^{\alpha+1} \leq z_i^\gamma$ iff $\Omega \vDash x_i^\alpha \leq z_i^\gamma$.
8.By assumption	for $i \leq n, \Lambda' \vDash \mathbf{x}_i^{\alpha+1} \leq \mathbf{z}_i^\gamma$ iff $\Lambda \vDash x_i^{\alpha+1} \leq z_i^\gamma$
9.By lines 5 and 8	$x^{\alpha+1} : \mathtt{C} \vdash_\Lambda Q :: y^\beta : \mathtt{B} \; \mathcal{R} \; \mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^\beta:\mathtt{B}}(Q) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$

Cases. The proof of other cases is similar to the previous ones.

$\leftarrow$.

$$\begin{tikzcd}
\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) \arrow[r, dash, "\mathcal{R}"] \arrow[d, hook, dashed] & \overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \arrow[d, Rightarrow] \\
\bar{z}^\gamma : \omega' \vdash_\Lambda \mathsf{Q} :: (w^\delta : \mathtt{D}) \arrow[r, dash, dashed, "\mathcal{R}"'] & \overline{\mathbf{z}^\gamma : [\bar{z}^\gamma : \omega']}, \mathbf{d}^\theta : \mathsf{Cfg}_{\bar{z}^\gamma:\omega', w^\delta:\mathtt{D}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{w}^\delta : [w^\delta : D]
\end{tikzcd}$$

The proof is by considering different cases for $\Rightarrow$.

Case 1. $(\mathbf{wait}Lx; \mathsf{Q})$

1.Premise	$\mathbf{x}^\alpha : [x^\alpha : 1], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:1, y^\beta:B}(\mathbf{wait}Lx; \mathsf{Q}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \Rightarrow$ $\mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot, y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$
2.By $1L$ typing rule	$x^\alpha : 1 \vdash_\Omega \mathbf{wait}Lx; \mathsf{Q} :: y^\beta : \mathtt{B} \hookrightarrow \cdot \vdash_\Omega \mathsf{Q} :: y^\beta : \mathtt{B}$
3.Definition of Λ'	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$.
4.By assumption and line 3	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega \vDash y_i^\beta \leq z_i^\gamma$
5.By line 4	$\cdot \vdash_\Omega \mathsf{Q} :: y^\beta : \mathtt{B} \; \mathcal{R} \; \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{\cdot, y^\beta:\mathtt{B}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$

Case 2. $(Lx.\nu_t; \mathsf{Q})$

1.Premise	$\mathbf{x}^\alpha : [x^\alpha : \mathtt{t}], \mathbf{c}^\eta : \mathsf{Cfg}_{x^\alpha:\mathtt{t}, y^\beta:\mathtt{B}}(Lx.\nu_\mathtt{t}; Q) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \Rightarrow$ $\mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^\beta:\mathtt{B}}(Q) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$
2.By νL typing rule	$x^\alpha : \mathtt{t} \vdash_\Omega \mathtt{Lx}.\nu_\mathtt{t}; \mathtt{Q} :: y^\beta : \mathtt{B} \hookrightarrow x^{\alpha+1} : \mathtt{C} \vdash_\Lambda Q :: y^\beta : \mathtt{B}$
3.Definition of Λ'	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$.
4.Definition of Λ	for $i \leq n, \Lambda \vDash y_i^\beta \leq z_i^\gamma$ iff $\Omega \vDash y_i^\beta \leq z_i^\gamma$.
5.By assumption	for $i \leq n, \Lambda' \vDash \mathbf{y}_i^\beta \leq \mathbf{z}_i^\gamma$ iff $\Lambda \vDash y_i^\beta \leq z_i^\gamma$
6.Definition of Λ'	for $i \leq n$ and $z^\gamma \neq x^{\alpha+1}, \Lambda' \vDash \mathbf{x}_i^{\alpha+1} \leq \mathbf{z}_i^\gamma$ iff $\Omega' \vDash \mathbf{x}_i^\alpha \leq \mathbf{z}_i^\gamma$.
7.Definition of Λ	for $i \leq n,$ and $z^\gamma \neq x^{\alpha+1}, \Lambda \vDash x_i^{\alpha+1} \leq z_i^\gamma$ iff $\Omega \vDash x_i^\alpha \leq z_i^\gamma$.
8.By assumption	for $i \leq n, \Lambda' \vDash \mathbf{x}_i^{\alpha+1} \leq \mathbf{z}_i^\gamma$ iff $\Lambda \vDash x_i^{\alpha+1} \leq z_i^\gamma$
9.By lines 5 and 8	$x^{\alpha+1} : \mathtt{C} \vdash_\Lambda Q :: y^\beta : \mathtt{B} \; \mathcal{R} \; \mathbf{x}^{\alpha+1} : [x^{\alpha+1} : \mathtt{C}], \mathbf{c}^{\eta+1} : \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{C}, y^\beta:\mathtt{B}}(Q) \vdash_{\Lambda'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$

Cases. Similar to the previous cases.

$\square$

Lemma 7.7. *If*

$$\bar{x}^\alpha : \omega \vdash_\emptyset \mathsf{P} :: (y^\beta : B)$$

is a guarded process, then a derivation built in Lemma 7.3 for

$$\overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_\emptyset \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$$

is valid.

Proof. By assumption there is an (infinitary) guarded typing derivation $\mathcal{D}_1$ for process $\bar{x}^\alpha : \omega \vdash_\emptyset \mathsf{P} :: (y^\beta : \mathtt{B})$. By Lemma 7.6, we build a bisimilar (infinite) derivation for $\star \, \overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_\emptyset \mathbf{y}^\beta : [y^\beta : \mathtt{B}]$ using Cases (1-13) in Lemma 7.3. Consider an infinite path p_2

in $\mathcal{D}_2$ and its bisimilar path p_1 in $\mathcal{D}_1$:

$$\begin{tikzcd}
\cdots \arrow[d, hook, "*"] & \cdots \arrow[d, Rightarrow, "*"] \\
\bar{x}^\alpha : \omega \vdash_\Omega \mathsf{P} :: (y^\beta : \mathtt{B}) \arrow[r, dash, "\mathcal{R}"] \arrow[d, hook, "k"] & \overline{\mathbf{x}^\alpha : [\bar{x}^\alpha : \omega]}, \mathbf{c}^\eta : \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathsf{P}) \vdash_{\Omega'} \mathbf{y}^\beta : [y^\beta : \mathtt{B}] \arrow[d, Rightarrow, "k"] \\
\bar{z}^\gamma : \omega' \vdash_\Lambda \mathsf{Q} :: (w^\delta : \mathtt{D}) \arrow[r, dash, "\mathcal{R}"'] \arrow[d, hook, "*"] & \overline{\mathbf{z}^\gamma : [\bar{z}^\gamma : \omega']}, \mathbf{d}^\theta : \mathsf{Cfg}_{\bar{z}^\gamma:\omega', w^\delta:\mathtt{D}}(\mathsf{Q}) \vdash_{\Lambda'} \mathbf{w}^\delta : [w^\delta : \mathtt{D}] \arrow[d, Rightarrow, "*"] \\
\cdots & \cdots
\end{tikzcd}$$

By definition of $\mathcal{R}$,

- if $\Lambda \vDash \mathsf{snap}(z^\gamma) < \mathsf{snap}(x^\alpha)$, then $\Lambda' \vDash \mathsf{snap}(\mathtt{z}^\gamma) < \mathsf{snap}(\mathtt{x}^\alpha)$, and
- if $\Lambda \vDash \mathsf{snap}(w^\delta) < \mathsf{snap}(y^\beta)$, then $\Lambda' \vDash \mathsf{snap}(\mathtt{w}^\delta) < \mathsf{snap}(\mathtt{y}^\beta)$.

By the above property if path p_1 is guarded, then p_2 is valid. □

Theorem 7.8. *For the judgment $\bar{x}^\alpha : \omega \Vdash \mathcal{C} :: (y^\beta : \mathtt{B})$ where $\mathcal{C}$ is a configuration of guarded processes, there is a valid cut-free proof for $[\bar{x}^\alpha : \omega], \mathsf{Cfg}_{x^\alpha:\mathtt{A}, y^\beta:B}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$ in FIMALL$^\infty_{\mu,\nu}$. Validity of this proof ensures the strong progress property of the configuration $\mathcal{C}$ when executed with a synchronous scheduler.*

Proof. We introduced a derivation for

$$\star\, \overline{[\bar{x}^\alpha : \omega]}, \mathsf{Cfg}_{\bar{x}^\alpha:\omega, y^\beta:\mathtt{B}}(\mathcal{C}) \vdash [y^\beta : \mathtt{B}]$$

in Lemma 7.3, and proved that for a configuration of guarded processes the derivation is valid. It is enough to show that this valid proof ensures the strong progress property of configuration $\mathcal{C}$. Here $\bar{x}^\alpha$ and y^β are external channels of the configuration $\mathcal{C}$.

Consider a run of a configuration of guarded processes $\mathcal{C}$ scheduled by a synchronous scheduler. We run the cut elimination algorithm on the valid proof introduced in Lemma 7.3 such that the steps of the cut elimination algorithm simulates the transition steps of the configuration. Moreover, we show that in the simulated run of the cut elimination algorithm if an external reduction is applied on a predicate of the form $[x^\alpha : A]$ then there is a process in the configuration willing to communicate along the channel $x^\alpha : A$.

Our cut elimination algorithm in Section 4.4 is non-deterministic in the sense that there might be two applicable internal reductions (Prd) available in the Treat function. We proved that no matter what reduction rule we choose, the algorithm will terminate on valid proofs. We can expand this non-determinism to other steps in the Treat function and choose non-deterministically between IdElim, Merge, and Prd. The proof of Lemma 4.7 (termination of the treat function) remains valid since we do not assume an order between these steps in the proof. We can go one step further and execute the termination condition (provided in the **while** clause) on the Treat function non-deterministically: when an external reduction rule is available, the Treat function can either terminate or continue with IdElim, Merge, or Prd. Of course, when IdElim, Merge, or Prd are not applicable, the function has to terminate. The proof of Lemma 4.7 (termination of the treat function) remains valid after this change: the trace of the algorithm remains the same, and we can form a contradiction from the assumption that the trace is infinite.

By the structure of the proof, we know that the first step in the cut elimination algorithm is to apply an external reduction (Flip rule) on $\mathsf{Cfg}_{x^\alpha:\omega, y^\beta:\mathtt{B}}(\mathcal{C})$ to unfold its definition. The tape transforms to

$$\overline{[\bar{x}^\alpha : \omega]}, T \vdash [y^\beta : \mathtt{B}]$$

where T is the definition given based on the pattern of $\mathcal{C}$.

In fact, we can prove that throughout the cut elimination procedure, we repeatedly get a branching tape of the form

$$\overline{[\bar{x}^\alpha : \omega]}, T_1 \vdash [z_1^{\eta_1} : \mathtt{D}_1], [z_1^{\eta_1} : \mathtt{D}_1], T_2 \vdash [z_2^{\eta_2} : \mathtt{D}_2], \cdots [z_m^{\eta_m} : \mathtt{D}_\mathtt{m}], T_{m+1} \vdash [y^\beta : \mathtt{B}]$$

where T_{i+1} is the definition of a predicate $\mathsf{Cfg}_{z_i^{\eta_i}:\mathtt{D}_i, z_{i+1}^{\eta_{i+1}}:\mathtt{D}_{i+1}}(\mathcal{C}_{i+1})$, and it explains the behavior of the computational continuation of $\mathcal{C}$ with regards to channels $\bar{z}_i^{\eta_i}$ and $\bar{z}_{i+1}^{\eta_{i+1}}$. Put $z_0^{\eta_0} = x^\alpha$ and $z_{m+1}^{\eta_{m+1}} = y^\beta$. The channels $z_i^{\eta_i}$ for $1 \leq i \leq m$ are the internal channels of the configuration, and channels x^α and y^β are the external channels.

Moreover, we prove that if an external reduction rule is applied on $[x^\alpha : \omega]$ or $[y^\beta : \mathtt{B}]$, there is a process willing to send or receive a message along x^α or y^β.

This property holds after the very first rule of cut elimination (an external reduction on $\mathsf{Cfg}(\mathcal{C})$ which leads to the tape $\overline{[\bar{x}^\alpha : \omega]}, T \vdash [y^\beta : \mathtt{B}]$. We want to prove that the property explained in the previous paragraph holds as an invariant on the tapes being produced by the cut elimination algorithm if we apply the algorithm in the order enforced by the transition steps of the configuration. The proof describes a weak simulation with the simulation relation relates

$$\overline{[\bar{x}^\alpha : \omega]}, T_1 \vdash [z_1^{\eta_1} : \mathtt{D}_1], [z_1^{\eta_1} : \mathtt{D}_1], T_2 \vdash [z_2^{\eta_2} : \mathtt{D}_2], \cdots [z_m^{\eta_m} : \mathtt{D}_\mathtt{m}], T_{m+1} \vdash [y^\beta : \mathtt{B}]$$

where T_{i+1} is the definition of a predicate $\mathsf{Cfg}_{z_i^{\eta_i}:\mathtt{D}_i, z_{i+1}^{\eta_{i+1}}:\mathtt{D}_{i+1}}(\mathcal{C}_{i+1})$ and the configuration of processes $\mathcal{C}_1 \mid_{z_1} \cdots \mid_{z_m} \mathcal{C}_{m+1}$. We show that for each transition step of a configuration, we can take one or more steps of the cut elimination algorithm such that the resulting branching tape is related to the configuration after taking the step:

Case 1. Forwarding:

There is a judgment on the tape which is of the form:

$$[u^\gamma : \mathtt{F}], u^\gamma = w^\delta \vdash [w^\delta : \mathtt{F}].$$

We apply an external reduction ($= L$) on the antecedent $u^\gamma = w^\delta$. This rule renames channels u^γ and w^δ with their most general unifier z^η. Since u^γ and w^δ are abstract variables, we can assume that the most general unifier is equal to the offering channel w^δ.

We apply identity elimination on the identity judgment $[z^\eta : \mathtt{F}] \vdash [z^\eta : \mathtt{F}]$. The rest of the tape preserves the property of interest, since we only renamed computationally identical channels in it.

Case 2. Spawn: There is a judgment on the tape that is of the form

$$[\bar{u}^\gamma : \omega'], \exists v.\exists \zeta.(\mathsf{Cfg}_{\bar{u}^\gamma:\omega', v^\zeta:\mathtt{E}}(\mathcal{C}_1) \otimes \mathsf{Cfg}_{v^\zeta:\mathtt{E}, w^\delta:\mathtt{F}}(\mathcal{C}_2)) \vdash [w^\delta : \mathtt{F}].$$

We apply two external reduction rules ($\exists$ and $\otimes$) on it to get the judgment

$$[\bar{u}^\gamma : \omega'], (\mathsf{Cfg}_{\bar{u}^\gamma:\omega', v^\zeta:\mathtt{E}}(\mathcal{C}_1), \mathsf{Cfg}_{v^\zeta:\mathtt{E}, w^\delta:\mathtt{F}}(\mathcal{C}_2))) \vdash [w^\delta : \mathtt{F}].$$

By the structure of the proof we built in Lemma 7.3 this judgment is proved using a cut rule. We apply Merge on the cut rule. It replaces them with two judgments connected with a fresh internal channel v^ζ:

$$[\bar{u}^\gamma : \omega'], (\mathsf{Cfg}_{\bar{u}^\gamma:\omega', v^\zeta:\mathtt{E}}(\mathcal{C}_1) \vdash [v^\zeta : \mathtt{E}] \qquad [v^\zeta : \mathtt{E}], \mathsf{Cfg}_{v^\zeta:\mathtt{E}, w^\delta:\mathtt{F}}(\mathcal{C}_2)) \vdash [w^\delta : \mathtt{F}].$$

$\mathcal{C}_1$ is the configuration (or a process) spawned and $\mathcal{C}_2$ is the continuation. With two other external reductions on the Cfg predicates, we unfold the definition of the predicates based on their pattern and get back to a tape satisfying the invariant.

Case 3. Communication along an internal channel: There is a process in the configuration that is willing to send along $w^\gamma : \mathtt{A}$ and one that is willing to receive along $w^\gamma : \mathtt{A}$. By the way that we built the derivation in Lemma 7.3, in the related branching tape there is a judgment in which a left rule is applied on $[w^\gamma : \mathtt{A}]$, and another judgment in which a right rule is applied on $[w^\gamma : \mathtt{A}]$.
We provide the steps of the cut elimination algorithm for a case in which the protocol of communication is an internal choice ($\oplus$) in Figure 7.4. The other cases are similar. It is straightforward to observe that both processes P and Q_k are the computational continuations of the original processes in the configuration: $(Rw.k; \mathsf{P})$ sends the label k along the internal channel w^γ and steps to P. Process **case**$Lw(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell \in L}$ when receiving label k along channel w^γ steps to Q_k. Similar to **Step 1.** with two external reductions on the Cfg predicates, we get back to a tape satisfying the invariant.

We get an extra (green) tape with the single judgment

$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^{\gamma+1}))$$

$$[z^\eta : \mathtt{C}], \mathsf{Cfg}_{z^\eta:\mathtt{C}, w^\gamma:\oplus\{\ell:A_\ell\}_{\ell\in L}}(Rw.k; \mathsf{P}) \vdash [w^\gamma : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}]$$
$$[w^\gamma : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \mathsf{Cfg}_{w^\gamma:\oplus\{\ell:A_\ell\}_{\ell\in L}, v^\delta:\mathtt{D}}(\mathbf{case} Lw(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L}) \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{External reduction}(\mu)$$

$$[\bar{z}^\eta : \omega], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash [w^\gamma : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}]$$
$$[w^\gamma : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \forall u^\eta.\&\{\ell : \mathsf{Msg}(w^\gamma.\ell(u^\eta)) \multimap \mathsf{Cfg}_{u^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{principal reduction}(\mu)$$

$$[\bar{z}^\eta : \omega], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash \oplus\{\ell : (\mathsf{Msg}(w^\gamma.\ell(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_\ell])\}_{\ell\in L}$$
$$\oplus\{\ell : (\mathsf{Msg}(w^\gamma.\ell(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_\ell])\}_{\ell\in L}, \forall u^\eta.\&\{\ell : \mathsf{Msg}(w^\gamma.\ell(u^\eta)) \multimap \mathsf{Cfg}_{u^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{principal reduction}(\oplus)$$

$$[\bar{z}^\eta : \omega], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash (\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k])$$
$$(\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k]), \forall u^\eta\&\{\ell : \mathsf{Msg}(w^\gamma.\ell(u^\eta)) \multimap \mathsf{Cfg}_{u^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{External reduction}(\forall/\&)$$

$$[\bar{z}^\eta : \omega], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash (\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k])$$
$$(\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k]), (\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \multimap \mathsf{Cfg}_{w^{\gamma+1}:\mathtt{A}_k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{External reduction}(\otimes)$$

$$[\bar{z}^\eta : \omega], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})), \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash (\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k])$$
$$(\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \otimes [w^{\gamma+1} : \mathtt{A}_k]), (\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \multimap \mathsf{Cfg}_{w^{\gamma+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{Principal reduction}(\otimes)$$

$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \quad [\bar{z}^\eta : \omega]\mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash [w^{\gamma+1} : \mathtt{A}_k]$$
$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})), [w^{\gamma+1} : \mathtt{A}_k], \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \multimap \mathsf{Cfg}_{w^{\gamma+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}]$$
$$\Downarrow \text{External reduction}(\multimap)$$

$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \quad [\bar{z}^\eta : \omega], \mathsf{Cfg}_{\bar{z}^\eta:\omega, w^{\gamma+1}:\mathtt{A}_k}(\mathsf{P}) \vdash [w^{\gamma+1} : \mathtt{A}_k]$$
$$[w^{\gamma+1} : \mathtt{A}_k], \mathsf{Cfg}_{w^{\gamma+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}] \quad \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^\gamma + 1))(\text{identity elimination})$$
$$\Downarrow \text{External reduction} \times 2$$

$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \quad [\bar{z}^\eta : \omega], T_P \vdash [w^{\gamma+1} : \mathtt{A}_k]$$
$$[w^{\gamma+1} : \mathtt{A}_k], T_{Q_k} \vdash [v^\delta : \mathtt{D}]$$

FIGURE 7.4: A run of the cut elimination algorithm on communicating processes.

We also add a similar extra judgment

$$\mathsf{Msg}(w^\gamma.k(w^{\gamma+1})) \vdash \mathsf{Msg}(w^\gamma.k(w^{\gamma+1}))$$

to the current tape by a $\otimes$ principal reduction. Both of these judgments are closed by an ID-elim.

Case 4. Communication along an external channel: there is a process in the configuration that wants to communicate along an external channel. By the way that we built the derivation in Lemma 7.3, in the related branching tape an external reduction can be

(1) $[x^\alpha : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \mathsf{Cfg}_{x^\alpha:\oplus\{\ell:A_\ell\}_{\ell\in L}, v^\delta:\mathtt{D}}(\mathbf{case}\,Lx(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L}) \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction(μ)

(2) $[x^\alpha : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}], \forall w^\eta.\&\{\ell : \mathsf{Msg}(x^\alpha.\ell(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction(μ)

(3) $\oplus\{\ell : (\mathsf{Msg}(x^\alpha.\ell(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{A}_\ell])\}_{\ell\in L}, \forall w^\eta.\&\{\ell : \mathsf{Msg}(x^\alpha.\ell(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction$(\oplus)$

(4) $\forall k \in L \quad (\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{A}_k]), \forall w^\eta.\&\{\ell : \mathsf{Msg}(x^\alpha.\ell(w^\eta)) \multimap \mathsf{Cfg}_{w^\eta:\mathtt{A}_\ell, v^\delta:\mathtt{D}}(\mathsf{Q}_\ell)\}_{\ell\in L} \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction$(\forall/\&)$

(5) $\forall k \in L \quad (\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \otimes [x^{\alpha+1} : \mathtt{A}_k]), (\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction$(\otimes)$

(6) $\forall k \in L \quad \mathsf{Msg}(x^\alpha.k(x^{\alpha+1})), [x^{\alpha+1} : \mathtt{A}_k], \mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \multimap \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}]$

$\Downarrow$ External reduction$(\multimap)$

(7) $\forall k \in L \quad [x^{\alpha+1} : \mathtt{A}_k], \mathsf{Cfg}_{x^{\alpha+1}:\mathtt{A}k, v^\delta:\mathtt{D}}(\mathsf{Q}_k) \vdash [v^\delta : \mathtt{D}] \qquad \mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \vdash \mathsf{Msg}(x^\alpha.k(x^{\alpha+1}))$(Id elim)

$\Downarrow$ External Reduction(μ)

(8) $\forall k \in L \quad [x^{\alpha+1} : \mathtt{A}_k], T_k \vdash [v^\delta : \mathtt{D}]$

FIGURE 7.5: A run of the cut elimination algorithm when there is a process communicating along an external channel.

applied on predicates $[x^\alpha : \omega]$ or $[y^\beta : \mathtt{B}]$.

In Figure 7.5 we provide the steps of our cut elimination algorithm when there is a process waiting to receive a message along $x^\alpha : \oplus\{\ell{:}A_\ell\}_{\ell\in L}$. In this case a rule can be applied on the predicate $[x^\alpha : \oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}]$. The cases for other types are similar. First observe that by the structure of the proof, this is only the case if in the configuration a process communicates along a left external channel x^α of type $\oplus\{\ell : \mathtt{A}_\ell\}_{\ell\in L}$. Moreover, in Line 4 of Figure 7.5 the algorithm creates multiple branches: the continuation of processes $\mathbf{case}\,Lx(\ell \Rightarrow \mathsf{Q}_\ell)_{\ell\in L}$ depends on the potential label $k \in L$ that it receives along the external channel x^α. The tape we have at the end of each branch corresponds to a potential configuration in the computation. On line (8) we unfold the definition of $\mathsf{Cfg}_{x^{\alpha+1};\mathtt{A}_k, v^\delta:\mathtt{D}}(Q_k)$ predicate to get the invariant we are looking for.

On line (7) we create an extra branch containing a single (green) judgment

$$\mathsf{Msg}(x^\alpha.k(x^{\alpha+1})) \vdash \mathsf{Msg}(x^\alpha.k(x^{\alpha+1})).$$

This tape can be closed by a single ID-elim rule.

Consider the cut-free proof returned by our algorithm. By the property proved above, it is enough to show that an external reduction will be applied on $[x^\alpha : \omega]$ or $[y^\beta : \mathtt{B}]$. We use linearity and validity of the cut-free output derivation. If there are no infinite branches in the proof then by linearity of the calculus we know that a rule is applied on $\overline{[\bar{x}^\alpha : \omega]}$ and $[y^\beta : \mathtt{B}]$. In the infinite case, recall that the predicate Cfg for a recursive process is defined coinductively; no subformula of it in the antecedents can be a part of an infinite μ-trace. Thus an external reduction (flip rule) has to be applied on $[x^\alpha : \omega]$ or $[y^\beta : \mathtt{B}]$ to produce a judgment of the derivation. This completes the proof as it shows that the configuration will eventually communicate with one of its external channels.

We can take one step further, and show that the configuration either terminates or it will eventually communicate with one of its external channels by *receiving a message*. Consider a branch in the cut-free valid proof as described above. If the branch is finite it is straightforward to see that the computation terminates. By a similar reasoning to the previous paragraph, in an infinite branch either $[x^\alpha : \omega]$ has to be a part of an (infinite) μ-trace or $[y^\beta : \mathtt{B}]$ has to be a part of an (infinite) ν-trace. Without loss of generality assume that $[x^\alpha : \omega]$ is a part of a μ-trace. Since the traces are infinite, there has to be an occurrence of a least fixed point type $\mathtt{t}$ in a predicate $[x^\gamma : \mathtt{t}]$ on the branch. As a result, there will be a process in the computation such that it communicates along x^γ, and by the type of x^γ, we know that it will be receiving a fixed point unfolding message. $\square$

Chapter 8

Implementation

We have implemented the guard condition introduced in Chapter 6 on top of an existing interpreter for subsingleton processes in SML; it is available publicly [22]. In this section, we discuss the details of our implementation.

8.1 Syntax

Tables 8.1 and 8.2 summarize the syntax we used for the programs. Each row of Table 8.1 presents an abstract session type and its corresponding presentation in the implementation. Table 8.2 shows the corresponding expression in the implementation for each process term.

The underlying implementation of subsingleton processes supports recursive definitions of session-types but does not differentiate them into positive and negative fixed points as required by our guard condition. To add positive and negative fixed points, we designed their syntax as particular cases of internal and external choices. A positive fixed point $t =_\mu A$ is implemented as a unary internal choice with a specific (reserved) label `mu_t` and continuation `A`. Similarly, a negative fixed point $t =_\nu A$ is implemented as a unary external choice with a label `nu_t` and continuation `A`. This design allows us to introduce positive and negative fixed points to the underlying implementation with minimal change to the syntax, and it perfectly captures the computational semantics of sending and receiving fixed point unfolding messages as defined in Chapter 5. We discuss our design choice for implementing the priorities of positive and negative fixed points in Section 8.3.

A program includes a list of fixed point definitions, process declarations, and process definitions.

```
type t = +{mu_t:A}
type s = &{nu_s:B}
proc f: t|-s
proc f = P
```

Listing 8.1: Program syntax.

Abstract Syntax	Concrete Syntax
1	`1`
$\oplus\{\ell{:}A, \cdots\}$	`+{l:A, ...}`
$\&\{\ell{:}A, \cdots\}$	`&{l:A, ...}`
$t =^i_\mu A$	`t = ⊕{mu_t:A}`
$t =^i_\nu A$	`t = &{nu_t:A}`

Table 8.1: Abstract and Corresponding Concrete Syntax for Types

Abstract Syntax	Concrete Syntax
$\mathbf{close} R x$	`close R`
$\mathbf{wait} L x$	`wait L`
$Rx.k$	`R.k`
$\mathbf{case} L\, x(\ell \Rightarrow \mathsf{P})_{\ell \in L}$	`case L(l => P | ...)`
$\mathbf{case} R\, x(\ell \Rightarrow \mathsf{P})_{\ell \in L}$	`case R(l => P | ...)`
$Lx.k$	`L.k`
$Rx.\mu_{\mathtt{t}}$	`R.mu_t`
$\mathbf{case} L\, x(\mu_{\mathtt{t}} \Rightarrow \mathsf{Q})$	`case L(mu_t => Q)`
$\mathbf{case} R\, x(\nu_{\mathtt{t}} \Rightarrow \mathsf{Q})$	`case R(nu_t => Q)`
$Lx.\nu_{\mathtt{t}}$	`L.nu_t`
$x \leftarrow y$	`<->`
$(x{:}A \leftarrow Q); P$	`Q [A] P`

Table 8.2: Abstract and Corresponding Concrete Syntax for Expressions

For example, the code given in Listing 8.1 defines type variable `t` as the positive fixed point of type `A`, and type variable `s` as the negative fixed point of `B`. Process variable `f` is declared in Line 3 such that it uses a resource of type `t` and offers a resource of type `s`. The last line defines process variable `f` as a process expression `P`.

8.2 Reconstruction of fixed points

Our implementation also supports an implicit syntax where the programmer codes using general (equirecursive) session types. In the implicit syntax, we synthesize the fixed points based on the provided general recursive types: if a general recursive type is defined as a positive type ($\oplus$ or 1), we consider it to be a positive fixed point, and if it is a negative type ($\&$), we consider it a negative fixed point. We then incorporate fixed point unfolding messages in the program from the given communication patterns. Supporting the implicit syntax liberates the programmer from handling fixed point unfolding messages and allows us to check the termination of many programs implemented for other purposes.

We designed two modes in the implementation: *iso* and *equi*, for the programs corresponding to the explicit and implicit use of fixed points, respectively. The programmer indicates the mode for executing their program using a flag in the program file. If the flag is set to *iso*, the programmer must define recursive types as positive and negative fixed points, and we do not need to perform any transformations on the program. If the flag is set to *equi*, the programmer writes the program considering only general recursive types. We then transform this given program with a similar method in prior work [23, 24]: insert sending a fixed point unfolding message as soon as possible (eagerly) and receiving a fixed point unfolding message just before the communication on that channel (lazily).

8.3 Termination checking

Once the program is parsed and the underlying implementation of subsingleton logic extracts its abstract syntax tree, we perform a termination check using our guard condition. Recall that our guard condition works based on priorities defined over type variables and the order between process variables. The current implementation collects constraints and uses them to construct a suitable priority ordering over type variables and a $\subset$ ordering over process variables if they exist and rejects the program otherwise.

Our implementation provides a detailed error message when a program does not satisfy the guard condition. Our experience suggests error messages could be improved further by requiring the programmer to supply priorities of type definitions, but we have left this for future work. Performing inference allowed us to check a variety of preexisting examples without change.

8.4 Examples

We have coded all programming examples (terminating or not) in this thesis in the implementation. For example, the following listing presents the code of `PingPong` process from Chapter 6. Line 2 (#test error) refers to the fact that our algorithm should not (and does not) accept this program since `PingPong` does not satisfy strong progress.

```
 1  #options --terminate=iso
 2  #test error
 3
 4  type ack=+{mu_ack: +{ack:astream}}
 5  type astream=&{nu_astream: &{head:ack, tail:astream}}
 6  type nat=+{mu_nat:+{z:1, s:nat}}
 7
 8
 9  proc Ping_Pong: nat |- nat
10  proc Ping_Pong= Ping [astream] Pong
11  proc Ping: nat |- astream
12  proc Ping= caseR(nu_astream => caseR (head=> R.mu_ack;R.ack;Ping
13                                  | tail=> Ping ))
14  proc Pong: astream |- nat
15  proc Pong= L.nu_astream; L.head; caseL(mu_ack =>
16                                   caseL (ack=> R.mu_nat;R.s;Pong ))
```

We used several test cases to examine our implementation. The following is an example of a program with an *equi* flag: it implements a constant function that returns the binary representation of number six, a copy process over binary numbers, and a process that computes the successor of a binary number.

```
#options --terminate=equi
#test success

type bits = +{b0 : bits, b1 : bits, $ : 1}

proc six : bits
proc six = R.b0 ; R.b1 ; R.b1 ; R.$ ; closeR

proc copy : bits |- bits
proc copy = caseL ( b0 => R.b0 ; copy
                  | b1 => R.b1 ; copy
                  | $ => R.$ ; waitL ; closeR )

proc plus1 : bits |- bits
proc plus1 = caseL ( b0 => R.b1 ; <->
                   | b1 => R.b0 ; plus1
                   | $ => R.$ ; waitL ; closeR )
```

Our experience with a range of programming examples shows that our local validity condition is surprisingly effective. In particular, we encoded Turing machines and observed that our implementation guarantees termination of a subclass of Turing machines corresponding to linear primitive recursive functions. The code of all examples and test cases is available publicly [22].

Chapter 9

Conclusion

This thesis establishes a logical foundation for recursive concurrent session types using infinitary linear logics with fixed points. To develop this logical foundation, we appeal to two well-known paradigms that relate programs to logical systems:

- We form a *Curry-Howard correspondence* between recursive processes and circular proofs as introduced by Fortier and Santocanale [36]. We provide an effectively decidable local guard criterion to recognize mutually recursive processes with a strong progress property. We show that our guard criterion imposes a stricter requirement than Fortier and Santocanale's validity condition, but is local and compositional and therefore more suitable as the basis for a programming language.
- We embed session-typed processes and their asynchronous semantics in an infinitary first order linear logic with fixed points using a *processes-as-formulas* interpretation. We then define the strong progress property as a predicate with nested least and greatest fixed points. We prove strong progress of guarded programs by providing a syntactic proof for this predicate in our calculus and verifying that this proof ensures strong progress of the underlying program when executed with a synchronous scheduler.

Next, we discuss potential lines of future work for exploring the intersection between non-wellfounded proof theory and recursive session types.

9.1 Strong progress as a logical relation

Logical relations is a proof method based on forming relations indexed by types. The relations are called *logical* since they are defined by induction on the structure of their underlying type. The first principal application of this method was presented by Tait [93], Girard [42], Plotkin [76], and Statman [91] for proving strong normalization of simply-typed λ-calculus.

The strong progress property for session typed processes is of the same nature as strong normalization in typed λ-calculus. In the setting of non-recursive session types, this property is reduced to termination of the computation and is proved using logical relations [32, 72]. Similar to using logical relations to prove strong normalization for simply typed λ-calculus, the proof of termination for processes relies on an induction over the type structure and no longer applies after adding recursive types. In response, step-indexed logical relations [3–5] have been developed to prove properties of typed calculi with recursive types. Step-indexed logical relations are indexed by both types and the number of available future steps. Later Dreyer et al. [34] provided a more elegant syntactic definition of logical relations without referring to steps to prove properties of system F with isorecursive least fixed points. Their definition is in the language of a second-order calculus called LSLR with a *future* modality. They encoded a logical relation as a well-founded recursive second-order relation.

However, neither strong normalization nor strong progress can be formalized as a logical relation indexed by the steps of computation as they are both associated with termination. The strong progress predicate presented in Chapter 7 is closely related to the concept of logical relation since it is also defined based on the structure of its underlying type. Moreover, we observed that in the presence of the greatest fixed points, the definition has a mutual inductive and coinductive nature. One main avenue for future work is to convert our mixed logical relation for strong progress as a logical relation indexed by (i) the number of unfoldings required for termination for inductive types and (ii) the number of observations allowed before termination for coinductive types. We have verified this method for a preliminary setting in which the signature consists of a least fixed point nested inside a greatest one. We will compare the results with prior work that combines inductive with coinductive reasoning for termination [47].

9.2 A more general guard condition for the subsingleton fragment

In Chapter 6 we showed that the main shortcoming of our guard condition arises when, intuitively, we need to know that a program's output is "smaller" than its input. Our goal is to capture more programs with this property as long as the algorithm is still effective, compositional, and predictable by the programmer. We need to generalize the guard condition by introducing a way to capture the relation between input and output size. We believe this generalization would be more feasible when our mixed logical relation have been fully developed.

Studying this generalization also allows us to compare our results with the sized-type approach introduced by Abel and Pientka [2]. In this approach, Abel and Pientka integrate induction and coinduction by pattern and copattern matching and explicit well-founded induction on ordinals [2], following a number of earlier representations of induction and coinduction in type theory [1]. The connection to this type-theoretic approach is an interesting item for future research. The first step in this general direction was taken by Sprenger and Dam [90] who justify cyclic inductive proofs using inflationary iteration and the work by Somayyajula and Pfenning [89] for shared memory concurrency.

9.3 Recursive binary session types in linear logic

One obvious line of future work is to generalize the results in the subsingleton fragment to recursive processes defined based on intuitionistic multiplicative additive linear logic [74]. In contrast to the subsingleton fragment, a process in the more general linear setting may use more than one service on its left. To develop a local guard condition on linear processes, we need to track the relation between all services that the process uses on the left and the service that it provides on the right. Moreover, we need to deal with channel delegation that appears in the linear setting as the semantic of multiplicative conjunction and linear implication.

9.4 Linear logic with adjoint modalities

Binary session types have also been studied in the setting of an adjoint logic [74, 78]. In this setting, formulas are not restricted to the linear contexts anymore. They can shift their modes using *upgrade* and *downgrade* adjoint modalities ($\uparrow, \downarrow$) to move back and forth between structural, affine, and linear contexts. It would be an interesting project to extend the results presented in this thesis to the logic with adjoint modalities. This generalization can be two-fold: 1) introducing $\downarrow$ and $\uparrow$ mode shift modalities into our infinitary first-order calculus. So far, we have a promising preliminary result for a restricted form of shift modality in $FIMALL^{\infty}_{\mu,\nu}$, in which we can only move formulas from the linear context to the structural one. 2) Considering strong progress property for a guarded subset of recursive session-typed processes defined based on the adjoint logic.

Bibliography

[1] Andreas Abel and Brigitte Pientka. 2013. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM, 185–196.

[2] Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *J. Funct. Program.* 26 (2016), e2.

[3] Amal Jamil Ahmed. 2004. *Semantics of types for mutable state*. Ph.D. Dissertation. Princeton University.

[4] Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 3924. Springer, 69–83.

[5] Andrew W Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 657–683.

[6] André Arnold and Damian Niwinski. 2001. *Rudiments of calculus*. Elsevier.

[7] David Baelde, Amina Doumane, and Alexis Saurin. 2016. Infinitary Proof Theory: the Multiplicative Additive Case. In *25th Annual Conference on Computer Science Logic (CSL 2016)*. LIPIcs 62, Marseille, France, 42:1–42:17.

[8] David Baelde and Dale Miller. 2007. Least and Greatest Fixed Points in Linear Logic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings (Lecture Notes in Computer Science)*, Vol. 4790. Springer, 92–106.

[9] Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 37:1–37:29.

[10] Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *28th Annual Symposium on Logic in Computer Science (LICS 2013)*. IEEE Computer Society, New Orleans, LA, USA, 213–222.

[11] Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

[12] James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings (Lecture Notes in Computer Science)*, Vol. 3702. Springer, 78–92.

[13] James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *J. Log. Comput.* 21, 6 (2011), 1177–1216.

[14] Paola Bruscoli. 2002. A Purely Logical Account of Sequentiality in Proof Search. In *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings (Lecture Notes in Computer Science)*, Vol. 2401. Springer, 302–316.

[15] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6269. Springer, 222–236.

[16] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. Special Issue on Behavioural Types.

[17] Iliano Cervesato and Andre Scedrov. 2009. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information and Computation* 207, 10 (Oct. 2009), 1044–1077.

[18] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. 2003. *A Judgmental Analysis of Linear Logic.* Technical Report CMU-CS-03-131R. Carnegie Mellon University, Department of Computer Science.

[19] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68.

[20] Haskell B Curry. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America* 20, 11 (1934), 584.

[21] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *34th IEEE Computer Security Foundations Symposium (CSF).* To appear.

[22] Ankush Das, Farzaneh Derakhshan, and Frank Pfenning. 2019. SubSingleton. `https://bitbucket.org/fpfenning/subsingleton` An implementation of subsingleton logic with ergometric and temporal types.

[23] Ankush Das and Frank Pfenning. 2020. Rast: Resource-Aware Session Types with Arithmetic Refinements (System Description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs)*, Vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:17.

[24] Ankush Das and Frank Pfenning. 2020. Verified Linear Session-Typed Concurrent Programming. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020.* ACM, 7:1–7:15.

[25] Anupam Das and Damien Pous. 2018. Non-Wellfounded Proof Theory for (Kleene+ Action) (Algebras+ Lattices). In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018).* LIPIcs 119.

[26] Willem P. de Roever. 1977. On Backtracking and Greatest Fixpoints. In *Automata, Languages and Programming, Fourth Colloquium, University of Turku, Finland, July 18-22, 1977, Proceedings (Lecture Notes in Computer Science)*, Arto Salomaa and Magnus Steinby (Eds.), Vol. 52. Springer, 412–429.

[27] Farzaneh Derakhshan and Frank Pfenning. 2019. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *arXiv preprint arXiv:1908.01909* (2019).

[28] Farzaneh Derakhshan and Frank Pfenning. 2021. Strong Progress for Session-Typed Processes in a Linear Metalogic with Circular Proofs. arXiv:cs.LO/2001.05132

[29] Henry DeYoung. 2020. *Session-Types Ordered Logical Specifications.* Ph.D. Dissertation. Carnegie Mellon University. Available as Technical Report CMU-CS-20-133.

[30] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Proceedings of the 21st Annual Conference on Computer Science Logic (CSL 2012).* LIPIcs 16, Fontainebleau, France, 228–242.

[31] Henry DeYoung and Frank Pfenning. 2016. Substructural Proofs as Automata. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 10017. 3–22.

[32] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. 2020. Semi-Axiomatic Sequent Calculus. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs)*, Vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:22.

[33] Amina Doumane. 2017. *On the Infinitary Proof Theory of Logics with Fixed Points.* Ph.D. Dissertation. Paris Diderot University, France.

[34] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 71–80.

[35] Lars-Henrik Eriksson. 1991. A finitary version of the calculus of partial inductive definitions. In *International Workshop on Extensions of Logic Programming*. Springer, 89–134.

[36] Jérôme Fortier and Luigi Santocanale. 2013. Cuts for Circular Proofs: Semantics and Cut-Elimination. In *22nd Annual Conference on Computer Science Logic (CSL 2013)*. LIPIcs 23, Torino, Italy, 248–262.

[37] Curtis Franks. 2010. Cut as consequence. *History and Philosophy of Logic* 31, 4 (2010), 349–379.

[38] Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (Jan. 2010), 19–50.

[39] Gerhard Gentzen. 1934-5. Untersuchungen über das logische Schließen I, II. *Mathematische Zeitschrift* 39, 1 (1934-5), 176–210, 405–431.

[40] Gerhard Gentzen. 1938. Neue Fassung des Widerspruchsfreiheitsbeweises für die reine Zahlentheorie. Forschung zur Logik und zur Grundlegung der exakten Wissenschaften. *Neue Folge 4, S. Hirzel:19–44* (1938).

[41] Jean-Yves Girard and Yves Lafont. 1987. Linear Logic and Lazy Computation. In *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP) (Lecture Notes in Computer Science)*, Vol. 250. Springer, 52–66.

[42] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Éditeur inconnu.

[43] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.

[44] Jean-Yves Girard. 1992. A fixpoint theorem in linear logic. *Linear Logic Mailing List, linear@ cs. stanford. edu* 5 (1992).

[45] Hans Brugge Grathwohl. 2016. *Guarded Recursive Type Theory.* Ph.D. Dissertation. Department of Computer Science, Aarhus University, Denmark.

[46] Dennis Griffith. 2016. *Polarized Substructural Session Types.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[47] Robert Harper. 2021. Termination for Natural and Unnatural Numbers. (2021). http://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/natco.pd

[48] Claudio Hermida and Bart Jacobs. 1998. Structural induction and coinduction in a fibrational setting. *Information and Computation* 145, 2 (1998), 107–152.

[49] David Hilbert and Paul Bernays. 1934. *Grundlagen der Mathematik, volume I.* Springer.

[50] David Hilbert and Paul Bernays. 1939. *Grundlagen der Mathematik, volume II.* Springer.

[51] Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR'93).* Springer LNCS 715, 509–523.

[52] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming Languages and Systems (ESOP 1998).* Springer LNCS 1381, 122–138.

[53] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* ACM, 273–284.

[54] Ross Horne. 2020. Session Subtyping and Multiparty Compatibility Using Circular Sequents. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs)*, Vol. 171. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:22.

[55] Ross Horne and Alwen Tiu. 2019. Constructing weak simulations from linear implications for processes with private names. *Mathematical Structures in Computer Science* 29, 8 (2019), 1275–1308.

[56] WA Howard. 1969. To HB Curry: The formulae-as-types notion of construction. *Essays on Combinatory Logic, Lambda Calculus, and Formalism* (1969).

[57] André Joyal. 1996. Free Lattices, Communication and Money Games. In *Logic and Scientific Methods: Volume One of the Tenth International Congress of Logic, Methodology and Philosophy of Science, Florence, August 1995*, Vol. 259. Springer Science & Business Media, 29.

[58] Reinhard Kahle and Michael Rathjen. 2020. *The Legacy of Kurt Schütte.* Springer.

[59] Dexter Kozen and Alexandra Silva. 2017. Practical coinduction. *Mathematical Structures in Computer Science* 27, 7 (2017), 1132–1152.

[60] Sam Lindley and J. Garrett Morris. 2016. Talking bananas: structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* ACM, 434–447.

[61] Paul Lorenzen. 1951. Algebraische und logistische Untersuchungen über freie Verbände. *Journal of Symbolic Logic* (1951), 81–106.

[62] Per Martin-Löf. 1971. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Studies in Logic and the Foundations of Mathematics.* Vol. 63. Elsevier, 179–216.

[63] Raymond McDowell and Dale Miller. 1997. A Logic for Reasoning with Higher-Order Abstract Syntax. In *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, Glynn Winskel (Ed.). IEEE Computer Society Press, Warsaw, Poland, 434–445.

[64] Raymond McDowell and Dale Miller. 2000. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science* 232, 1-2 (2000), 91–119.

[65] Nax P. Mendler. 1987. Recursive types and type constraints in second-order lambda calculus. In *LICS*, Vol. 87. 30–36.

[66] Nax P. Mendler. 1991. Inductive Types and Type Constraints in the Second-Order lambda Calculus. *Ann. Pure Appl. Log.* 51, 1-2 (1991), 159–172. https://doi.org/10.1016/0168-0072(91)90069-X

[67] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1, 4 (1991), 497–536.

[68] Dale Miller. 1992. The π-calculus as a theory in linear logic: Preliminary results. In *International Workshop on Extensions of Logic Programming*. Springer, 242–264.

[69] Robin Milner. 1990. Functions as processes. In *International Colloquium on Automata, Languages, and Programming*. Springer, 167–180.

[70] Robin Milner and Mads Tofte. 1991. Co-Induction in Relational Semantics. *Theor. Comput. Sci.* 87, 1 (1991), 209–220.

[71] Alberto Momigliano and Alwen Tiu. 2003. Induction and co-induction in sequent calculus. In *International Workshop on Types for Proofs and Programs*. Springer, 293–308.

[72] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 539–558.

[73] Frank Pfenning. 2016. Substructural Logics. (Dec. 2016). http://www.cs.cmu.edu/~fp/courses/15816-f16/lectures/substructural-logics.pdf Lecture notes for course given at Carnegie Mellon University, Fall 2016.

[74] Frank Pfenning and Dennis Griffith. 2015. Polarized substructural session types. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 3–22.

[75] Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

[76] Gordon D Plotkin. 1973. Lambda-definability and logical relations, 1973. *Memorandum SAI-RM* 4 (1973).

[77] Dag Prawitz and Natural Deduction. 1965. A Proof-Theoretical Study". *Almqvist˜ Wiksell, Stockholm* (1965).

[78] Klaas Pruiksma and Frank Pfenning. 2019. A Message-Passing Interpretation of Adjoint Logic. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019 (EPTCS)*, Vol. 291. 60–79.

[79] Grigore Rosu. 2017. Matching Logic. *Logical Methods in Computer Science* 13, 4 (2017).

[80] Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 4 (2009), 1–41.

[81] Luigi Santocanale. 2002. A Calculus of Circular Proofs and Its Categorical Semantics. In *5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2002).* Springer LNCS 2303, Grenoble, France, 357–371.

[82] Luigi Santocanale. 2002. From parity games to circular proofs. *Electronic Notes in Theoretical Computer Science* 65, 1 (2002), 305–316.

[83] Luigi Santocanale. 2002. μ-Bicomplete Categories and Parity Games. *Informatique Théorique et Applications* 36, 2 (2002), 195–227.

[84] Peter Schroeder-Heister. 1993. Rules of Definitional Reflection. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993.* IEEE Computer Society, 222–232.

[85] Peter Schroeder-Heister. 2006. Validity concepts in proof-theoretic semantics. *Synthese* 148, 3 (2006), 525–571.

[86] Helmut Seidl. 1996. Fast and simple nested fixpoints. *Inform. Process. Lett.* 59, 6 (1996), 303–308.

[87] Wilfried Sieg. 2012. In the shadow of incompleteness: Hilbert and Gentzen. In *Epistemology versus Ontology.* Springer, 87–127.

[88] Wilfried Sieg. 2013. *Hilbert's programs and beyond.* Oxford University Press.

[89] Siva Somayyajula and Frank Pfenning. 2021. Circular Proofs as Processes: Type-Based Termination via Arithmetic Refinements. *arXiv preprint arXiv:2105.06024* (2021).

[90] Christoph Sprenger and Mads Dam. 2003. On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Vol. 2620. Springer, 425–440.

[91] Richard Statman. 1985. Logical relations and the typed λ-calculus. *Information and control* 65, 2-3 (1985), 85–97.

[92] Colin Stirling. 2014. A Tableau Proof System with Names for Modal Mu-calculus. In *HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday*. EPiC Series in Computing, Vol. 42. EasyChair, 306–318.

[93] William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212.

[94] William W Tait. 1968. Normal derivability in classical logic. In *The Syntax and Semantics of Infinitary Languages*. Springer, 204–236.

[95] Alfred Tarski et al. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

[96] Alwen Tiu and Dale Miller. 2010. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Transactions on Computational Logic (TOCL)* 11, 2 (2010), 1–35.

[97] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Vol. 7792. Springer, 350–369.

[98] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*. ACM Press, Copenhagen, Denmark, 273–286.