Generic Process Shape Types and the Poly⋆ System

Jan Jakubův

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematical and Computer Sciences

September 2010

Abstract

Shape types are a general concept of process types which allows verification of various properties of processes from various calculi. The key property is that shape types "look like processes", that is, they resemble process structure and content. POLY✶, originally designed by Makholm and Wells, is a type system scheme which can be instantiated to a shape type system for many calculi. Every POLY✶ instantiation has desirable properties including subject reduction, polymorphism, the existence of principal typings, and a type inference algorithm.

In the first part of this thesis, we fix and describe inconsistencies found in the original POLY✶ system, we extend the system to support name restriction, and we provide a detailed proof of the correctness of the system.

In the second part, we present a description of the type inference algorithm which we use to constructively prove the existence of principal typings.

In the third part, we present various applications of shape types which demonstrate their advantages. Furthermore we prove that shape types can provide the same expressive power as and also strictly superior expressive power than predicates of three quite dissimilar analysis systems from the literature, namely, (1) an implicitly typed π-calculus, (2) an explicitly typed Mobile Ambients, (3) and a flow analysis system for BioAmbients.

Acknowledgments

First of all, I would like to thank my first supervisor Dr. Joe Wells for his patience and encouraging guidance throughout my PhD studies, and for his well-directed advices and comments which have influenced my work and extended my abilities. I am grateful to my second supervisor Prof. Fairouz Kamareddine for her kind support and for creating a familial working environment. Thanks to Vincent Rahli, Robert Lamar, Sergueï Lenglet, and all other members of the ULTRA group for their kindness, help, and support both in and outside of the office. I am thankful to Dr. Pierluigi Frisco and to Dr. Andrew D. Gordon for being my examiners.

I would also like to thank all the staff of the School of Mathematical and Computer Sciences at Heriot-Watt University in Edinburgh for creating a stimulating, supportive, well-equipped, well-maintained, clean, and secure research environment. I would like to express my gratitude to the supervisor of my master thesis Prof. Petr Štěpánek from Charles University in Prague for introducing me to the inspiring world of type systems and formal logics. I am thankful to Dr. Josef Urban for introducing me to the members of the ULTRA group. It would not be possible to write this thesis without a financial support of EPSRC grant EP/C013573/1 for which I am greatly grateful.

Last but not least, I would like to express my gratitude to my parents and to all members of my family for their patience, kindness, and for the opportunity to undertake my studies. Finally, let me express my gratitude to all good people who have led science and society to this point where I was able to write this thesis.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation: Why Formal Models?

This thesis deals with formal models of concurrent systems and thus the question of their purpose and usefulness should be addressed in the first place. A concurrent system is any system where several units engage in activity at the same time. The units can interact with each other and thus mutually affect their behavior. Many examples of artificial concurrent systems are found in computer science, for example, large computer networks containing various number of interacting computers, or several processes running simultaneously in a single computer. Formal models were originally developed for these computer systems but nowadays the same modeling techniques are also used to model real-world concurrent systems like complex molecular and biological systems or work flow in business management.

Formal models of concurrent systems use precise mathematical methods to capture overall or specific behavior of a selected system. Different formal models are designed to achieve different aims. The common goals of formal modeling include the following.

To study the behavior of a concurrent system. We create a formal model and we compare its behavior with the behavior of the real system. This comparison can provide us a valuable insight into the nature of the studied system and it often allows us to improve the formal model. The newly improved model can be again compared with the real system. Every iteration of this cycle gives us a more refined formal model as well as it increases our understanding of the real system.

To develop/reason about a concurrent system. Formal models provides an invaluable help with development of a new artificial concurrent systems. They can help us to reveal mistakes in the system design before the system is implemented or used in practice. In this way formal models help us to save expensive resources. For example, if we had a faithful model of complex bio-

logical systems, we would be able to test new drugs on this model instead of on living animals. Unfortunately, current formal models of biological systems do not yet reach the level of reliability required to achieve this aim.

A reliable model of a concurrent system can be used to reason about and to prove various system properties which are of interest, including critical properties like security and correctness of artificially constructed concurrent systems. Formal reasoning about a concurrent system can help us to develop the most efficient ways to use and to interact with the system.

To study the concepts of interaction and communication. The aim here is to capture and describe the basic principles and mechanisms on which interaction and communication are based. A better understanding of these basic principles allows us to construct more accurate formal models and thus it improves the results obtained from applications of formal models. Moreover, a better understanding of these principles improves our understanding of the world and of ourselves, and thus it moves us one step towards the ultimate goal of science.

1.2 Structure of this Chapter

The rest of this introductory chapter is structured as follows. Section 1.3 introduces a basic terminology of process calculi which are one of the formalisms used to model concurrent systems. Section 1.4 provides a very brief historical overview of process calculi. Section 1.5 introduces basic ideas of type and analysis systems for process calculi which are used to verify and reason about various properties of concurrent systems. Section 1.6 provides a brief historical overview of the generic analysis system Poly✶ which is the main topic of this thesis. Section 1.7 provides a short overview of Poly✶. Finally Section 1.8 summarizes the thesis contributions.

1.3 Basics of Process Calculi

Various formal models of concurrent systems are found in the literature. In this thesis we concentrate solely at process calculi which constitute one of the possible approaches to model concurrent systems. Different process calculi are designed to model different systems but they share the basic idea. Any process calculus C defines the set of *processes* which are used to represent interacting units of the system. One process can represent either a single unit or a more complex system consisting of several units. Let B range over processes of calculus C.

In order to describe the behavior of a modeled system, calculus C defines a binary *rewriting relation* on processes, written $B_0 \rightarrow B_1$, which is read as "B_0 rewrites

to B_1". The statement $B_0 \to B_1$ means that the system described by process B_0 evolves in one step to the system described by B_1, that is, that B_1 is an immediate successor of B_0. A single process can have more than one possible successor and thus the rewriting relation can describe a *nondeterministic* behavior. As opposed to rewriting systems used to describe the behavior of functions, like the λ-calculus, consecutive applications of the rewriting relation which start with the same process B_0 do not necessarily need to converge to the single final state. Sometimes it is even desirable that some rewritings do not converge at all because the modeled system exhibits this behavior. In some process calculi the rewriting relation can be additionally labeled with various labels but in this thesis we work only with *unlabeled* rewriting relations.

Many process calculi share common operators which are used to construct processes. Commonly, "0" is used to denote a finished or inactive process, "$B_0 \mid B_1$" is used to denote two processes B_0 and B_1 running in parallel, and "$N.B$" is used to denote a process which executes action N and then continues as process B. Furthermore, "$!B$" is used to describe a process which behaves like infinitely many copies of B running in parallel, that is, like "$B \mid B \mid \cdots$". Processes are usually constructed from atomic units called *names*. A common operation is that of *name restriction*, written "$\nu n.B$", which makes the name n in B to be different from any other name outside B (even though also called n).

Although there is no general consensus on which calculus should be considered a base calculus to model concurrency, the π-calculus [MPW92b, Mil99] and the Mobile Ambients calculus [CG98] are best-known nowadays. Many extensions, variations, and combinations of these two calculi were introduced to model and to reason about various properties of concurrent systems.

1.3.1 The π-calculus

The π-calculus [MPW92b, Mil99] models interaction as a channel-based communication where atomic entities called *names* are send and received over named *channels*. Channels are identified by names which means that channel identifiers can be transmitted during communication. There are two kinds of executable communication actions: (1) sending a name n over a channel c (written "$c\langle n \rangle$"), and (2) receiving a name over a channel c and saving it in (that is, substituting it for) x (written "$c(x)$"). Processes in the π-calculus are constructed from communication actions by parallel ("|") and sequential (".") compositions. For example, the process "c<a>.0" executes the action "c<a>" and ends while the process "c(x).x<d>.0" executes the action "c(x)" and then the action "x<d>". A communication is actually executed when a sending and receiving process appear in parallel like in "c<a>.0|c(x).x<d>.0". In this case name a is substituted for x and the process evolves (rewrites) to its next

state as follows.

$$\mathsf{c\langle a\rangle.0 \mid c(x).x\langle d\rangle.0} \quad \rightarrow \quad \mathsf{0 \mid a\langle d\rangle.0}$$

More details on the π-calculus can be found in Chapter 14.

1.3.2 Mobile Ambients

The Mobile Ambients calculus [CG98] of Cardelli and Gordon places processes into separated abstract *locations* called *ambients*. An ambient is a named bounded place and a process in one ambient can not directly interact with a process in another ambient. A process B running inside an ambient n is written "$n\,\mathsf{[}B\mathsf{]}$". An ambient can contain processes and other ambients, like in "$\mathsf{a[}B_0 \mathsf{\mid b[}B_1\mathsf{]]}$", and thus ambients form a tree-like hierarchy. A process can execute instructions called *capabilities* whose execution changes the ambient hierarchy. Processes in the same ambient can also communicate by sending names and sequences of capabilities. Process are constructed from capabilities and communication actions by parallel ("$\mathsf{|}$") and sequential (".") compositions.

There are three kinds of capabilities in Mobile Ambients: in, out, and open. The capability "$\mathsf{in}\ n$" instructs the ambient which contains (the process that executes) the capability to enter the sibling ambient n. The capability "$\mathsf{in}\ n$" can actually be executed only when there is some sibling ambient n, like in "$\mathsf{a[in\ b.in\ c.0] \mid b[0]}$". This process evolves (rewrites) as follows.

$$\mathsf{a[in\ b.in\ c.0] \mid b[0]} \quad \rightarrow \quad \mathsf{b[0 \mid a[in\ c.0]]}$$

Similarly the capability "$\mathsf{out}\ n$" instructs the ambient to move out of its parent n. The capability "$\mathsf{open}\ n$" instructs the ambient to dissolve the ambient boundary of its child ambient n. Their semantics is described by the following rewriting axioms.

$$\begin{array}{lcl} m\mathsf{[}n\mathsf{[out}\ m.B_0 \mid B_1\mathsf{]} \mid B_2\mathsf{]} & \rightarrow & n\mathsf{[}B_0 \mid B_1\mathsf{]} \mid m\mathsf{[}B_2\mathsf{]} \\ \mathsf{open}\ n.B_0 \mid n\mathsf{[}B_1\mathsf{]} & \rightarrow & B_0 \mid B_1 \end{array}$$

Mobile Ambients also allow communication over (unnamed) channels. This is described by the following axiom.

$$\mathsf{(}n\mathsf{).}B_0 \mid \mathsf{\langle}N\mathsf{\rangle.}B_1 \quad \rightarrow \quad B_0\{n \mapsto N\} \mid B_1$$

In the above rule N is a metavariable ranging over sequences of capabilities and names, and $B_0\{n \mapsto N\}$ denotes the application of the substitution $\{n \mapsto N\}$ to B_0. More details on Mobile Ambients can be found in Chapter 16.

1.4 A Very Brief History of Process Calculi

The history of formal descriptions of concurrent systems can be traced back to 1960s. Several formalisms intended to capture the concept of a computable function, like Turing Machines and the λ-calculus, were already proposed in the first half of the 20th century. A need for more subtle definition of computation arose with the expansion of Computer Science.

It is common for computations executed by computers to interact with the environment, for example, with users or with other computers. Thus the result of a computation can depend on the state of the environment and does not need to be uniquely determined by input parameters. Behavior of these computations can not be straightforwardly described by functions. Formalisms were developed to model similar concurrent systems where several interactive units engage in activity at the same time.

As the first work that mentions concurrency we can point out Petri nets, for the first time published in the PhD thesis [Pet62] of Petri in 1962. Petri nets model a concurrent system by a (bipartite) graph with two kind of nodes which represent states and events of the system. Petri nets are also used nowadays but they use a different approach to concurrency than the one used in process calculi.

Another important researcher studying behavior of concurrent systems was Bekič, who worked for IBM and was well-known for his work on semantics of programming languages in the 60s and 70s. In his paper [Bek84] from 1971 he addresses parallel execution of processes. He was the first one who used an operator to denote a parallel composition of processes, in particular to denote what he called a quasi-parallel execution of processes. This parallel composition operator plays a central role in every modern process calculus.

The first process calculi are due to the independent work of Milner and Hoare. The work of Milner between the year 1973 and 1980 culminated in the Calculus of Communicating Systems (CCS) described in his book published in 1980 [Mil80]. CCS already defines operators for sequential, parallel and alternative composition which are milestones of process calculi. In 1978 Hoare published the paper that describes the language Communicating Sequential Processes (CSP) [Hoa78]. CSP provides a way to describe synchronous communication and also has been practically applied in industry to formal verification of the concurrent aspect of several systems. The subsequent development of CSP was influenced by the development of CCS and *vice versa*. Both the theory of CCS and CSP are still the subject of active research. While process calculi like CCS and CSP usually use transition systems to give a semantics to programs, there is also a different approach that uses algebraic equations to describe the behavior of the calculus. These approaches are usually called process algebras. Among them we can mention probably the first one: Algebra

of Communicating Processes (ACP) [BK84] of Bergstra and Klop. Furthermore, there exist also algebraic approaches to CCS and CSP. Algebraic approaches are used to prove various properties of CCS and CSP, and a huge amount of formal proofs elaborated in details can be found in the literature. This implies a great level of reliability and is one of reasons while CCS and CSP are still used, even though their successors are in some sense either more expressive, simpler or more suitable for different purposes.

From the 1960s to now a large variety of process calculi have been developed. There are several different aspects that they are trying to address. Among these aspects are data treatment, time treatment, probability (a stochastic information treatment), and mobility. Probably the most popular modern process calculi concerning mobility are the π-calculus, which is the successor of CCS, and Mobile Ambients. The π-calculus and Mobile Ambients have attracted many researchers and have led to a spreading of the process calculi approach and its applications. Now, for both the π-calculus and Mobile Ambients many extensions, variations, and combinations exist. A more detailed historical overview can be found in a paper of Baeten [Bae05].

1.5 Basics of Type and Analysis Systems

Type and static *analysis systems* formalize certain kinds of reasoning about properties of processes. For any process calculus C, one or more type/static analysis systems can be designed. An analysis system S_C for calculus C is usually designed to formally reason about and to verify a specific property of processes from calculus C. Different analysis system can be designed to reason about different process properties and thus to reason about different properties of the modeled concurrent system.

A typical type or static analysis system S_C for process calculus C works as follows. Firstly, it defines the set of *predicates*. Let ρ range over these predicates. Predicates in many systems consist of several parts, typically they contain all non-process entities which form typing judgments. For example, judgments of a type system for the π-calculus described in Chapter 14 have the form "$\Delta \vdash B$" in which case predicates are contexts Δ. Judgments of a type system for Mobile Ambients described in Chapter 16 have the form "$\Delta \vdash B : \kappa$" in which case predicates are pairs (Δ, κ). Finally, Chapter 18 describes a flow analysis system of a biologically inspired process calculus BioAmbients with statements of the shape "$(\mathcal{S}, \mathcal{N}) \models^l B$", and in this case, predicates are triples $(\mathcal{S}, \mathcal{N}, l)$.

Predicates formally represent properties which the system reasons about and verifies. Secondly, the analysis system defines a binary relation on processes and predicates. Let us write the relation as $\rhd B : \rho$. This relation formally represents

the statement "B has the property ρ". The relation is desired to be effectively verifiable. Thirdly, the system (usually) enjoys the *subject reduction* property, which states that the relation $\triangleright$ is preserved under rewriting of processes, that is, $\triangleright B_0 : \rho$ and $B_0 \rightarrow B_1$ imply $\triangleright B_1 : \rho$.

Usually it is easy to verify that one process B has a specific property ρ. On the other hand, to verify that the process B and all its successors have the property ρ is in general a much more complicated task because the set of all successors of B can be infinite. Type systems considerably simplify this task because subject reduction implies that it is enough to verify ρ only for the initial state B, that is, it is enough to check $\triangleright B : \rho$.

1.5.1 Principal Typings

For every predicate ρ we can define its meaning $[\![\rho]\!]$ to be the set of all processes B such that $\triangleright B : \rho$. A *principal predicate* of a process B is a predicate such that $\triangleright B : \rho$ and $[\![\rho]\!] \subseteq [\![\rho_0]\!]$ for any other ρ_0 such that $\triangleright B : \rho_0$. Principal predicates are usually called *principal typings* [Wel02]. Existence of a principal typing for every process is a desirable property of an analysis system. It is important for efficient type inference, compositionally, and reusing of results and it is further discussed in Chapter 10.

1.5.2 Polymorphism

Polymorphic predicates uniformly describe behavior of processes which concerns values of various concrete types. Thus they support reusing of code in programming languages, and they allow more comfortable description and modeling of concurrent systems.

Some analysis systems for the π-calculus assign to every channel c the type of values that can be transmitted over c. These analysis systems can be, for example, used to guarantee that only integers are sent over channel c. This is useful to avoid type errors which can occur when a receiving process receives a value of unexpected type, for example, a string instead of an integer. Some channels can be, however, used to legally (that is, without a type error) transfer values of various concrete types. A typical example is the repeater process "`c(x).c<x>.0`" where the channel `c` can safely transfer a value of any type. Analysis systems whose predicates describe processes where the same channel name can be used to transfer values of various types are called *polymorphic*. This particular case of polymorphism is called *channel polymorphism*.

Another kind of polymorphism can be encountered in analysis systems of Mobile Ambients and similar systems which work with ambients. Some analysis systems for Mobile Ambients assign to every ambient n an allowed communication topic

which describes type values values that can be exchanged inside n. Some ambients can, however, allow exchange of values of different types depending on the position of the ambient in the ambient hierarchy. For example, the exchange of integers can be allowed in a when a is inside ambient b while the exchange of strings can be allowed in a when a is inside ambient c. We call this kind of polymorphism, where communication actions and capabilities allowed inside an ambient depends of the ambient position in the ambient hierarchy, *spatial polymorphism*. Spatial polymorphism was firstly describe in the PolyA system. Spatial polymorphism in the Poly✶ system is further discussed in Section 7.7.

1.6 The History of the Poly✶ System

Poly✶ was presented by Makholm and Wells [MW05] in 2005, previously presented in the technical report [MW04a] in 2004. Poly✶ was developed from the previous work of the above authors and Amtoft on PolyA [AMW04a, AMW04b]. PolyA is a type system for Mobile Ambients and it is motivated by the previous work of Amtoft and Wells [AW02].

1.6.1 The PolyA System

Unlike Poly✶, PolyA only works for one specific process calculus, Mobile Ambients. PolyA does not assign a fixed communication topic to each ambient as described in the previous section. Instead, it assigns a type to each process that gives upper bounds on (1) a possible ambient hierarchy tree contained in the process, (2) values that may be communicated, and (3) capabilities that may be used. PolyA allows, for example, typing of a messenger ambient that can collect a message of non-predetermined type and deliver it to a non-predetermined location. PolyA provides spatial polymorphism described in the previous section. Spatial polymorphism in PolyA means that a type of an ambient process may depend on a location where it is found.

Types in PolyA are *dependent* in the sense that they are build from the same building blocks as process (from names in PolyA and later in Poly✶ from type tags). PolyA types are selected from the set of *shape predicates*. A shape predicate is a graph which represents all possible future states of a process merged together. *Shape types* are those shape predicates which are provably closed under rewriting. The basic idea of shape predicates is that they resemble process structure and content. A shape predicate looks like a process term syntax tree. A process term matches the shape predicate if its syntax tree can be "bent into shape" described by the shape predicate (edges of the shape predicate can be used more than once during the matching). This basic idea comes to trouble because there may be a

term that can evolve to a term with an arbitrarily deep syntax tree (for example, "!a[!in a.0]" in Mobile Ambients). It means that we would have to consider infinite shape predicates. On the other hand, it is desired to keep types finite. Thus PolyA restricts itself to possibly infinite trees with finite representations, specifically, *regular trees*. Although PolyA defines a linear notion of shape predicates called *shape expression*, it is easiest to use directly graphs.

Not all shape predicates are *shape types* in PolyA. Because a desired property here is subject reduction, only those shape predicates that are closed under rewriting are called types. Shape predicates which are closed under rewriting are called *semantically closed* in PolyA. Because the recognition of semantic closure was found far from easy, another, easier to recognize notion of *syntactically closed* shape predicates is defined in PolyA. Syntactic closure implies the semantics closure. Syntactically closed shape predicates are called *shape types* in PolyA.

Although the recognition of shape types is relatively easy to implement, it is still not enough to prove existence of *principal typings* [Wel02]. That is why PolyA defines a subclass of types called *restricted* types. Restricted types are those which satisfy two conditions called a *discrete* and a *modest condition*[1]. Among restricted types, the existence of principal typings is proved and a type inference algorithm has been implemented [MW04b]. The existence of principal typings among all unrestricted PolyA types has never been either proved or disproved. As noted in Section 1.5, principal typings are important for efficient type inference algorithm, compositionally, and reusing of results.

1.6.2 From PolyA to Poly✶

The work on PolyA gives rise to Poly✶, a generalization from a type system for Mobile Ambients to a family of type systems for a large family of process calculi. Poly✶ takes from PolyA the concept of shape types. It provides the way to describe reduction semantics of the process calculus in question. Based on this description, the reduction relation is automatically inferred. Again, notions similar to the semantic and the syntactic closure from PolyA are defined in Poly✶. A notable change is that Poly✶ leaves off the *discrete* and *modest* restrictions, and instead, it defines simpler conditions on types called a *width* and a *depth restriction*. The main reasons are that the discrete and modest restrictions, although more powerful, were very complex and hard to understand. Types which satisfy the width and depth restriction are called *restricted* and the existence of principal typings is proved only among these restricted types.

Problems were found in the previously published Poly✶ [MW05, MW04a]. They are fixed and described in this thesis, which is the first publication which contains

[1] We will not even try to explain these conditions here because they are exceedingly complex.

detailed proofs of Poly✶ properties (subject reduction, principal typings, and others) as well as a detailed description and correctness proofs of the type inference algorithm.

1.7 Overview of the Poly✶ System

Poly✶ is a generic type system scheme which can be used to verify various properties of processes from various calculi. Poly✶ is built on top of the metacalculus Meta✶ which can be instantiated to many calculi including, for example, the π-calculus, Mobile Ambients, numerous variations of these, and other systems. The instantiation of Meta✶ to a process calculus is done by a straightforward description $\mathcal{R}$ of the rewriting rules in the syntax that Meta✶ provides for this purpose. A rule description $\mathcal{R}$ instantiates Meta✶ to the calculus $C_{\mathcal{R}}$ and the very same rule description $\mathcal{R}$ is the only thing that is necessary to instantiate Poly✶ to the type system $S_{\mathcal{R}}$ for $C_{\mathcal{R}}$.

The type system $S_{\mathcal{R}}$ provided by Poly✶ is not designed to verify and reason about just one specific property of processes. Rather, Poly✶ uses the generic notion of *shape predicates* which describe allowed syntactic configurations of Meta✶ processes. *Shape $\mathcal{R}$-types* are those shape predicates which are provably closed under rewriting with $\mathcal{R}$ by a simple procedure. Every shape ($\mathcal{R}$-)type Π describes the set of Meta✶ processes which have the syntactic configuration allowed by Π. Many interesting properties of processes can be expressed as properties of shape types. The type system $S_{\mathcal{R}}$ can be thus used to verify and reason about all the properties which can be expressed as properties of shape types. The question of the expressiveness of shape types is further investigated in Part III of this thesis where we formally prove that shape types can have both the same expressive power as and also superior expressive power than predicates of three selected analysis systems which earlier researchers handcrafted to verify specific process properties of specific calculi. Every Poly✶ instance $S_{\mathcal{R}}$ has desirable properties such as subject reduction, the existence of principal typings [Wel02], and an already-implemented type inference algorithm.

1.8 Contributions of the Thesis

The contributions of this is thesis are briefly summarized in the following list.

Extensions of the Poly✶ System. A major extension of the Poly✶ system introduced in this thesis is the support of name restriction ("ν"). Details can be found in Section 9.1. The support of name restriction is significant because the functionality of name restriction is very often used when modeling concurrent systems.

Fixes of the theory. Many problems were found in the theory of the previously published POLY✶ system. These are fixed and described in this thesis. Section 9.2 contains a detailed list of changes together with references to related parts of this thesis.

Clarifications of the theory. Some notations in the previously published POLY✶ theory were used in an informal way without an exact definition. Explicit definitions of these notions are provided in this thesis. Some definitions were also simplified and clarified. Details can be found in Section 9.2.

Formalization of a type inference algorithm. Although an implementation of the type inference algorithm accompanied the previously published POLY✶ system, no formal description of type inference was available before this thesis. The type inference algorithm is described and proved correct in Part II of this thesis.

Proofs. An enormous amount of detailed proofs can be found in this thesis. No proofs of POLY✶ properties were previously published except for a very short (1 page) proof sketch of subject reduction. The discovery of the above-mentioned inconsistencies in the original POLY✶ system called for much more detailed proofs. Proofs of subject reduction, principal typings, correctness of the type inference algorithm, and other important properties of the POLY✶ system are first presented in this thesis.

Expressiveness evaluation of POLY✶. POLY✶ can be instantiated to a type system for a large variety of process calculi. The expressiveness of type systems provided by POLY✶ has, however, not been evaluated before the presentation of results from this thesis. Part III of this thesis deals with this question of expressiveness and it shows that shape types can have both the same expressive power as and also superior expressive power than predicates of three quite dissimilar analysis systems from the literature, namely, (1) an implicitly typed π-calculus, (2) an explicitly typed Mobile Ambients, (3) and a flow analysis system for BioAmbients.. We believe that the results reached and the diversity of the three systems justify the claim that shape types can be widely used instead of predicates of many other systems.

Applications of POLY✶. Apart from the proofs of superior expressiveness, Part III of this thesis also shows on concrete examples how to use the POLY✶ system to achieve specific tasks. This helps to bridge over the problem of complexity of POLY✶ which is inevitably implied by its high generality and which has been daunting to some readers of earlier papers. We also demonstrate spatial polymorphism, which is not common for other systems, on concrete examples.

Handling of infinite sets of rewriting rules. The previously published POLY✶ system works only with process calculi with finite sets of rewriting rules. Some

process calculi support *polyadic* communication which allows exchange of arbitrarily long tuples of objects. Polyadic communication is usually described by infinitely many rewriting rules (a separate rule for every tuple arity). Handling of infinite sets of rewriting rules which is sufficient to support polyadic calculi is first presented in Section 10.3 of this thesis.

Non-existence of principal typings among unrestricted types. As noted in Section 1.6.2, the existence of principal types in Poly✶ is proved only among restricted types. Section 10.4 constructs a Poly✶ instantiation with no principal typings among all Poly✶ types. This result, which is first published in this thesis, explains the reasons for introduction of restricted types.

Part I of this thesis partially overlaps with previous Poly✶ publications of Makholm and Wells [MW05, MW04a]. It, however, contains some extensions and the differences are summarized in Section 9.2. The materials from Part II have not been published before. Part III is mainly based on recent publications of Jakubův and Wells [JW09, JW10].

1.9 Structure of the Thesis

Basic mathematical and other notations used throughout the thesis are introduced in Chapter 2. The rest of the thesis is divided into three parts which deal with the following topics.

Part I: Shape Types and the Poly✶ System. This part fixes and extends the generic Poly✶ type system previously published by Makholm and Wells [MW05, MW04a]. See Section 1.7 for basic overview of Poly✶ and shape types. Differences between the previously published Poly✶ system and the one presented in this thesis are summarized in Section 9.2.

Part II: Principal Typings and Type Inference. Part II presents a type inference algorithm and proves it to be correct and complete. The type inference algorithm together with the proof of its correctness and completeness provide a constructive proof of the existence of principal typings (see Section 1.5 and Chapter 10 about principal typings). These results are published for the first time in this thesis.

Part III: Application and Expressiveness of Shape Types. The last part demonstrates how to use Poly✶ with concrete calculi from the literature, namely, with the π-calculus, Mobile Ambients, and BioAmbients. For each of the three calculi we select its analysis system from the literature and we prove that shape types can provide the same results as the original analysis system. Furthermore, we prove that Poly✶ can additionally provide greater expressiveness than the original system.

Some topics are presented in two consecutive chapters where the first chapter provides a compact overview of the topic and the second chapter contains additional details, explanations, and proofs. The overview chapters contain all the definition necessary to comprehend the rest (of the overview chapters) of the thesis. The detail chapters can be skipped for the first reading and the reader can look them up later as necessary, either the whole chapter or just some particular parts. The chapter with details always follows the corresponding overview chapter and it is titled "Technical Details on ...".

Chapter 2

Notations and Definitions

This short chapter presents some definitions which are not specific to the work of this thesis.

Throughout this thesis let i, j, k range over natural numbers. Let (u, v) denote the pair of u and v. A function f is a pair set such that $(u, v) \in f$ and $(u, w) \in f$ implies $v = w$. Let $u \mapsto v$ be an alternate pair notation used when writing functions. Given the function f and the sets U and V we suppose the following definitions.

$\text{power}(U) = \{V : V \subseteq U\}$	the power set
$\text{power}_{\text{fin}}(U) = \{V : V \subseteq U \;\&\; V \text{ is finite}\}$	the set of finite subsets
$U \backslash V = \{u : u \in U \;\&\; u \notin V\}$	set subtraction
$U \times V = \{(u, v) : u \in U \;\&\; v \in V\}$	Cartesian product
$\text{dom}(f) = \{u : (u \mapsto v) \in f\}$	function domain
$\text{rng}(f) = \{v : (u \mapsto v) \in f\}$	function range
$f^{-1} = \{(v, u) : (u \mapsto v) \in f\}$	inverse function/relation
$f[u \mapsto v] = \{(u' \mapsto v') \in f : u \neq u'\} \cup \{u \mapsto v\}$	function extension/replacement
$U \to V = \{f \subseteq (U \times V) \mid f \text{ is a function}\}$	all functions from U to V
$U \to_{\text{fin}} V = \{f \in (U \to V) \mid f \text{ is finite}\}$	all finite functions from U to V

We shall use following BNF-like statements to define sets with members of a particular syntax.

$$i, j, k \quad \in \quad \text{Nat} \quad ::= \quad 0 \mid 1 \mid 2 \mid \cdots$$

The above statement defines a set called Nat to be the set of natural numbers $\{0, 1, \ldots\}$ and it states that metavariables i, j, and k (possibly with indexes) will be used to range over Nat. We also use similar statements with equality "=" instead of "::=" to describe a set directly. The following has the same meaning as the previous statement.

$$i, j, k \quad \in \quad \text{Nat} \quad = \quad \{0, 1, 2, \ldots\}$$

Syntactic sets can be additionally defined recursively as in the following example.

$$T \in \text{Term} ::= i \mid -T \mid (T_0 + T_1) \mid (T_0 * T_1)$$

The above set Term can be equivalently defined by the following recursive definition.

(1) Every natural number is in Term.

(2) When $T \in$ Term then "$-T$" is in Term.

(3) When T_0 and T_1 are from Term then "$(T_0 + T_1)$" and "$(T_0 * T_1)$" are in Term.

(4) Any member of Term is constructed by finitely many applications of (*1*)-(*3*).

We use metavariables consistently, that is, all occurrences of the same metavariable in the same chapter always range over the same set. Upper case metavariables range over more complicated (usually recursively defined) sets. Greek metavariables range over type entities. An index of metavariables can be found at the end of this thesis.

In Part II we describe a type inference algorithm using a C-like pseudo-programming language. We do not formally define its semantics in favor of the following description. We assume the call-by-value semantics, that is, every function call makes a copy of its arguments. Names of variables correspond to names of metavariables used throughout the thesis and thus the name of a variable determines the type of its value. For example, variables i and i_0 can hold only natural number values.

The scope of a variable is the whole function where it is used. Exceptions are variables which are introduced by an existential quantification in conditions of **if** and **while** statements. The scope of these existentially quantified variables is only the body of the **if** branch or the **while** cycle whose condition introduces the variable. Existentially quantified variables are read-only. Variables which are used as control variables in **for** and **foreach** cycles have also only the corresponding block as their scope and they are read-only. There are no global variables.

An attempt to read an uninitialized variable to which no value has been assigned yet terminates the execution with failure. Some assignment uses a simple pattern matching, for example "$(i, j) := (1, 2)$;". The execution terminates with failure when the right-hand side has not the required shape, like for example in "$(i, j) := 3$;". Finally, the execution terminates with failure if the argument of **switch** command has a shape which is not described by any **case** branch and there is no **otherwise** branch. Above failures do not, however, happen in algorithms from Part II if arguments have expected values. After the execution of a **case** branch which is not finished by **return**, the execution continues with the first command after the **switch** statement (and not by the next **case** branch like in C).

Part I

Shape Types and the Poly✶ System

Chapter 3

The Metacalculus Meta✶

Poly✶ is built on the metacalculus Meta✶ which is based on the observation that many syntactic constructions have similar semantics in many process calculi found in the literature. Examples of these constructions are *parallel composition* ("|"), prefixing a process with an executable or non-executable prefix ("."), replication ("!"), and *name restriction* ("ν"). Process calculi differ mainly in the set of prefixes and their meanings. Meta✶ collects constructors shared among process calculi and introduces a general concept of *forms* used to encode various prefixes of other calculi. Meta✶ is instantiated with a rewriting rule set $\mathcal{R}$ that specifies the behavior of prefixes (forms). Meta✶ can be instantiated to many calculi including, for example, the π-calculus, Mobile Ambients, numerous variations of these, and other systems.

We stress that metacalculus Meta✶ is mainly intended to provide a base for the generic type system Poly✶. Meta✶ is not supposed to be used on its own. In this chapter we describe generic syntax of Meta✶ processes together with general operations and relations which are used by all Meta✶ instances. How to instantiate Meta✶ to a concrete process calculus is described in Chapter 5.

3.1 Generic Syntax of Processes

Here we introduce the Meta✶ process syntax which is designed to allow straightforward encodings of other calculi and we introduce some useful conventions.

The syntax of Meta✶ processes is given in Figure 3.1. A Meta✶ *entity* is any entity defined in Figure 3.1, that is, any basic name, type tag, name, sequence, message, element, form, or process. Let metavariable Z range over all Meta✶ entities.

Processes in process calculi are usually built from atomic names. A Meta✶ name a^ι is a pair of the atomic *basic name* a and the *type tag* ι. Later we shall define α-conversion of bound names to preserve type tags. Thus the main point of type tags is to provide identifiers of bound names which are not changed by α-

a, b	$\in$	BasicName	$::=$	$\mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \mid \cdots \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{open} \mid \cdots \mid \mathsf{[]} \mid \bullet \mid \cdots$
ι	$\in$	TypeTag	$=$	BasicName
x, y	$\in$	Name	$::=$	a^{ι}
s	$\in$	Sequence	$::=$	$x_0 \ldots x_k$
M	$\in$	Message	$::=$	$\mathsf{0} \mid s \mid M_0.M_1$
E	$\in$	Element	$::=$	$x \mid \mathsf{(}x_1, \ldots, x_k\mathsf{)} \mid \mathsf{<}M_1, \ldots, M_k\mathsf{>}$
F	$\in$	Form	$::=$	$E_0 \ldots E_k$
P, Q, R	$\in$	Process	$::=$	$\mathsf{0} \mid F.P \mid (P \mid Q) \mid \nu x.P \mid !P$

Figure 3.1: Syntax of META✶ processes.

conversion. Type tags and basic names are taken from the same set and we shall abbreviate a^a simply as a when no confusion can arise. This abbreviation allows us to resemble process syntax of other calculi.

Process constructors have standard semantics. The *null process* "0" is an inactive or finished process, "$P \mid Q$" runs processes P and Q in *parallel*, "$\nu x.P$" behaves as P with *private name* x (i.e., x in P differs from all names outside P), and finally "$!P$" acts as infinitely many copies of P in parallel ("$P \mid P \mid \cdots$").

The *input element* $\mathsf{(}x_1, \ldots, x_k\mathsf{)}$ is used to encode name input binders of other calculi and it binds the names x_1, ..., x_k. The *output element* $\mathsf{<}M_1, \ldots, M_k\mathsf{>}$ can be used to encode message sending. Both elements can be empty (when $k = 0$). *Forms* can encode executable action prefixes from various calculi such as π-calculus communication actions (as "$x\mathsf{(}y\mathsf{)}$" and "$x\mathsf{<}y\mathsf{>}$") or Mobile Ambients capabilities (as "$\mathsf{in}\ x$", "$\mathsf{out}\ x$", and "$\mathsf{open}\ x$"). When a form F encodes an executable action then "$F.P$" encodes the process that runs F and continues as described by P. Forms are also used to encode non-executable prefixes or other calculus-specific constructions like ambients boundaries from Mobile Ambients. We encode the Mobile Ambient syntax "$x\mathsf{[}P\mathsf{]}$" in META✶ as "$x\mathsf{[]}.P$" and we use the former syntax as an abbreviation ("$\mathsf{[]}$" is a single name).

We omit parenthesis in $(P \mid Q)$ when possible. Let "." bind more tightly than "$\mid$", that is, "$F.P \mid Q = (F.P) \mid Q$" and "$\nu x.P \mid Q = (\nu x.P) \mid Q$". Let the composition of messages "." associate to the right, that is, "$M_0.M_1.M_2 = M_0.(M_1.M_2)$". Let "$\mid$" associate to the right.

Name restriction "νx" binds all underneath occurrences of x and the input element "$\mathsf{(}x_1, \ldots, x_k\mathsf{)}$" binds x_1, ..., x_k. The occurrence of type tag ι in a^{ι} is bound when this occurrence of a^{ι} is. For every P we define the set $\mathrm{fn}(P)$ of free names, the set $\mathrm{ftags}(P)$ of free type tags, the set $\mathrm{itags}(P)$ of input-bound type tags, the set $\mathrm{ntags}(P)$ of ν-bound type tags, and the set $\mathrm{tags}(P)$ of all type tags of process P. A detailed definition of these notions and sets can be found in Section 4.1.

A bound occurrence of a^{ι} can be α-converted to b^{ι} but the type tag ι has to be preserved. We identify α-convertible processes. Details on α-conversion can be

found in Section 4.2.

3.2 Well Formed Processes

Some names can have a special meaning in some process calculi like for example in, out, and open in Mobile Ambients. It is desirable not to allow these special names to be bound in META✶ processes. In the 2004 technical report [MW04a] special names can be bound and this causes an inconsistency in the subject reduction property. This is further discussed in Remark 8.7.2. Because of the introduction of type tags in the version of POLY✶ presented here, special names might be allowed to be bound but it would unnecessarily complicate proofs and encodings of other calculi.

We suppose that the set SpecialTag $\subset$ TypeTag contains all special type tags of names to which a special meaning is assigned by rewriting rules. The set is not fixed but can be extended as necessary to cover all special name tags in rewriting rules. For example in the case of Mobile Ambients we assess SpecialTag $= \{\bullet, [\,], \mathsf{in}, \mathsf{out}, \mathsf{open}\}$. We suppose $\bullet \in$ SpecialTag for any META✶ instantiation. We introduce the set SpecialTag here in order to make the following definition of well formed processes independent on descriptions of rewriting rules.

DEFINITION 3.2.1. *The process P is* **well formed** *when all the following hold.*

(W1) *The type tags* $\mathrm{ftags}(P) \cup \mathrm{ntags}(P)$ *are disjoint with* $\mathrm{itags}(P)$.

(W2) *When $F.Q$ is a subprocess of P then* $\mathrm{itags}(F)$ *and* $\mathrm{itags}(Q)$ *are disjoint.*

(W3) *Any F in P contains exactly one occurrence of ι for every $a^{\iota} \in \mathrm{bn}(F)$.*

(W4) *Bound tags* $\mathrm{itags}(P) \cup \mathrm{ntags}(P)$ *are disjoint with* SpecialTag. ∎

Henceforth we suppose only well scoped processes. Section 4.3 describes the changes in this definition from the previous META✶ [MW05]. The following remark explains the purpose of the conditions.

REMARK 3.2.2. All the well-formedness conditions except W3 are required only for subject reduction and type inference in POLY✶ and are not required for a proper functionality of META✶ itself.

Condition W1 forbids mixing of input bound tags with other (not input bound) tags. For example, "νx.x.0 | (x).x.0" and "a^{y}.0 | (b^{y}).b^{y}.0" are forbidden. Recall that a standalone x at a name position stands for x^{x}. Condition W1 is necessary for subject reduction and it is further discussed in Remark 6.3.3, Remark 8.2.1, Remark 8.5.2, and mainly in Remark 8.7.1. Free name tags and ν-bound tags can mix, for example, "x.0 | νx.x.0" is a well formed process. Processes like this have to be allowed to achieve the property that any subprocess of a well formed process is well formed. This property is essential for proofs which use structural induction on processes.

Message decomposition operator:

$$\mathtt{0}_* P = P \qquad s_* P = s.P \qquad (M_0.M_1)_* P = M_{0*}(M_{1*}P)$$

Application of a substitution to names, sequences, elements, and forms:

$$\bar{\mathbb{S}}x = \begin{cases} \mathbb{S}(x) & \text{if } \mathbb{S}(x) \in \text{Name} \\ x & \text{if } x \notin \text{dom}(\mathbb{S}) \\ \bullet & \text{otherwise} \end{cases}$$

$$\bar{\mathbb{S}}(x_0 \ldots x_k) = (\bar{\mathbb{S}}x_0) \ldots (\bar{\mathbb{S}}x_k)$$
$$\bar{\mathbb{S}}(x_1, \ldots, x_k) = (x_1, \ldots, x_k)$$
$$\bar{\mathbb{S}}\texttt{<}M_1, \ldots, M_k\texttt{>} = \texttt{<}\dot{\mathbb{S}}M_1, \ldots, \dot{\mathbb{S}}M_k\texttt{>}$$
$$\bar{\mathbb{S}}(E_0 \ldots E_k) = (\bar{\mathbb{S}}E_0) \ldots (\bar{\mathbb{S}}E_k)$$

Application of a substitution to messages:

$$\dot{\mathbb{S}}(M_0.M_1) = \dot{\mathbb{S}}M_0.\dot{\mathbb{S}}M_1 \qquad \dot{\mathbb{S}}s = \begin{cases} \mathbb{S}(x) & \text{if } s = x \in \text{dom}(\mathbb{S}) \\ \bar{\mathbb{S}}s & \text{otherwise} \end{cases}$$
$$\dot{\mathbb{S}}\mathtt{0} = \mathtt{0}$$

Application of a substitution to processes:

$$\bar{\mathbb{S}}\mathtt{0} = \mathtt{0} \qquad \bar{\mathbb{S}}(P \mid Q) = \bar{\mathbb{S}}P \mid \bar{\mathbb{S}}Q \qquad \bar{\mathbb{S}}(!P) = !\bar{\mathbb{S}}P$$
$$\bar{\mathbb{S}}(\nu x.P) = \nu x.\bar{\mathbb{S}}P \quad \text{if } x \notin \text{dom}(\mathbb{S}) \cup \text{fn}(\mathbb{S})$$
$$\bar{\mathbb{S}}(F.P) = \begin{cases} \mathbb{S}(x)_* \bar{\mathbb{S}}P & \text{if } F = x \in \text{dom}(\mathbb{S}) \\ \bar{\mathbb{S}}F.\bar{\mathbb{S}}P & \text{if } F \notin \text{dom}(\mathbb{S}) \;\&\; \text{bn}(F) \cap (\text{dom}(\mathbb{S}) \cup \text{fn}(\mathbb{S})) = \emptyset \end{cases}$$

Figure 3.2: Application of a substitution to META✶ entities.

Condition W2 forbids nesting of input binders which bind the same type tag. For example "$\texttt{(a}^{\texttt{y}}\texttt{).(b}^{\texttt{y}}\texttt{).b}^{\texttt{y}}\texttt{.0}$" is banned. This is essential for the subject reduction property to hold and it is further discussed in Remark 8.7.1. Condition W3 disallows a single type tag to be bound more than once in a single form. For example the following forms are not allowed in any well formed process: "$\texttt{(x,x)}$", "$\texttt{(x)(x)}$", "$\texttt{x(x)}$", and "$\texttt{(a}^{\texttt{x}}\texttt{,b}^{\texttt{x}}\texttt{)}$". These could lead to a formation of an invalid substitution. The purpose of W4 has already been discussed. ∎

3.3 Substitution

Now we define substitutions in META✶ and their actions on processes and other entities.

DEFINITION 3.3.1. *A* META✶ *substitution, denoted* $\mathbb{S}$*, is a finite function from* Name *to* Message. ∎

Application of $\mathbb{S}$ to various META✶ entities is defined in Figure 3.2. Application of $\mathbb{S}$ to messages is written $\dot{\mathbb{S}}M$ while application to all other META✶ entities is written $\bar{\mathbb{S}}Z$. Especially, application of $\mathbb{S}$ to process P is written $\bar{\mathbb{S}}P$. Let substitution application bind more tightly than other operators, that is, let "$\bar{\mathbb{S}}P \mid Q$" stand for "$(\bar{\mathbb{S}}P) \mid Q$".

Application of a substitution to processes uses an auxiliary message decomposition operation M_*P defined in the top part of Figure 3.2. It discards empty

$$\frac{}{P \equiv P}\ (\textsc{SRef}) \qquad \frac{P \equiv Q}{Q \equiv P}\ (\textsc{SSym}) \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}\ (\textsc{STra}) \qquad \frac{P \equiv Q}{P \mid R \equiv Q \mid R}\ (\textsc{SPar})$$

$$\frac{P \equiv Q}{F.P \equiv F.Q}\ (\textsc{SFrm}) \qquad \frac{P \equiv Q}{!P \equiv !Q}\ (\textsc{SRep}) \qquad \frac{P \equiv Q}{\nu x.P \equiv \nu x.Q}\ (\textsc{SNu})$$

$$\frac{}{P \mid Q \equiv Q \mid P}\ (\textsc{SPCom}) \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}\ (\textsc{SPAsc}) \qquad \frac{}{P \mid \mathsf{0} \equiv P}\ (\textsc{SPNul})$$

$$\frac{}{\mathsf{0} \equiv !\mathsf{0}}\ (\textsc{SRNul}) \qquad \frac{}{\nu x.\nu y.P \equiv \nu y.\nu x.P}\ (\textsc{SNuNu}) \qquad \frac{}{!P \equiv P \mid !P}\ (\textsc{SBang})$$

$$\frac{x \notin \mathrm{fn}(F)}{F.\nu x.P \equiv \nu x.F.P}\ (\textsc{SNuFrm}) \qquad \frac{x \notin \mathrm{fn}(P)}{P \mid \nu x.Q \equiv \nu x.(P \mid Q)}\ (\textsc{SNuPar})$$

Figure 3.3: Structural equivalence of META✶.

messages $\mathsf{0}$ from M and pushes components of M from right to left onto P (for example $((\mathsf{a}.\mathsf{b}).\mathsf{c})_* P = \mathsf{a}.\mathsf{b}.\mathsf{c}.P$). In other calculi this operation is often incorporated into a structural equivalence relation.

Substitution replaces names by messages, but non-name messages are META✶ syntax errors at some name positions. For example, substituting "$\mathsf{in\ a}$" for x in "$\mathsf{open\ x}$" would yield "$\mathsf{open\ (in\ a)}$" which is invalid syntax. In some process calculi, the syntax allows such expressions but they are semantically inert. In META✶, substitution places a special name "$\bullet$" at positions that would otherwise be syntax errors, that is, the above substitution yields "$\mathsf{open}\ \bullet$".

Note that we really need two different application operators because a substitution is applied in different ways to names inside forms or sequences, and to single name messages. For example, when $\mathbb{S} = \{\mathsf{x} \mapsto \mathsf{in\ a}\}$ then "$\dot{\mathbb{S}}\mathsf{x} = \mathsf{in\ a}$" but "$\bar{\mathbb{S}}\mathsf{x} = \bullet$" and hence we have "$\bar{\mathbb{S}}(\mathsf{x}\langle\mathsf{x}, \mathsf{in\ x}\rangle) = \bullet\langle\mathsf{in\ a}, \mathsf{in}\ \bullet\rangle$". The result of $\bar{\mathbb{S}}$ applied to a name is always a name. Also note that substitution application does not touch names inside input elements, that is, we have for example $\bar{\mathbb{S}}(\mathsf{(x)}) = \mathsf{(x)}$ for any $\mathbb{S}$ even when $\mathsf{x} \in \mathrm{dom}(\mathbb{S})$. This is because input-elements act as binders.

Basic properties of substitutions are proved in Section 4.6. In Section 4.5 we describe changes in the definition of substitution application from the previous META✶ [MW05].

3.4 Structural Equivalence

The META✶ *structural equivalence* relation $\equiv$ is the smallest binary relation on META✶ processes that satisfies the rules in Figure 3.3. Labels like SRef or SSym

are rule names with no impact on semantics. Structural equivalence expresses the following standard properties of parallel composition, name restriction, and replication. Parallel composition is commutative and associative and has `0` as its unit (rules SPCOM, SPASC, and SPNUL). The scope of name restriction can be extruded from name restriction, parallel composition, and form when there is no binding conflict (rules SNUNU, SNUPAR, and SNUFRM). Replication implements repetitive behavior (rule SBANG). This basic semantics of operators described by structural equivalence is fixed and does not vary with instantiation of META✶. Basic properties of structural equivalence are proved in Section 4.7.

Chapter 4

Technical Details on META✶

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

4.1 Free and Bound Names

We proceed by defining notions of free and bound names and type tags. These notions are used to define α-equivalence in the next section.

DEFINITION 4.1.1. *All occurrences of the name x in "$\nu x.P$" are called* **(ν-)bound**. *When a form F contains an element "$(x_1, \ldots, x_k)$" then all occurrences of x_1, ..., x_k in "$F.P$" as well as in F on its own are called* **(input-)bound**. *An occurrence of x that is not bound is called* **free**. *The occurrence of ι in a^ι is called bound (resp. free) when this occurrence of a^ι is.* ∎

The set $\mathrm{fn}(P)$ of free names of P and the set $\mathrm{bn}(F)$ of bound names of the form F are defined in Figure 4.1. We do not define, however, the set of bound names of a process because we identify processes up to α-conversion and this set would not be preserved under α-conversion of bound names in the process.

The function $\overline{x}$ which extracts the type tag from the name x and the element-wise extension of this function to sets of names are defined in the top part of Figure 4.2. The set $\mathrm{ftags}(P)$ of free type tags, the set $\mathrm{itags}(P)$ of input-bound type tags, the set $\mathrm{ntags}(P)$ of ν-bound type tags, and the set $\mathrm{tags}(P)$ of all type tags of process P are defined in Figure 4.2. In contrast to the previous, we define the sets of bound type tags for processes because we require type tags to be preserved under α-conversion.

4.2 Name Swapping and α-equivalence

Different occurrences of the same bound name under different binders are supposed to be handled as occurrences of different names. For example, the process "νa.in a.0 |

Free names of sequences and messages:

$$\mathrm{fn}(x_0 \ldots x_k) = \{x_0, \ldots, x_k\} \qquad \mathrm{fn}(\mathsf{0}) = \emptyset \qquad \mathrm{fn}(M_0.M_1) = \mathrm{fn}(M_0) \cup \mathrm{fn}(M_1)$$

Free and bound names of elements and forms:

$$\begin{array}{ll} \mathrm{fn}(x) = \{x\} & \mathrm{bn}(x) = \emptyset \\ \mathrm{fn}(\mathsf{(}x_1, \ldots, x_k\mathsf{)}) = \emptyset & \mathrm{bn}(\mathsf{(}x_1, \ldots, x_k\mathsf{)}) = \{x_1, \ldots, x_k\} \\ \mathrm{fn}(\mathsf{<}M_1, \ldots, M_k\mathsf{>}) = \bigcup_{i=1}^{k} \mathrm{fn}(M_i) & \mathrm{bn}(\mathsf{<}M_1, \ldots, M_k\mathsf{>}) = \emptyset \\ \mathrm{fn}(E_0 \ldots E_k) = \bigcup_{i=0}^{k} \mathrm{fn}(E_i) & \mathrm{bn}(E_0 \ldots E_k) = \bigcup_{i=0}^{k} \mathrm{bn}(E_i) \end{array}$$

Free names of processes:

$$\begin{array}{ll} \mathrm{fn}(F.P) = \mathrm{fn}(F) \cup (\mathrm{fn}(P) \backslash \mathrm{bn}(F)) & \mathrm{fn}(\mathsf{0}) = \emptyset \\ \mathrm{fn}(P \mid Q) = \mathrm{fn}(P) \cup \mathrm{fn}(Q) & \mathrm{fn}(\nu x.P) = \mathrm{fn}(P) \backslash \{x\} \\ & \mathrm{fn}(!P) = \mathrm{fn}(P) \end{array}$$

Figure 4.1: Free and bound names of META✶ process entities.

Type tags of names and sets of names:

$$\overline{a^{\iota}} = \iota \qquad \text{for } X \subseteq \text{Name: } \overline{X} = \{\overline{x} \colon x \in X\}$$

Input-bound and ν-bound type tags of processes:

$$\begin{array}{ll} \mathrm{itags}(F) = \overline{\mathrm{bn}(F)} & \\ \mathrm{itags}(F.P) = \mathrm{itags}(F) \cup \mathrm{itags}(P) & \mathrm{ntags}(F.P) = \mathrm{ntags}(P) \\ \mathrm{itags}(\nu x.P) = \mathrm{itags}(P) & \mathrm{ntags}(\nu x.P) = \{\overline{x}\} \cup \mathrm{ntags}(P) \\ \mathrm{itags}(\mathsf{0}) = \emptyset & \mathrm{ntags}(\mathsf{0}) = \emptyset \\ \mathrm{itags}(P \mid Q) = \mathrm{itags}(P) \cup \mathrm{itags}(Q) & \mathrm{ntags}(P \mid Q) = \mathrm{ntags}(P) \cup \mathrm{ntags}(Q) \\ \mathrm{itags}(!P) = \mathrm{itags}(P) & \mathrm{ntags}(!P) = \mathrm{ntags}(P) \end{array}$$

Free and bound type tags of processes:

$$\mathrm{ftags}(P) = \overline{\mathrm{fn}(P)} \qquad \mathrm{tags}(P) = \mathrm{ftags}(P) \cup \mathrm{itags}(P) \cup \mathrm{ntags}(P)$$

Figure 4.2: Free and all type tags of META✶ process entities.

νa.out a.0" should behave as the process "νa.in a.0 | νb.out b.0". Because names under different binders can interact with each other, we eventually need to rename bound names to avoid name conflicts. Renaming of bound names is commonly referred to as α-conversion and two processes which differ only by renaming of bound names are called α-convertible or α-equivalent. We now define α-conversion for META✶ processes using the name swapping operation.

DEFINITION 4.2.1. *Let $(x \rightleftarrows y)P$ be the process P with all occurrences (free, bound, or binding) of x and y* **swapped**. ∎

The *α-equivalence relation* $P \sim Q$ is the smallest binary relation on META✶ processes which satisfies the rules from Figure 4.3. In other words, it is the smallest equivalence relation congruent with META✶ process constructors which satisfies rule

$$\frac{}{P \sim P}\ (\text{AREF}) \qquad \frac{P \sim Q}{Q \sim P}\ (\text{ASYM}) \qquad \frac{P \sim Q \quad Q \sim R}{P \sim R}\ (\text{ATRA})$$

$$\frac{P \sim Q}{P \mid R \sim Q \mid R}\ (\text{APAR}) \qquad \frac{P \sim Q}{F.P \sim F.Q}\ (\text{AFRM}) \qquad \frac{P \sim Q}{!P \sim !Q}\ (\text{AREP})$$

$$\frac{P \sim Q}{\nu x.P \sim \nu x.Q}\ (\text{ANU}) \qquad \frac{a^\iota \notin \text{fn}(P) \quad b^\iota \notin \text{fn}(P)}{P \sim (a^\iota \leftrightarrows b^\iota)P}\ (\text{ASWAP})$$

Figure 4.3: The α-equivalence relation.

ASWAP. Processes P and Q are called *α-convertible* when $P \sim Q$. Henceforth α-convertible processes are *identified*, that is, considered equal.

Without α-equivalence being defined as congruent with process constructors the following could not be proved:

$$(\nu \mathsf{a}^\iota.\mathsf{0} \mid \nu \mathsf{a}^\iota.\mathsf{0}) \sim (\nu \mathsf{a}^\iota.\mathsf{0} \mid \nu \mathsf{b}^\iota.\mathsf{0})$$

Because free names and type tags are preserved under α-equivalence, it is easy to see that all previously defined functions on processes give equal values for α-convertible processes and thus these functions are still correctly defined functions after the identification of α-equivalent processes. Also note that we have α-identified processes but not forms which is to say that the forms "$\mathsf{(a^a)}$" and "$\mathsf{(b^a)}$" are different but the processes "$\mathsf{(a^a).0}$" and "$\mathsf{(b^a).0}$" are equal. Thus the function $\text{bn}(F)$ is still correctly defined on forms.

4.3 Changes in Well-Formedness

This section describes the differences in handling of α-renaming and well-formedness between the version of META✶ presented here and the META✶ version previously published in the ESOP 2005 paper [MW05] and in the 2004 technical report [MW04a].

The definition of well formed processes is different in the previous version of META✶ [MW05, MW04a]. In the previous META✶, names in processes had no type tags assigned to them and α-equivalent processes were not identified. Instead, α-renaming of ν-bound names was built into the structural equivalence relation. Moreover, input-bound names were not α-renamed at all. The need to α-rename input binders can be avoided in any process calculus where the rewriting rules can not invent processes with nested input-binders binding the same name. This is the case of majority of process calculi in the literature with the exception of the Higher-Order π-calculus (HOπ) [San93] where the following rewritings can happen, starting

with a process where all binders bind different names.

$$\mathsf{d(Z).(Z \mid a\langle Z\rangle) \mid d\langle a(Y).b(x).Y\rangle} \rightarrow_{\mathrm{HO}\pi}$$
$$\mathsf{a(Y).b(x).Y \mid a\langle a(Y).b(x).Y\rangle} \rightarrow_{\mathrm{HO}\pi}$$
$$\mathsf{b(x).a(Y).b(x).Y}$$

When we think about input-binders as about λ-abstractions in the λ-calculus then the λ-calculus does not satisfy the above property either. Let us consider the following reductions.

$$(\lambda \mathsf{z.zz})(\lambda \mathsf{xy.xy}) \rightarrow_{\beta} (\lambda \mathsf{xy.xy})(\lambda \mathsf{xy.xy}) \rightarrow_{\beta} \lambda \mathsf{y.}(\lambda \mathsf{xy.xy})\mathsf{y} \rightarrow_{\beta\alpha} \lambda \mathsf{yy'.yy'}$$

Note that α-renaming was required in the last reduction step to avoid name capture. This means that we can not directly instantiate META✶ to HOπ and to the (call-by-value) λ-calculus. We can still, however, work with both HOπ and with the λ-calculus indirectly, for example, via their encodings in the π-calculus. We could also directly instantiate META✶ to the name passing λ-calculus [Bou97, Table 1].

Thus α-renaming of input-binders can be avoided in a process calculus which meets the following requirements.

(1) It is possible to require nested input-binders to bind different name.

(2) It is possible to require input-bound names to be disjoint with other names.

(3) The properties required by (*1*) and (*2*) are preserved under rewriting.

One still, however, needs to α-rename ν-binders because a single ν-binder can be replicated and the replicated copy can extend its scope to contain the original binder.

All the above three requirements are met for the most of process calculi found in the literature including the π-calculus, Mobile Ambient, and their variants. Thus the need to α-rename input-binders was avoided in the previous META✶. Unfortunately, there was a mistake in the definition of well formed processes which allowed name captures. This is further discussed in Section 4.5.

4.4 Properties of Well Formed Processes

The following proves that a well formed process has only well formed subprocesses. This property is essential for proofs that that use structural induction on processes because we implicitly assume that all processes are well formed. Thus without this property one would not be able to apply the induction hypothesis because the proof of the induction hypothesis can make use of this implicit assumption and thus the induction hypothesis can be valid only for well formed processes.

LEMMA 4.4.1. *A subprocess of a well formed process is well formed.*

PROOF. *Let Q be a subprocess of a well formed process P. Clearly* W2, W3, *and* W4 *have to be satisfied for Q. Let us check* W1 *for Q. Let $\iota_0 \in \mathrm{itags}(Q)$. Clearly $\iota_0 \in \mathrm{itags}(P)$. We need to proof that $\iota_0 \neq \iota_1$ for any $\iota_1 \in \mathrm{ftags}(Q) \cup \mathrm{ntags}(Q)$. When $\iota_1 \in \mathrm{ntags}(Q)$ then clearly $\iota_1 \in \mathrm{ntags}(P)$ and thus $\iota_0 \neq \iota_1$. On the other hand $\iota_1 \in \mathrm{ftags}(Q)$ does not necessarily imply $\iota_1 \in \mathrm{ftags}(P)$ because ι_1 can be bound by some binder in P with Q is its scope. So let $\iota_1 \in \mathrm{ftags}(Q)$. Now ι_1 has to be in $\mathrm{tags}(P)$. When $\iota_1 \in \mathrm{ftags}(P)$ or $\iota_1 \in \mathrm{ntags}(P)$ then clearly $\iota_0 \neq \iota_1$ because otherwise P would not be well formed as $\iota_0 \in \mathrm{itags}(P)$. Finally, when $\iota_1 \in \mathrm{itags}(P)$ then P has some subprocess $F.P_0$ such that $\iota_1 \in \mathrm{itags}(F)$ and P_0 has Q as a subprocess. Clearly $\iota_0 \in \mathrm{itags}(Q)$ implies $\iota_0 \in \mathrm{itags}(P_0)$ and thus* W2 *for P says that $\iota_0 \neq \iota_1$. Hence Q is well formed.* ∎

4.5 Changes in Substitution Application

This section describes issues regarding the definition of a substitution and well-formedness of processes in the previous version of META✶ [MW05, MW04a].

As mentioned in Section 4.3, in the previous META✶ [MW05, MW04a] α-equivalent processes were not identified and type tags were not used. For every process P, the set $\mathrm{FN}(P)$ of free names of P and the set $\mathrm{BN}(P)$ of the input-bound names of P were defined in the previous META✶. Hence ν-bound names of P were neither in $\mathrm{FN}(P)$ nor in $\mathrm{BN}(P)$. Then the definition of a well formed process[1] in the previous META✶ [MW05, Section 2.2] was as follows.

> The process term P is *well scoped* iff it contains no nested binding of the same name and none of its free names also appear bound in the term. Formally, it is required that (1) $\mathrm{BN}(P)$ and $\mathrm{FN}(P)$ are disjoint, (2) whenever P contains $F.Q$, $\mathrm{BN}(F)$ and $\mathrm{BN}(Q)$ are disjoint, and (3) whenever P contains $\nu x.Q$, $x \notin \mathrm{BN}(Q)$.

Unfortunately this definition accidentally labels the process "(x).νx.x.0" as well scoped because, as stated above, ν-bound names are not in the set $\mathrm{BN}(P)$ of bound names. Moreover, application of substitution $\mathcal{S}$ to process P, which was written $\mathcal{S}^{\mathbf{P}}P$ in the previous META✶, did not guard against name capture which was justified as follows [MW05, Section 2.3].

> The definitions in Figure 3 do not worry about name capture. In general, therefore, $\mathcal{S}^{\mathbf{P}}P$ is only intuitively correct if $\mathrm{BN}(P)$ is disjoint from the names mentioned in $\mathcal{S}$. In practice, this will always follow from the assumption that all terms are well scoped.

[1] Actually the phrasing "well scoped process" is used in the previous META✶.

As a result the following name capture occurred when we had applied $\mathcal{S} = \{\mathsf{x} \mapsto \mathsf{a}\}$ to the afore mentioned process "$(\mathsf{x}).\nu\mathsf{x}.\mathsf{x}.0$".

$$\mathcal{S}^{\mathbf{P}}((\mathsf{x}).\nu\mathsf{x}.\mathsf{x}.0) = \nu\mathsf{x}.\mathsf{a}.0$$

This is a problem because the instantiation of the previous META✶ to some process calculus did not behave as the original calculus. For example, in the instantiation of the previous META✶ to Mobile Ambients one obtained that "$(\mathsf{x}).\nu\mathsf{x}.\mathsf{x}.0 \mid \langle\mathsf{a}\rangle.0$" rewrites to "$\nu\mathsf{x}.\mathsf{a}.0$" which does not happen in Mobile Ambients. This issue can, however, be solved by fixing the definition of well-scopedness[2].

Another related issue is that the definition of well scoped processes in the previous META✶ did not ruled out invalid forms like "(x, x)" which can lead to formation of an invalid substitution. The last issue is that special names mentioned in rewriting rules were not prevented from being bound. This breaks the subject reduction of the extension of POLY✶ from the 2004 technical report which handles name restriction [MW04a, Section 5.3]. This is further discussed in Remark 8.7.2.

4.6 Properties of Substitutions

Here we prove some trivial properties of substitution applications which will be used later. The following defines the set $\mathrm{fn}(\mathbb{S})$ of free names of $\mathbb{S}$.

DEFINITION 4.6.1. *Let* $\mathrm{fn}(\mathbb{S})$ *be the names in messages in the range of* $\mathbb{S}$*. Formally* $\mathrm{fn}(\mathbb{S}) = \{x \in \mathrm{fn}(M) \colon M \in \mathrm{rng}(\mathbb{S})\}$. ∎

The following lemma says that a substitution application does not change bound type tags in a process, that is, it can neither introduce a new type tag nor discard an existing one.

LEMMA 4.6.2. *All of the following hold for any* F*,* P*, and* $\mathbb{S}$*.*

(1) $\mathrm{bn}(\bar{\mathbb{S}}F) = \mathrm{bn}(F)$

(2) $\mathrm{itags}(\bar{\mathbb{S}}P) = \mathrm{itags}(P)$

(3) $\mathrm{ntags}(\bar{\mathbb{S}}P) = \mathrm{ntags}(P)$

PROOF. *By induction on the structure of* F *or* P *using (1) to prove (2).* ∎

In contrast, a substitution application can introduce a new free name. However, any newly introduced name is either from the range of the substitution or it is $\bullet$ in the case of a syntactic error.

[2]Then one has to also restrict α-renaming of ν-binders so that a ν-bound name is not renamed to some name which is input-bound elsewhere. This fix has actually never been done.

LEMMA 4.6.3. *All of the following hold for any F, P, and $\mathbb{S}$.*

(1) $\mathrm{fn}(\bar{\mathbb{S}}F) \subseteq \mathrm{fn}(F) \cup \mathrm{fn}(\mathbb{S}) \cup \{\bullet\}$

(2) $\mathrm{fn}(\bar{\mathbb{S}}P) \subseteq \mathrm{fn}(P) \cup \mathrm{fn}(\mathbb{S}) \cup \{\bullet\}$

PROOF. *By induction on the structure of F or P using (1) to prove (2).* ∎

4.7 Properties of Structural Equivalence

Basic properties of structural equivalence are proved in this section. The following lemma states that structurally equivalent processes have the same bound tags and free names.

LEMMA 4.7.1. *When $P \equiv Q$ then all the following hold.*

(1) $\mathrm{itags}(P) = \mathrm{itags}(Q)$

(2) $\mathrm{ntags}(P) = \mathrm{ntags}(Q)$

(3) $\mathrm{fn}(P) = \mathrm{fn}(Q)$

PROOF. *Proof by induction on the derivation of $P \equiv Q$. The only two cases which are not absolutely trivial are the following two cases of (3).*

SNUFRM: *Here $P = F.\nu x.P_0$ and $Q = \nu x.F.P_0$ for some x, F, and P_0. Moreover we have the following sets of free names*

$$\mathrm{fn}(P) = \mathrm{fn}(F) \cup \big((\mathrm{fn}(P_0)\backslash\{x\})\backslash\mathrm{bn}(F)\big)$$
$$\mathrm{fn}(Q) = \big(\mathrm{fn}(F) \cup (\mathrm{fn}(P_0)\backslash\mathrm{bn}(F))\big)\backslash\{x\}$$

which are equal because $x \notin \mathrm{fn}(F)$.

SNUPAR: *Here $P = P_0 \mid \nu x.Q_0$ and $Q = \nu x.(P_0 \mid Q_0)$ for some x, P_0, and Q_0. Moreover we have the following sets of free names*

$$\mathrm{fn}(P) = \mathrm{fn}(P_0) \cup (\mathrm{fn}(Q_0)\backslash\{x\})$$
$$\mathrm{fn}(Q) = (\mathrm{fn}(P_0) \cup \mathrm{fn}(Q_0))\backslash\{x\}$$

which are equal because $x \notin \mathrm{fn}(P_0)$. ∎

The following lemma says that structural equivalence preserves well-formedness of processes.

LEMMA 4.7.2. *Let $P \equiv Q$. Then P is well formed iff Q is well formed.*

PROOF. *Let $P \equiv Q$ and let P be well formed. We implicitly assume that all processes are well formed and thus Lemma 4.7.1 is valid only for well formed processes. But by an inspection of its proof it is easy to check the lemma does not make use of this implicit assumption and thus it is valid even for non-well formed processes. Thus clearly* W1 *and* W4 *are satisfied for Q by Lemma 4.7.1. Also* W3 *is satisfied for Q because structural equivalence does not introduce new forms. Finally,* W2 *is satisfied for Q because structural equivalence neither changes nesting of input-binders nor it can introduce a new input-binder.* ∎

Chapter 5

Instantiations of META✶

Semantics of many process calculi is given by a rewriting system which defines a binary rewriting relation on processes. Different calculi usually contain similar structural rewriting rules and main differences are found in the rewriting axioms. These axioms specify the semantics of action and other prefixes which we encode in META✶ using forms. Thus instead of fixing the semantics of forms, META✶ provides syntax for specifying rewriting rules that give meaning to forms and also defines how these rules yield a rewriting relation on processes.

The only thing necessary to instantiate META✶ to a working process calculus is to provide a straightforward description $\mathcal{R}$ of its rewriting axioms. This instantiates META✶ to the calculus $C_{\mathcal{R}}$ with the rewriting relation $\overset{\mathcal{R}}{\hookrightarrow}$. The same $\mathcal{R}$ is the only thing required to instantiate POLY✶ to the type system $S_{\mathcal{R}}$ for $C_{\mathcal{R}}$. Thus one obtains for free a type system for any calculus whose rewriting rules can be described in the META✶ syntax. This section describes this syntax of rule descriptions $\mathcal{R}$ and how the process calculus $C_{\mathcal{R}}$ is obtained.

5.1 Templates and Rewriting Rule Descriptions

Figure 5.1 presents the syntax used to describe rewriting rules. Process templates are used to describe both left and right-hand sides of rewriting rules. Template syntax resembles the syntax of processes except that leaves of syntax trees can be variables in addition to names. Name, message, and process variables are used in rules at positions of metavariables which range over arbitrary names, messages and processes respectively. Element templates describe elements of a specific shape. Similarly form templates describe forms, and process templates describe processes. In element templates, name variables describe positions in an element where an arbitrary name can occur. In contrast, a specific META✶ name in an element template requires the exactly same name to appear in an element at the specified position. For example, the form template "in å" describes all 2-length forms whose first element is name in

$$
\begin{array}{rcll}
\mathring{x}, \mathring{y} & \in & \text{NameVar} & ::= \ \mathtt{\mathring{a}} \mid \mathtt{\mathring{b}} \mid \mathtt{\mathring{c}} \mid \cdots \\
\mathring{m} & \in & \text{MessageVar} & ::= \ \mathtt{\mathring{M}} \mid \mathtt{\mathring{N}} \mid \cdots \\
\mathring{p} & \in & \text{ProcessVar} & ::= \ \mathtt{\mathring{P}} \mid \mathtt{\mathring{Q}} \mid \mathtt{\mathring{R}} \mid \cdots \\
\mathring{s} & \in & \text{Substitute} & ::= \ \mathring{x} \mid \mathring{m} \\
\mathring{E} & \in & \text{ElementTpl} & ::= \ x \mid \mathring{x} \mid \mathtt{(}\mathring{x}_1, \ldots, \mathring{x}_k\mathtt{)} \mid \mathtt{<}\mathring{m}_1, \ldots, \mathring{m}_k\mathtt{>} \\
\mathring{F} & \in & \text{FormTpl} & ::= \ \mathring{E}_0 \ldots \mathring{E}_k \\
\mathring{P}, \mathring{Q} & \in & \text{ProcessTpl} & ::= \ \mathtt{0} \mid \mathring{p} \mid \mathring{F}.\mathring{P} \mid (\mathring{P} \mid \mathring{Q}) \mid \mathtt{\{}\mathring{x}_0 \mathtt{:=} \mathring{s}_0, \ldots, \mathring{x}_k \mathtt{:=} \mathring{s}_k\mathtt{\}}\,\mathring{p} \\
\mathring{L} & \in & \text{Rule} & ::= \ \textbf{rewrite}\mathtt{\{}\,\mathring{P} \hookrightarrow \mathring{Q}\,\mathtt{\}} \mid \textbf{active}\mathtt{\{}\,\mathring{p}\ \textbf{in}\ \mathring{P}\,\mathtt{\}} \\
\mathcal{R} & \in & \text{RuleSet} & = \ \text{power}_{\text{fin}}(\text{Rule})
\end{array}
$$

Figure 5.1: Syntax of META✶ templates and rule descriptions.

and whose second element is an arbitrary name.

Let metavariable $\mathring{z}$ range over template variables, that is, name, message, and process variables. Let metavariable $\mathring{Z}$ range over all template entities, that is, over all the entities defined in Figure 5.2 except rules and rule sets. In templates, we use the same abbreviation for ambient syntax as in processes, that is, "$\mathring{E}\mathtt{[}\mathring{P}\mathtt{]}$" stands for "$\mathring{E}\mathtt{[]}.\mathring{P}$".

Variables in templates are replaced during rule instantiation by values of appropriate sorts, that is, name variables by names, message variables by messages, and processes variables by processes. A substitution application template "$\mathtt{\{}\mathring{x}_0 \mathtt{:=} \mathring{s}_0, \ldots, \mathring{x}_k \mathtt{:=} \mathring{s}_k\mathtt{\}}\,\mathring{p}$" describes a substitution to be applied on the right-hand side of some rule. A single process variable template $\mathring{P} = \mathring{p}$ could be as well implemented as "$\mathring{P} = \mathtt{\{\}}\ \mathring{p}$" which would reduce the number of different grammatical cases in the definition of process templates. However, we find it useful to separate the two cases in order to clarify the presentation. The **rewrite** rules specify ordinary rewriting rules while **active** rules describe rewriting contexts, that is, positions in processes other than at top-level where rewriting rules are to be applied. For example, in Mobile Ambients rewriting rules can be applied inside any ambient which is expressed by the rule "**active**$\mathtt{\{}\,\mathring{\mathtt{P}}$ **in** $\mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{P}}\mathtt{]}\,\mathtt{\}}$". In contrast, the rule "**active**$\mathtt{\{}\,\mathring{\mathtt{P}}$ **in** $\mathtt{a}\mathtt{[}\mathring{\mathtt{P}}\mathtt{]}\,\mathtt{\}}$" where name variable $\mathring{\mathtt{a}}$ is changed to an ordinary name $\mathtt{a}$, would allow rewriting rules to be applied only in the specific ambient with name $\mathtt{a}$.

Process instantiations fills in values for variables in templates and thus they turn process templates into processes.

DEFINITION 5.1.1. *A* **process instantiation** $\mathbb{P}$ *is a finite function which maps* NameVar *to* Name$\backslash\{\bullet\}$*,* MessageVar *to* Message*, and* ProcessVar *to* Process. ∎

Application of $\mathbb{P}$ to $\mathring{P}$, written $\mathbb{P}[\![\mathring{P}]\!]$, instantiates template $\mathring{P}$ to make a process by filling in values for variables in $\mathring{P}$ as assigned by $\mathbb{P}$. We forbid the name "$\bullet$" as the value of some name variable to prevent distinct earlier error results from being

$$\frac{\mathbf{rewrite}\{\,\mathring{P} \hookrightarrow \mathring{Q}\,\} \in \mathcal{R}}{\mathbb{P}[\![\mathring{P}]\!] \overset{\mathcal{R}}{\hookrightarrow} \mathbb{P}[\![\mathring{Q}]\!]}\ (\text{RRW}) \qquad \frac{\mathbf{active}\{\,\mathring{p}\ \mathbf{in}\ \mathring{P}\,\} \in \mathcal{R} \quad P \overset{\mathcal{R}}{\hookrightarrow} Q}{(\mathbb{P}[\mathring{p} \mapsto P])[\![\mathring{P}]\!] \overset{\mathcal{R}}{\hookrightarrow} (\mathbb{P}[\mathring{p} \mapsto Q])[\![\mathring{P}]\!]}\ (\text{RACT})$$

$$\frac{P \overset{\mathcal{R}}{\hookrightarrow} Q}{P \mid R \overset{\mathcal{R}}{\hookrightarrow} Q \mid R}\ (\text{RPAR}) \qquad \frac{P \overset{\mathcal{R}}{\hookrightarrow} Q}{\nu x.P \overset{\mathcal{R}}{\hookrightarrow} \nu x.Q}\ (\text{RNU}) \qquad \frac{P' \equiv P \quad P \overset{\mathcal{R}}{\hookrightarrow} Q \quad Q \equiv Q'}{P' \overset{\mathcal{R}}{\hookrightarrow} Q'}\ (\text{RSTR})$$

Figure 5.2: META✶ rewriting relation generated by $\mathcal{R}$.

treated as the same name. An instantiation $\mathbb{P}$ applies to templates componentwise. The only non-trivial case is when $\mathbb{P}$ fills in a substitution application template "$\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$". It is defined as follows.

$$\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!] = \bar{\mathbb{S}}(\mathbb{P}[\![\mathring{p}]\!])$$
$$\text{where } \mathbb{S} = \{\mathbb{P}(\mathring{x}_0) \mapsto \mathbb{P}(\mathring{s}_0), \ldots, \mathbb{P}(\mathring{x}_k) \mapsto \mathbb{P}(\mathring{s}_k)\}$$

We suppose that $\mathbb{P}[\![\mathring{P}]\!]$ is not defined when some variable from $\mathring{P}$ is not in $\mathrm{dom}(\mathbb{P})$ or when the instantiation of a substitution application construction yields an invalid substitution which is not a function. The full definition of $\mathbb{P}[\![\mathring{P}]\!]$ can be found in Figure 6.2 in Section 6.3. Additional notions concerning process templates, like the scope of a bound variable, are defined in Section 6.1.

5.2 META✶ Rewriting Relation

Given a rewriting rule set $\mathcal{R}$, Figure 5.2 defines the rewriting relation $\overset{\mathcal{R}}{\hookrightarrow}$. Rules RPAR and RNU are standard structural rules for "$\mid$" and "ν". Another standard rule RSTR incorporates structural equivalence into the rewriting relation. Rule RRW instantiates process templates inside a rule into processes using an arbitrary process instantiation $\mathbb{P}$. Example instances of RRW are given below in Section 5.3. We implicitly suppose that rule RRW is used only when both processes in the rule conclusion are properly defined. To see how RACT works let us consider the ambient active rule "$\mathbf{active}\{\,\mathring{\mathtt{P}}\ \mathbf{in}\ \mathring{\mathtt{a}}[\mathring{\mathtt{P}}]\,\}$" mentioned above. In this case rule RACT becomes

$$\frac{P \overset{\mathcal{R}}{\hookrightarrow} Q}{(\mathbb{P}(\mathring{\mathtt{a}}))\,\mathtt{[}P\mathtt{]} \overset{\mathcal{R}}{\hookrightarrow} (\mathbb{P}(\mathring{\mathtt{a}}))\,\mathtt{[}Q\mathtt{]}}$$

for an arbitrary $\mathbb{P}$ which in turn becomes equivalent to the standard Mobile Ambients rule (where $x \neq \bullet$)

$$\frac{P \overset{\mathcal{R}}{\hookrightarrow} Q}{x\,\mathtt{[}P\mathtt{]} \overset{\mathcal{R}}{\hookrightarrow} x\,\mathtt{[}Q\mathtt{]}}$$

Additional requirements which apply to rules inside $\mathcal{R}$ together with reasons for them are described in Section 6.2. Rewriting rules which satisfy the additional conditions from Section 6.2 are called *well formed* rewriting rules. These conditions are naturally satisfied by all process calculi found in the literature. Well formed rewriting rules also preserves well-formedness of processes. All the correctness results from this thesis are valid only for well formed rewriting rules. Basic properties of the rewriting relation $\overset{\mathcal{R}}{\hookrightarrow}$ are described in Section 6.4.

5.3 Example Instantiations

Now we explain META✶ instantiations on examples. The following set $\mathcal{P}_{\mathsf{sync}}$ with one rule

$$\mathcal{P}_{\mathsf{sync}} = \{\mathbf{rewrite}\{\, \mathring{c}\texttt{<}\mathring{M}\texttt{>}.\mathring{P} \mid \mathring{c}\texttt{(}\mathring{x}\texttt{)}.\mathring{Q} \hookrightarrow \mathring{P} \mid \{\mathring{x} := \mathring{M}\}\mathring{Q} \,\}\}$$

instantiates META✶ to the monadic synchronous π-calculus. *Monadic* means that only single names (that is, not tuples of names) are objects of communication and *synchronous* means that an output action can be followed by an arbitrary continuation process. A brief introduction to the π-calculus can be found in Chapter 14 of this thesis. With this only rule from $\mathcal{P}_{\mathsf{sync}}$, rule RRw becomes the following.

$$\frac{x_0 \neq \bullet \quad x_1 \neq \bullet \quad y \neq \bullet}{x_0\texttt{<}y\texttt{>}.P \mid x_0(x).Q \xrightarrow{\mathcal{P}_{\mathsf{sync}}} P \mid \overline{\{x \mapsto y\}}(Q)}\ (\text{RRw})$$

In the *asynchronous* π-calculus, output communication actions are not allowed to have any continuations at all. This can be expressed in META✶ by allowing only the null process to be the continuation of an output action as follows.

$$\mathcal{P}_{\mathsf{async}} = \{\mathbf{rewrite}\{\, \mathring{c}\texttt{<}\mathring{M}\texttt{>}.0 \mid \mathring{c}\texttt{(}\mathring{x}\texttt{)}.\mathring{Q} \hookrightarrow \{\mathring{x} := \mathring{M}\}\mathring{Q} \,\}\}$$

The following set $\mathcal{A}_{\mathsf{mon}}$ instantiates META✶ to monadic synchronous Mobile Ambients. A brief introduction to Mobile Ambients can be found in Chapter 16 of this thesis.

$$\begin{aligned}\mathcal{A}_{\mathsf{mon}} = \{\ & \mathbf{active}\{\, \mathring{P} \ \mathbf{in}\ \mathring{a}\texttt{[}\mathring{P}\texttt{]} \,\},\\ & \mathbf{rewrite}\{\, \mathring{a}\texttt{[in }\mathring{b}\texttt{.}\mathring{P} \mid \mathring{Q}\texttt{]} \mid \mathring{b}\texttt{[}\mathring{R}\texttt{]} \hookrightarrow \mathring{b}\texttt{[}\mathring{a}\texttt{[}\mathring{P} \mid \mathring{Q}\texttt{]} \mid \mathring{R}\texttt{]} \,\},\\ & \mathbf{rewrite}\{\, \mathring{a}\texttt{[}\mathring{b}\texttt{[out }\mathring{a}\texttt{.}\mathring{P} \mid \mathring{Q}\texttt{]} \mid \mathring{R}\texttt{]} \hookrightarrow \mathring{a}\texttt{[}\mathring{R}\texttt{]} \mid \mathring{b}\texttt{[}\mathring{P} \mid \mathring{Q}\texttt{]} \,\},\\ & \mathbf{rewrite}\{\, \mathsf{open}\ \mathring{a}\texttt{.}\mathring{P} \mid \mathring{a}\texttt{[}\mathring{R}\texttt{]} \hookrightarrow \mathring{P} \mid \mathring{R} \,\},\\ & \mathbf{rewrite}\{\, \texttt{<}\mathring{M}\texttt{>.}\mathring{P} \mid \texttt{(}\mathring{x}\texttt{).}\mathring{Q} \hookrightarrow \mathring{P} \mid \{\mathring{x} := \mathring{M}\}\,\mathring{Q} \,\}\ \}\end{aligned}$$

The **active** rule from this set was already described in the previous section. Also note that a message variable can be instantiated to $\bullet$ which reflects the fact that even the meaningless capabilities like "$\mathsf{in}\ (\mathsf{out}\ \mathsf{a})$" can be communicated in standard

Mobile Ambients.

Chapter 6

Technical Details on Instantiations

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

6.1 Scope of Variables in Templates

Here we introduce the notion of free and bound template variables and the notion of the scope of bound variables. These are used in next Section 6.2 to specify additional requirements on templates used in rule descriptions.

Definition 6.1.1. *When $\mathring{F}$ contains an element template "$(\mathring{x}_1, \ldots, \mathring{x}_k)$" then all occurrences of name variables $\mathring{x}_1$, ..., $\mathring{x}_k$ in "$\mathring{F}.\mathring{P}$" are said to be* **bound**. *Also name variables $\mathring{x}_0$, ..., $\mathring{x}_k$ are said to be* **bound** *in "$\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$".* ∎

Only a name variable can be bound. The set $\mathrm{fv}(\mathring{Z})$ of free variables, the set $\mathrm{bv}(\mathring{Z})$ of bound variables and the set $\mathrm{fn}(\mathring{Z})$ of free names of a template entity $\mathring{Z}$ are defined in Figure 6.1. For example, given the process template

$$\mathring{P} = \mathsf{do}(\mathring{\mathtt{y}}).\mathring{\mathtt{a}}[].(\mathsf{out}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \{\mathring{\mathtt{x}} := \mathring{\mathtt{M}}\}\ \mathring{\mathtt{Q}})$$

we have

$$\mathrm{fv}(\mathring{P}) = \{\mathring{\mathtt{a}}, \mathring{\mathtt{b}}, \mathring{\mathtt{M}}, \mathring{\mathtt{P}}, \mathring{\mathtt{Q}}\} \qquad \mathrm{bv}(\mathring{P}) = \{\mathring{\mathtt{x}}, \mathring{\mathtt{y}}\} \qquad \mathrm{fn}(\mathring{P}) = \{\mathsf{do}, [], \mathsf{out}\}$$

The sets of variables, free names, and type tags of process templates and rule descriptions are defined as follows.

Definition 6.1.2. *The set $\mathrm{fv}(\mathring{P}) \cup \mathrm{bv}(\mathring{P})$ of* **all variables** *of $\mathring{P}$ is denoted $\mathrm{var}(\mathring{P})$. The set $\mathrm{fn}(\mathcal{R})$ of* **free names** *of a rule set $\mathcal{R}$ is defined as follows.*

$$\begin{aligned}\mathrm{fn}(\mathcal{R}) = \{x : x \in \mathrm{fn}(\mathring{P}) \cup \mathrm{fn}(\mathring{Q})\ \&\ \mathbf{rewrite}\{\,\mathring{P} \hookrightarrow \mathring{Q}\,\} \in \mathcal{R}\} \cup \\ \{x : x \in \mathrm{fn}(\mathring{P})\ \&\ \mathbf{active}\{\,\mathring{p}\ \mathbf{in}\ \mathring{P}\,\} \in \mathcal{R}\}\end{aligned}$$

$\mathring{Z}$	$\mathrm{fv}(\mathring{Z})$	$\mathrm{bv}(\mathring{Z})$	$\mathrm{fn}(\mathring{Z})$
x	$\emptyset$	$\emptyset$	$\{x\}$
$\mathring{x}$	$\{\mathring{x}\}$	$\emptyset$	$\emptyset$
$(\mathring{x}_1, \ldots, \mathring{x}_k)$	$\emptyset$	$\{\mathring{x}_1, \ldots, \mathring{x}_k\}$	$\emptyset$
$\texttt{<}\mathring{m}_1, \ldots, \mathring{m}_k\texttt{>}$	$\{\mathring{m}_1, \ldots, \mathring{m}_k\}$	$\emptyset$	$\emptyset$
$\mathring{E}_0 \ldots \mathring{E}_k$	$\bigcup_{i=0}^{k} \mathrm{fv}(\mathring{E}_i)$	$\bigcup_{i=0}^{k} \mathrm{bv}(\mathring{E}_i)$	$\bigcup_{i=0}^{k} \mathrm{fn}(\mathring{E}_i)$
$\texttt{0}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathring{p}$	$\{\mathring{p}\}$	$\emptyset$	$\emptyset$
$\mathring{F}.\mathring{P}$	$\mathrm{fv}(\mathring{F}) \cup (\mathrm{fv}(\mathring{P}) \backslash \mathrm{bv}(\mathring{F}))$	$\mathrm{bv}(\mathring{F}) \cup \mathrm{bv}(\mathring{P})$	$\mathrm{fn}(\mathring{F}) \cup \mathrm{fn}(\mathring{P})$
$\mathring{P} \mid \mathring{Q}$	$\mathrm{fv}(\mathring{P}) \cup \mathrm{fv}(\mathring{Q})$	$\mathrm{bv}(\mathring{P}) \cup \mathrm{bv}(\mathring{Q})$	$\mathrm{fn}(\mathring{P}) \cup \mathrm{fn}(\mathring{Q})$
$\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$	$\{\mathring{s}_1, \ldots, \mathring{s}_k, \mathring{p}\}$	$\{\mathring{x}_1, \ldots, \mathring{x}_k\}$	$\emptyset$

Figure 6.1: Free names and free/bound variables of META✶ template entities.

Let $\mathrm{tags}(\mathring{P}) = \overline{\mathrm{fn}(\mathring{P})}$ *and* $\mathrm{tags}(\mathcal{R}) = \overline{\mathrm{fn}(\mathcal{R})}$. ▪

The following definition defines the notion of the scope of a bound name variable and some useful relations.

DEFINITION 6.1.3. *We say that an occurrence of* $\mathring{z}$ *in* $\mathring{P}$ *is* **under the scope** *of* $\mathring{x}$ *when* $\mathring{P}$ *contains either*

(U1) $\mathring{F}.\mathring{Q}$ *with* $\mathring{x} \in \mathrm{bv}(\mathring{F})$ *and with the given occurrence of* $\mathring{z}$ *in* $\mathring{Q}$*, or*

(U2) $\{\ldots \mathring{x} := \mathring{s} \ldots\}\,\mathring{p}$ *with* $\mathring{p} = \mathring{z}$ *being the given occurrence of* $\mathring{z}$.

Write $\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{z}$ *when there is an occurrence of* $\mathring{z}$ *in* $\mathring{P}$ *under the scope of* $\mathring{x}$.
Write $\mathring{P} \vdash_{\forall} \mathring{x} \succ \mathring{z}$ *when all occurrences of* $\mathring{z}$ *in* $\mathring{P}$ *are under the scope of* $\mathring{x}$. ▪

Note that $\mathring{z}$ can be a bound variable and thus, for example, "$\texttt{(}\mathring{\texttt{x}}\texttt{).(}\mathring{\texttt{y}}\texttt{).0} \vdash_{\exists} \mathring{\texttt{x}} \succ \mathring{\texttt{y}}$".

6.2 Additional Requirements on Rewriting Rules

It is desirable to forbid rules and inferences that would cause a name capture, release of a bound name, unleash a nested input-binders, or that would introduce a nesting of previously not nested input-binders. To ensure that the aboves do not happen we need additional syntactic restrictions on rewriting rules. This section describes these conditions and their purpose. In the previous META✶ [MW05, MW04a] these conditions were stated only informally which was not found satisfactory to carry out the proofs presented in this thesis. There were also some inadequacies, for example, the rule "**rewrite**$\texttt{\{ (}\mathring{\texttt{x}}\texttt{).0} \hookrightarrow \mathring{\texttt{x}}\texttt{.0 \}}$" that can produce a non-well formed (scoped) process was accidentally allowed.

The following defines additional restrictions that apply to the left-hand side template in a rewriting rule. The purpose of these conditions is explained in the consequent Remark 6.2.2

DEFINITION 6.2.1. *We say that* $\mathring{P}$ *is a* **well formed lhs-template** *when* $\mathring{P}$ *satisfies the following properties.*

(L1) $\mathrm{tags}(\mathring{P}) \subseteq \mathrm{SpecialTag}$

(L2) $\mathrm{fv}(\mathring{P}) \cap \mathrm{bv}(\mathring{P}) = \emptyset$

(L3) *any message and process variable occurs at most once in* $\mathring{P}$

(L4) *every* $\mathring{x} \in \mathrm{bv}(\mathring{P})$ *occurs exactly once in* $\mathring{P}$

(L5) *when* $\mathring{P} \vdash_{\exists} \mathring{x} \gtrdot \mathring{z}$ *then* $\mathring{P} \vdash_{\forall} \mathring{x} \gtrdot \mathring{z}$

(L6) $\mathring{P}$ *does not contain* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$ ∎

REMARK 6.2.2. Condition L1 allows only type tags from SpecialTag to be used as tags of free names specifically mentioned in $\mathring{P}$. The reason for this was explained in Section 3.2 and it is further discussed in Remark 6.2.4 below. Condition L2 prevents mixing of free and bound name variables. A process template that does not satisfy L2 would not instantiate to a well formed process. Condition L3 says that a reduction rule can not depend on the fact that two entire messages or processes are identical. On the other hand, a reduction rule can depend on the fact that the same single name occurs in a different positions in a process. There are several reasons for condition L3. Mainly, comparing of whole messages or processes is not necessary for describing rewriting rules of process calculi found in the literature. Moreover implementation of this comparison would be time expansive. Condition L4 firstly disallows form templates like "$(\mathring{\mathtt{x}}, \mathring{\mathtt{x}})$" which would not instantiate to a well formed process. Secondly, it forbids templates like "$(\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \mid (\mathring{\mathtt{x}}).\mathring{\mathtt{Q}}$" on the left-hand side of a rule because the right-hand side would then be able to construct a process like "$(\mathring{\mathtt{x}}).(\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}})$" thus causing a scope mixture. Condition L5 says that whenever some variable $\mathring{z}$ occurs under the scope of some $\mathring{x}$ then all other occurrences of $\mathring{z}$ have to be found under the scope of the same $\mathring{x}$ as well. For example, the template "$(\mathring{\mathtt{x}}).\mathring{\mathtt{a}}.\mathtt{0} \mid (\mathring{\mathtt{y}}).\mathring{\mathtt{a}}.\mathtt{0}$" is banned in order to avoid possible name captures. Note that $\mathring{P} \vdash_{\forall} \mathring{x} \gtrdot \mathring{z}$ holds when $\mathring{z}$ does not occur in $\mathring{P}$ at all and thus the opposite implication of L5 is not required. Finally, condition L6 forbids the use of the substitution application construction on the left-hand side of a rule. This construction is intended to be part of the right-hand side of a rule only. Implementation of rules with substitution on the left-hand side would be complicated and it is not necessary for our intentions. ∎

Similarly the following restrictions apply to the right-hand side template in a rewriting rule. The well-formedness of the right-hand side template depends on the corresponding left-hand side from the rewriting rule. This is because we need , for example, to ensure that the right-hand side of some rule does not invent a variable that is not mentioned by the rule left hand side.

DEFINITION 6.2.3. *We say that $\mathring{Q}$ is a* **well formed rhs-template** *w.r.t. a well formed lhs-template $\mathring{P}$ when $\mathring{Q}$ satisfies the following properties.*

(R1) $\mathrm{tags}(\mathring{Q}) \subseteq \mathrm{SpecialTag}$

(R2) $\mathrm{fv}(\mathring{Q}) \subseteq \mathrm{fv}(\mathring{P})$

(R3) $\mathrm{bv}(\mathring{Q}) \subseteq \mathrm{bv}(\mathring{P})$

(R4) *every $\mathring{x} \in \mathrm{bv}(\mathring{Q})$ occurs exactly once in $\mathring{Q}$*

(R5) *when $\mathring{Q} \vdash_\exists \mathring{x} \gtrdot \mathring{z}$ then $\mathring{Q} \vdash_\forall \mathring{x} \gtrdot \mathring{z}$*

(R6) *for $\mathring{z} \in \mathrm{var}(\mathring{Q})$ and any $\mathring{x}$ holds that $\mathring{P} \vdash_\forall \mathring{x} \gtrdot \mathring{z}$ iff $\mathring{Q} \vdash_\forall \mathring{x} \gtrdot \mathring{z}$*

(R7) *when $\{\mathring{x}, \mathring{y}\} \subseteq \mathrm{bv}(\mathring{F}_1)$ for $\mathring{F}_1$ in $\mathring{Q}$ then $\{\mathring{x}, \mathring{y}\} \subseteq \mathrm{bv}(\mathring{F}_0)$ for some $\mathring{F}_0$ in $\mathring{P}$* ∎

REMARK 6.2.4. Condition R1 ensures that the right-hand side of some rule does not invent a name with some type tag that is not in SpecialTag. Consider two names a and b which are not in SpecialTag and the following non-well formed rule.

$$\mathcal{R} = \{\mathbf{rewrite}\{\, \mathsf{a.0} \hookrightarrow \mathsf{b.0} \,\}\}$$

Using $\mathcal{R}$ we can prove "$\mathsf{a.0} \overset{\mathcal{R}}{\hookrightarrow} \mathsf{b.0}$". From this we prove inference with a name capture, like "$\nu\mathsf{b.a.0} \overset{\mathcal{R}}{\hookrightarrow} \nu\mathsf{b.b.0}$", or with a name release, like "$\nu\mathsf{a.a.0} \overset{\mathcal{R}}{\hookrightarrow} \nu\mathsf{a.b.0}$". That is why the above rule has to be forbidden as long as a and b are not in SpecialTag. When a and b are in SpecialTag then condition W4 ensures that the processes participating in the above problematic inferences are not well formed.

Conditions R2 and R3 ensures that the right-hand side does not contain a variable that is not mentioned on the left-hand side and that it does not mix free and bound variables. Mixing of free and bound variables might cause name capture or name release. Note that R1, R2, and L2 implies that $\mathrm{fv}(\mathring{Q}) \cap \mathrm{bv}(\mathring{Q}) = \emptyset$. Condition R4 forbids creation of a non-well formed process using form templates like "$(\mathring{\mathtt{x}}, \mathring{\mathtt{x}})$". It also forbids rewriting rules like

$$\mathbf{rewrite}\{\, (\mathring{\mathtt{y}}).\mathring{\mathtt{y}}.(\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \hookrightarrow (\mathring{\mathtt{y}}).\{\mathring{\mathtt{x}} := \mathring{\mathtt{y}}\}\mathring{\mathtt{P}} \,\}$$

where the bound variable $\mathring{\mathtt{y}}$ is used to form a substitution which is applied under the binder of $\mathring{\mathtt{y}}$. Rewriting rules like this could probably be allowed but here we prefer a more restrictive condition to simplify proofs because rules like the above are not required to describe process calculi from the literature. Condition R4 also forbids rules with right-hand sides like "$(\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \mid (\mathring{\mathtt{x}}).\mathring{\mathtt{P}}$" which can again probably be allowed (with an appropriate left-hand side) but we do not find it necessary.

Condition R5 ensures that every variable occurs under the scope of the same binders thus preventing name captures and name releases. For example "$\mathring{\mathtt{P}} \mid (\mathring{\mathtt{x}}).\mathring{\mathtt{P}}$" is not a well formed rhs-template for any lhs-template because there is not guarantee

that (the instantiation of) $\mathring{\mathtt{P}}$ does not contain a free occurrence of (the instantiation of) $\mathring{\mathtt{x}}$. Condition R6 is the main condition that prevents name captures and name releases. It says that every variable has to occur under the same scopes on both sides of a rule. For example, the following rules are banned

$$\begin{aligned}&\textbf{rewrite}\{\ (\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \hookrightarrow \mathring{\mathtt{P}}\ \}\\&\textbf{rewrite}\{\ \mathring{\mathtt{P}} \mid (\mathring{\mathtt{x}}).\mathring{\mathtt{Q}} \hookrightarrow (\mathring{\mathtt{x}}).\mathring{\mathtt{P}}\ \}\\&\textbf{rewrite}\{\ (\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \mid (\mathring{\mathtt{y}}).\mathring{\mathtt{Q}} \hookrightarrow (\mathring{\mathtt{x}},\mathring{\mathtt{y}}).(\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}})\ \}\end{aligned}$$

because they could cause name capture or name release, or they could produce a non-well formed process. The equivalence in condition R6 is required to hold only when $\mathring{z} \in \mathrm{var}(\mathring{Q})$ because the right-hand side is allowed to forget some variable $\mathring{z}$ that is mentioned by the left-hand side and then $\mathring{Q} \vdash_\forall \mathring{x} \gg \mathring{z}$ would trivially hold for any $\mathring{x}$. To simplify proofs, condition R6 is again little bit more restrictive than necessary and thus, for example, the following rules are forbidden but they could probably be allowed.

$$\begin{aligned}&\textbf{rewrite}\{\ (\mathring{\mathtt{x}},\mathring{\mathtt{y}}).\mathring{\mathtt{P}} \hookrightarrow (\mathring{\mathtt{x}}).(\mathring{\mathtt{y}}).\mathring{\mathtt{P}})\ \}\\&\textbf{rewrite}\{\ (\mathring{\mathtt{x}}).(\mathring{\mathtt{y}}).\mathring{\mathtt{P}} \hookrightarrow (\mathring{\mathtt{x}},\mathring{\mathtt{y}}).\mathring{\mathtt{P}}\ \}\end{aligned}$$

Note that condition R6 does not forbid the rule

$$\textbf{rewrite}\{\ (\mathring{\mathtt{x}}).0 \mid (\mathring{\mathtt{y}}).0 \hookrightarrow (\mathring{\mathtt{x}},\mathring{\mathtt{y}}).0\ \}$$

which can produce a non-well scoped process and thus has to be forbidden. The problem here is that there is no variable under the scope of binders which would allow us to apply R6. Condition R7 is introduced specifically to solve this problem. It says that bound names on the right-hand side which are inside a single form template have to come from a single form template on the left-hand side. The only purpose of condition R7 is that a non-well formed process which would violate W3 is not constructed. ∎

The following defines well formed rewriting rule sets.

Definition 6.2.5. *The rule* $\textbf{rewrite}\{\ \mathring{P} \hookrightarrow \mathring{Q}\ \}$ *is said to be* **well formed** *when* $\mathring{P}$ *is a well formed lhs-template and* $\mathring{Q}$ *is a well formed rhs-template w.r.t.* $\mathring{P}$*. The rule* $\textbf{active}\{\ \mathring{p}\ \textbf{in}\ \mathring{P}\ \}$ *is said to be* **well formed** *when* $\mathring{P}$ *is a well formed lhs-template and* $\mathring{p} \in \mathrm{var}(\mathring{P})$*. The rule set* $\mathcal{R}$ *is called a* **well formed rule set**, *when all its rules are well formed.* ∎

The following defines *well lhs-formed* element and form templates which are templates that can legally occur as a part of some well formed lhs-template. Note that, for example, an element template "$\mathtt{<}\mathring{\mathtt{M}},\mathring{\mathtt{M}}\mathtt{>}$" can never occur as a part of a well

$$
\begin{array}{ll}
\mathbb{P}[\![x]\!] = x & \mathbb{P}[\![(\mathring{x}_1, \ldots, \mathring{x}_k)]\!] = (\mathbb{P}(\mathring{x}_1), \ldots, \mathbb{P}(\mathring{x}_k)) \\
\mathbb{P}[\![\mathring{x}]\!] = \mathbb{P}(\mathring{x}) & \mathbb{P}[\![\texttt{<}\mathring{m}_1, \ldots, \mathring{m}_k\texttt{>}]\!] = \texttt{<}\mathbb{P}(\mathring{m}_1), \ldots, \mathbb{P}(\mathring{m}_k)\texttt{>} \\
\mathbb{P}[\![\mathring{m}]\!] = \mathbb{P}(\mathring{m}) & \mathbb{P}[\![\mathring{E}_0 \ldots \mathring{E}_k]\!] = \mathbb{P}[\![\mathring{E}_0]\!] \ldots \mathbb{P}[\![\mathring{E}_k]\!] \\
\mathbb{P}[\![\texttt{0}]\!] = \texttt{0} & \mathbb{P}[\![\mathring{F}.\mathring{P}]\!] = \mathbb{P}[\![\mathring{F}]\!].\mathbb{P}[\![\mathring{P}]\!] \\
\mathbb{P}[\![\mathring{p}]\!] = \mathbb{P}(\mathring{p}) & \mathbb{P}[\![\mathring{P} \mid \mathring{Q}]\!] = \mathbb{P}[\![\mathring{P}]\!] \mid \mathbb{P}[\![\mathring{Q}]\!]
\end{array}
$$

$$\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!] = \bar{\mathbb{S}}(\mathbb{P}[\![\mathring{p}]\!]) \text{ where } \mathbb{S} = \{(\mathbb{P}(\mathring{x}_i) \mapsto \mathbb{P}(\mathring{s}_i)) \colon i \in \{0, \ldots, k\}\}$$

Figure 6.2: Instantiation of META✶ templates.

form lhs-template because it violates L3. It can, however, legally occur on the right side of some rule. These two notions are used later in the proofs in Chapter 12.

DEFINITION 6.2.6. *An element template $\mathring{E}$ (resp. form template $\mathring{F}$) is* **well lhs-formed** *when $\mathring{E}.\texttt{0}$ (resp. $\mathring{F}.\texttt{0}$) is a well formed lhs-template.* ∎

6.3 Properties of Process Instantiations

The full definition of application $\mathbb{P}[\![\mathring{Z}]\!]$ of a process instantiation $\mathbb{P}$ to various META✶ process entities $\mathring{Z}$ is defined Figure 6.2. Next, we prove some properties of well formed templates and some properties of template instantiations which are to be used later by various proofs.

The following lemma says that, in well formed rewriting rules, all the variables bound by a substitution application construction on the right-hand side of a rule have to come from a single form template on the left-hand side.

LEMMA 6.3.1. *Let* **rewrite**$\{\,\mathring{P} \hookrightarrow \mathring{Q}\,\}$ *be well formed. When $\mathring{Q}$ contains $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}$ then there is $\mathring{F}$ in $\mathring{P}$ such that $\{\mathring{x}_0, \ldots, \mathring{x}_k\} \subseteq \mathrm{bv}(\mathring{F})$.*

PROOF. *Let $\mathring{Q}$ contains $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$. By* R2 *and* R3 *we obtain that $\mathring{p} \in \mathrm{fv}(\mathring{P})$ and $\{\mathring{x}_0, \ldots, \mathring{x}_k\} \subseteq \mathrm{bv}(\mathring{P})$. We see that $\mathring{Q} \vdash_\exists \mathring{x}_i \gtrdot \mathring{p}$ for all $i \in \{0, \ldots, k\}$. Thus by* R5 *and* R6 *and by above we obtain $\mathring{P} \vdash_\exists \mathring{x}_i \gtrdot \mathring{p}$ for all $i \in \{0, \ldots, k\}$. Now when some $\mathring{x}_i$ and $\mathring{x}_j$ does not occur in the same form template then we have either $\mathring{P} \vdash_\exists \mathring{x}_i \gtrdot \mathring{x}_j$ or $\mathring{P} \vdash_\exists \mathring{x}_j \gtrdot \mathring{x}_i$ because $\mathring{p}$ occurs in $\mathring{P}$ under the scope of both $\mathring{x}_i$ and $\mathring{x}_j$. But this leads to a contradiction because neither $\mathring{Q} \vdash_\forall \mathring{x}_i \gtrdot \mathring{x}_j$ nor $\mathring{Q} \vdash_\forall \mathring{x}_j \gtrdot \mathring{x}_i$ which are required by* L5 *and* R6 *holds. Hence $\mathring{x}_i$ and $\mathring{x}_j$ has to occur in the same form template.* ∎

Let us suppose that "$\mathring{P} \vdash_\exists \mathring{x} \gtrdot \mathring{p}$", that is, that $\mathring{p}$ occurs in $\mathring{P}$ under the scope of some bound variable $\mathring{x}$. The following lemma says that when $\mathring{x}$ is instantiated to some name a^ι then type tag ι can not be input-bound in (the instantiation of) $\mathring{p}$ as long as $\mathring{P}$ is (instantiated to) a well formed process.

LEMMA 6.3.2. *Let $\mathring{P}$ be a well formed lhs-template such that $\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{p}$. Let $\mathbb{P}[\![\mathring{P}]\!]$ be defined and well formed. Then $\overline{\mathbb{P}[\![\mathring{x}]\!]} \notin \mathrm{itags}(\mathbb{P}[\![\mathring{p}]\!])$.*

PROOF. *Let $x = \mathbb{P}[\![\mathring{x}]\!]$ and $P = \mathbb{P}[\![\mathring{P}]\!]$ and $P_0 = \mathbb{P}[\![\mathring{p}]\!]$. Now, because $\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{p}$ and P is defined, there is F with $\overline{x} \in \mathrm{itags}(F)$ and there is P_1 such that P has a subprocess $F.P_1$ and P_1 has a subprocess P_0. Thus $\overline{x} \in \mathrm{itags}(F.P_1)$. Now $F.P_1$ is well formed because P is well formed. Thus $\overline{x} \notin \mathrm{itags}(P_1)$ by* W2 *for $F.P_1$. Hence the claim because* $\mathrm{itags}(P_0) \subseteq \mathrm{itags}(P_1)$. ∎

The following remark and lemma describes a property of process instantiations closely related to the one described by the previous lemma.

REMARK 6.3.3. Consider again the situation from the previous lemma when a well formed lhs-template contains a process variable $\mathring{p}$ under the scope of some input bound variable $\mathring{x}$ ($\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{p}$) which is instantiated to some name x with type tag ι. The following lemma says that when (the instantiation of) $\mathring{p}$ contains a free name y with type tag ι then it has to hold that $x = y$. An instantiation without this property would not instantiate the template to a well formed process. To clarify this issue let us consider the following template instantiation $\mathbb{P}$ which instantiates $\mathring{P}$ to a non-well formed process.

$$\mathring{P} = (\mathring{\mathtt{x}}).\mathring{\mathtt{P}} \qquad \mathbb{P} = \{\mathring{\mathtt{x}} \mapsto \mathtt{a}^{\mathtt{a}}, \mathring{\mathtt{P}} \mapsto \mathtt{<z^a>.0}\}$$

We see that "$\mathbb{P}[\![\mathring{P}]\!] = \mathtt{(a^a).<z^a>.0}$" violates W1. Suppose that we have the above $\mathring{P}$ and some $\mathbb{P}'$ such that $\mathbb{P}'(\mathring{\mathtt{x}}) = a^{\iota}$. Then the following lemma says that, when $\mathbb{P}'[\![\mathring{P}]\!]$ is well formed then the only free name with type tag ι in $\mathbb{P}'(\mathring{\mathtt{P}})$ is a^{ι}. This property will be important later for the subject reduction property and it is further discussed Remark 8.2.1 and Remark 8.7.1. ∎

LEMMA 6.3.4. *Let $\mathring{P}$ be a well formed lhs-template such that $\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{p}$. Let $\mathbb{P}[\![\mathring{P}]\!]$ be defined and well formed and let $y \in \mathrm{fn}(\mathbb{P}[\![\mathring{p}]\!])$. Then $\overline{\mathbb{P}[\![\mathring{x}]\!]} = \overline{y}$ implies $\mathbb{P}[\![\mathring{x}]\!] = y$.*

PROOF. *Let $x = \mathbb{P}[\![\mathring{x}]\!]$ and $P = \mathbb{P}[\![\mathring{P}]\!]$ and $P_0 = \mathbb{P}[\![\mathring{p}]\!]$. Now, because $\mathring{P} \vdash_{\exists} \mathring{x} \succ \mathring{p}$ and P is defined, there is F with $x \in \mathrm{bv}(F)$ and there is P_1 such that P has a subprocess $F.P_1$ and P_1 has a subprocess P_0. Thus $\overline{x} \in \mathrm{itags}(F.P_1)$. Take $y \in \mathrm{fn}(P_0)$ such that $\overline{x} = \overline{y}$. To prove the lemma we need to show $x = y$. Now $F.P_1$ is well formed because P is well formed. Thus $\overline{y} \notin \mathrm{ftags}(F.P_0)$ as well as $\overline{y} \notin \mathrm{ntags}(F.P_0)$ because tags of free and ν-bound names do not overlap with tags of input-bound names by* W1 *for $F.P_0$. Thus y is input-bound in $F.P_0$. Moreover by* W2 *for $F.P_0$ we see that $\overline{y} \notin \mathrm{itags}(P_0)$. Thus the only possibility is that $y \in \mathrm{fn}(P_0)$ because $\overline{y} \in \mathrm{ntags}(P_0)$ would imply $\overline{y} \in \mathrm{ntags}(F.P_0)$. But now it has to be $y \in \mathrm{bv}(F)$ because $\overline{y} \in \mathrm{itags}(F.P_0)$. Thus $x = y$ by* W3 *for F and $F.P_0$.* ∎

6.4 Properties of META✶ Rewriting Relation

In this section we prove some basic properties of META✶ rewriting relation, mainly that $\overset{\mathcal{R}}{\hookrightarrow}$ preserves well-formedness provided $\mathcal{R}$ is well formed. Although this property is not necessarily required for subject reduction it is a desirable property, for example, because it makes the system more intuitive and its behavior more expectable. Without this property we might, for example, expect different behavior of processes which differ only by renaming of names

The following lemma is the first step to prove that the rewriting relation preserves well-formedness of processes. It states that the rewriting relation can not invent new bound tags and that it can introduce only names from $\mathrm{fn}(\mathcal{R})$ or $\bullet$. The proof is technical and it mainly uses conditions L1-6 and R1-7. Below we do not suppose that Q is well formed because we want to use this lemma to prove its well-formedness.

LEMMA 6.4.1. *Let P and $\mathcal{R}$ be well formed. Then $P \overset{\mathcal{R}}{\hookrightarrow} Q$ implies*

(1) $\mathrm{itags}(Q) \subseteq \mathrm{itags}(P)$,

(2) $\mathrm{ntags}(Q) \subseteq \mathrm{ntags}(P)$, *and*

(3) $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$.

PROOF. *By induction on the derivation of $P \overset{\mathcal{R}}{\hookrightarrow} Q$. Let $P \overset{\mathcal{R}}{\hookrightarrow} Q$ be derived by*

(RRW)**:** *There is some* **rewrite**$\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *and there is some $\mathbb{P}$ such that $P = \mathbb{P}[\![\mathring{P}]\!]$ and $Q = \mathbb{P}[\![\mathring{Q}]\!]$. Let us prove the three claims separately.*

(1) Let $\iota \in \mathrm{itags}(Q)$. At least one of the following cases applies.

(a) The input-binder of ι in Q comes from some leaf of template $\mathring{Q}$ which is a standalone process variable $\mathring{p}$ (that is, not a substitution operator). Thus we have $\mathring{p} \in \mathrm{fv}(\mathring{Q})$ and $\iota \in \mathrm{itags}(\mathbb{P}(\mathring{p}))$. By R3 *we obtain $\mathring{x} \in \mathrm{fv}(\mathring{P})$. Hence $\iota \in \mathrm{itags}(P)$.*

(b) The input-binder of ι in Q comes from some leaf of template $\mathring{Q}$ which is a substitution operator $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$. In this case we have $\iota \in \mathrm{itags}(\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!])$ and $\mathring{p} \in \mathrm{fv}(\mathring{Q})$. Let $\mathbb{S} = \{\mathbb{P}[\![\mathring{x}_0]\!] \mapsto \mathbb{P}[\![\mathring{s}_0]\!], \ldots \mathbb{P}[\![\mathring{x}_k]\!] \mapsto \mathbb{P}[\![\mathring{s}_k]\!]\}$. Hence we have $\iota \in \mathrm{itags}(\bar{\mathbb{S}}(\mathbb{P}(\mathring{p})))$. Thus we obtain $\iota \in \mathrm{itags}(\mathbb{P}(\mathring{p}))$ by Lemma 4.6.2. From $\mathring{p} \in \mathrm{fv}(\mathring{Q})$ we obtain $\mathring{p} \in \mathrm{fv}(\mathring{P})$ by R2. *Hence $\iota \in \mathrm{itags}(P)$.*

(c) The input-binder of ι in Q comes from some form template in template $\mathring{Q}$. Thus there is some $\mathring{x} \in \mathrm{bv}(\mathring{Q})$ such that $\overline{\mathbb{P}(\mathring{x})} = \iota$. By R3 *we obtain $\mathring{x} \in \mathrm{bv}(\mathring{P})$. Hence $\iota \in \mathrm{itags}(P)$.*

(2) Let $\iota \in \mathrm{ntags}(Q)$. At least one of the following cases applies.

(a) The ν-binder of ι in Q comes from some leaf of template $\mathring{Q}$ which is a standalone process variable $\mathring{p}$ (that is not a substitution operator). In

this case we have $\iota \in \mathrm{ntags}(\mathbb{P}(\mathring{p}))$ *and* $\mathring{p} \in \mathrm{fv}(\mathring{Q})$. *We obtain* $\mathring{p} \in \mathrm{fv}(\mathring{P})$ *by* R2. *Hence* $\iota \in \mathrm{ntags}(P)$.

(b) *The* ν*-binder of* ι *in* Q *comes from some leaf of template* $\mathring{Q}$ *which is a substitution operator* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$. *In this case we have* $\iota \in \mathrm{ntags}(\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!])$ *and* $\mathring{p} \in \mathrm{fv}(\mathring{Q})$. *Let* $\mathbb{S} = \{\mathbb{P}[\![\mathring{x}_0]\!] \mapsto \mathbb{P}[\![\mathring{s}_0]\!], \ldots \mathbb{P}[\![\mathring{x}_k]\!] \mapsto \mathbb{P}[\![\mathring{s}_k]\!]\}$. *Hence we have* $\iota \in \mathrm{ntags}(\bar{\mathbb{S}}(\mathbb{P}(\mathring{p})))$. *Thus we obtain* $\iota \in \mathrm{ntags}(\mathbb{P}(\mathring{p}))$ *by Lemma 4.6.2. From* $\mathring{p} \in \mathrm{fv}(\mathring{Q})$ *we obtain* $\mathring{p} \in \mathrm{fv}(\mathring{P})$ *by* R2. *Hence* $\iota \in \mathrm{ntags}(P)$.

(3) *Let* $x \in \mathrm{fn}(Q)$. *At least one of the following cases applies.*

(a) *It is* $x \in \mathrm{fn}(\mathring{Q})$. *Thus clearly* $x \in \mathrm{fn}(\mathcal{R})$ *and the claim holds.*

(b) *The occurrence of* x *that contributes to* $\mathrm{fn}(Q)$ *comes from the value of some variable* $\mathring{z}$ *in template* $\mathring{P}$ *which is not under the substitution operator. Clearly it has to be* $\mathring{z} \in \mathrm{fv}(\mathring{Q})$ *(because* R2, R3, *and* L2*). We have* $x \in \mathrm{fn}(\mathbb{P}(\mathring{z}))$ *We can prove that* $\mathring{Q} \vdash_\exists \mathring{y} \succ \mathring{z}$ *implies* $\mathbb{P}(\mathring{y}) \neq x$ *for any* $\mathring{y}$ *because the occurrence of* x *in* $\mathbb{P}(\mathring{z})$ *contributes to* $\mathrm{fn}(Q)$ *and thus can not occur under a binder which bind* x. *Rule* R5 *ensures that this property is satisfied for all occurrences of* $\mathring{z}$ *in* $\mathring{P}$. *Using* R6 *we prove that* $\mathring{P} \vdash_\exists \mathring{y} \succ \mathring{z}$ *implies* $\mathbb{P}(\mathring{y}) \neq x$ *for any* $\mathring{y}$ *as well. Thus* $x \in \mathrm{fn}(\mathbb{P}(\mathring{z}))$ *implies* $x \in \mathrm{fn}(P)$. *Hence the claim.*

(c) *The occurrence of* x *that contributes to* $\mathrm{fn}(Q)$ *comes from some leaf of template* $\mathring{Q}$ *which is a substitution operator* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$. *In this case we have* $x \in \mathrm{fn}(\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!])$ *and* $\mathring{p} \in \mathrm{fv}(\mathring{Q})$. *Let* $\mathbb{S} = \{\mathbb{P}[\![\mathring{x}_0]\!] \mapsto \mathbb{P}[\![\mathring{s}_0]\!], \ldots \mathbb{P}[\![\mathring{x}_k]\!] \mapsto \mathbb{P}[\![\mathring{s}_k]\!]\}$. *Hence we have* $x \in \mathrm{fn}(\bar{\mathbb{S}}(\mathbb{P}(\mathring{p})))$. *Thus we obtain* $x \in \mathrm{fn}(\mathbb{P}(\mathring{p})) \cup \mathrm{fn}(\mathbb{S}) \cup \{\bullet\}$ *by Lemma 4.6.2.*

i. *Let* $x \in \mathrm{fn}(\mathbb{P}(\mathring{p}))$. *We shall prove that the (only) occurrence of* $\mathring{p}$ *in* $\mathring{P}$ *is not under the scope of any binder which binds* x. *Let* $\mathring{P} \vdash_\exists \mathring{y} \succ \mathring{p}$. *Thus* $\mathring{Q} \vdash_\forall \mathring{y} \succ \mathring{p}$ *by* L5 *and* R6. *We need to prove that* $\mathbb{P}[\![\mathring{y}]\!] \neq x$. *There are two possibilities. Firstly, when* $\mathring{y} = \mathring{x}_i$ *for some* $i \in \{0, \ldots, k\}$. *We know that* $x \in \mathrm{fn}(\bar{\mathbb{S}}(\mathbb{P}(\mathring{p})))$ *and thus it has to be* $x \notin \mathrm{dom}(\mathbb{S})$. *Hence* $\mathbb{P}(\mathring{y}) = \mathbb{P}(\mathring{x}_i) \neq x$. *Secondly, when* $\mathring{y} \neq \mathring{x}_i$ *for any* $i \in \{0, \ldots, k\}$. *Then the whole substitution operator* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$ *in* $\mathring{Q}$ *is under the scope of* $\mathring{y}$ *and thus clearly* $\mathbb{P}(\mathring{y}) \neq x$ *because the occurrence of* x *in* $\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}]\!]$ *contributes to* $\mathrm{fn}(Q)$. *Hence* $x \in \mathrm{fn}(P)$.

ii. *Let* $x \in \mathrm{fn}(\mathbb{S})$. *Hence there is some* $i \in \{0, \ldots, k\}$ *such that* $x \in \mathrm{fn}(\mathbb{P}(\mathring{s}_i))$. *Rule* R4 *implies that* $\mathring{s}_i \in \mathrm{fv}(\mathring{Q})$ *(because if* $\mathring{s}_i$ *was bound in* $\mathring{Q}$ *then there would have to be a second occurrence of* $\mathring{s}_i$ *in* $\mathring{Q}$ *inside some binder which is forbidden by* R4*). Now*

when $\mathring{P} \vdash_\exists \mathring{y} \gg \mathring{s}_i$ *then we have* $\mathring{Q} \vdash_\forall \mathring{y} \gg \mathring{s}_i$ *by* L5 *and* R6. *But then* $\mathbb{P}(\mathring{y}) \neq x$ *because we know that an occurrence of* x *in* $\mathbb{P}[\![\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\, \mathring{p}]\!]$ *contributes to* $\mathrm{fn}(Q)$ *and thus can occur under a binder that binds* x. *Thus we have proved that any occurrence of* $\mathring{s}_i$ *in* $\mathring{P}$ *is not under a binder that binds (after instantiation)* x. *Thus* $x \in \mathrm{fn}(P)$. *Hence the claim.*

iii. Let $x = \bullet$. *Then the claim* $x \in \mathrm{fn}(P) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$ *clearly holds.*

(RAct)**:** *There is some* **active**$\{\, \mathring{p}$ **in** $\mathring{P}\, \} \in \mathcal{R}$ *and there are some* P_0, Q_0, *and* $\mathbb{P}$ *such that* $P = (\mathbb{P}[\mathring{p} \mapsto P_0])[\![\mathring{P}]\!]$ *and* $Q = (\mathbb{P}[\mathring{p} \mapsto Q_0])[\![\mathring{P}]\!]$ *and* $P_0 \xhookrightarrow{\mathcal{R}} Q_0$. *Let* $\mathbb{P}^P = \mathbb{P}[\mathring{p} \mapsto P_0]$ *and* $\mathbb{P}^Q = \mathbb{P}[\mathring{p} \mapsto Q_0]$. *That is, we have* $P = \mathbb{P}^P[\![\mathring{P}]\!]$ *and* $Q = \mathbb{P}^Q[\![\mathring{P}]\!]$. *Let us prove the three claims separately.*

(1) Let $\iota \in \mathrm{itags}(Q)$. *At least one of the following two cases applies.*

(a) The input-binder of ι *in* Q *comes from some leaf of template* $\mathring{P}$ *which has to be a process variable* $\mathring{p}_0$ *by* L6. *Thus* $\iota \in \mathrm{itags}(\mathbb{P}^Q(\mathring{p}_0))$ *and* $\mathring{p}_0 \in \mathrm{fv}(\mathring{P})$. *Now it is easy to prove* $\iota \in \mathrm{itags}(\mathbb{P}^P(\mathring{p}_0))$ *using the induction hypothesis when* $\mathring{p} = \mathring{p}_0$. *Hence the claim* $\iota \in \mathrm{itags}(P)$ *because* $\mathring{p}_0 \in \mathrm{fv}(\mathring{P})$.

(b) The input-binder of ι *in* Q *comes from some form template in* $\mathring{P}$. *In this case there is some name variable* $\mathring{x} \in \mathrm{bv}(\mathring{P})$ *such that* $\overline{\mathbb{P}(\mathring{x})} = \iota$. *Clearly* $\mathbb{P}^Q(\mathring{x}) = \mathbb{P}(\mathring{x}) = \mathbb{P}^P(\mathring{x})$. *Hence the claim* $\iota \in \mathrm{itags}(P)$ *because* $\mathring{x} \in \mathrm{bv}(\mathring{P})$.

(2) When $\iota \in \mathrm{ntags}(Q)$ *then it has to be the case that* $\iota \in \mathrm{ntags}(\mathbb{P}^Q(\mathring{p}_0))$ *for some* $\mathring{p}_0 \in \mathrm{fv}(\mathring{P})$ *because a form template can not introduce a* ν*-binder. Now it is easy to prove* $\iota \in \mathrm{ntags}(\mathbb{P}^P(\mathring{p}_0))$ *using the induction hypothesis when* $\mathring{p} = \mathring{p}_0$. *Hence the claim because* $\mathring{p}_0 \in \mathrm{fv}(\mathring{P})$.

(3) Let $x \in \mathrm{fn}(Q)$. *When* $x \in \mathrm{fn}(\mathring{P})$ *then* $x \in \mathrm{fn}(\mathcal{R})$ *and thus the claim clearly holds. Let us suppose* $x \notin \mathrm{fn}(\mathring{P})$. *Thus the occurrence of* x *in* Q *which contributes to* $\mathrm{fn}(Q)$ *comes from the value of some variable* $\mathring{z}$ *in template* $\mathring{P}$. *Clearly* $\mathring{z}$ *has to be a free variable, that is,* $\mathring{z} \in \mathrm{fv}(\mathring{P})$, *and we also know by* L6 *that* $\mathring{z}$ *is not a process variable under a substitution operator (*$\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}$*). The variable* $\mathring{z}$ *can, however, be a standalone process variable. Thus we know* $x \in \mathrm{fn}(\mathbb{P}^Q(\mathring{z}))$. *Now it is easy to prove that* $x \in \mathrm{fn}(\mathbb{P}^P(\mathring{z})) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$ *using the induction hypothesis when* $\mathring{z} = \mathring{p}$. *We can prove that* $\mathring{P} \vdash_\exists \mathring{y} \gg \mathring{z}$ *implies* $\mathbb{P}^Q(\mathring{y}) \neq x$ *for any* $\mathring{y}$ *because the occurrence of* x *in* $\mathbb{P}^Q(\mathring{z})$ *contributes to* $\mathrm{fn}(Q)$ *and thus can not occur under a binder which bind* x. *Rule* L5 *ensures that this property is satisfied for all occurrences of* $\mathring{z}$ *in* $\mathring{P}$. *Clearly* $\mathring{P} \vdash_\exists \mathring{y} \gg \mathring{z}$ *implies* $\mathbb{P}^P(\mathring{y}) \neq x$ *for any* $\mathring{y}$ *as well because* $\mathbb{P}^P$ *and* $\mathbb{P}^Q$ *agree on values of name variables. Thus* $x \in \mathrm{fn}(\mathbb{P}^P(\mathring{z}))$ *implies* $x \in \mathrm{fn}(P)$. *Above we*

have proved $x \in \mathrm{fn}(\mathbb{P}^P(\mathring{z})) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$ *which thus implies the claim* $x \in \mathrm{fn}(P) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$.

(RPar): *All the three claims follow directly from the induction hypothesis.*

(RNu): *There are* x, P_0, *and* Q_0 *such that* $P = \nu x.P_0$ *and* $Q = \nu x.Q_0$ *and* $P_0 \overset{\mathcal{R}}{\hookrightarrow} Q_0$. *Let us prove the three claims separately.*

(1) Clearly $\mathrm{itags}(P) = \mathrm{itags}(P_0)$ *and* $\mathrm{itags}(Q) = \mathrm{itags}(Q_0)$. *By the induction hypothesis we obtain that* $\mathrm{itags}(P_0) \subseteq \mathrm{itags}(Q_0)$. *Hence the claim.*

(2) Let $\iota = \overline{x}$. *Clearly* $\mathrm{ntags}(P) = \mathrm{ntags}(P_0) \cup \{\iota\}$ *and* $\mathrm{ntags}(Q) = \mathrm{ntags}(Q_0) \cup \{\iota\}$. *By the induction hypothesis we obtain that* $\mathrm{itags}(P_0) \subseteq \mathrm{itags}(Q_0)$. *Hence the claim.*

(3) We see that $\mathrm{fn}(P) = \mathrm{fn}(P_0) \backslash \{x\}$ *and* $\mathrm{fn}(Q) = \mathrm{fn}(Q_0) \backslash \{x\}$. *Thus* $\mathrm{fn}(Q) \subseteq \mathrm{fn}(Q_0)$. *By the induction hypothesis we obtain that* $\mathrm{fn}(Q_0) \subseteq \mathrm{fn}(P_0) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$. *Now whenever* $y \in \mathrm{fn}(Q)$ *then* $y \neq x$ *and* $y \in \mathrm{fn}(Q_0)$. *Thus* $y \in \mathrm{fn}(P_0) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$. *Hence* $y \in \mathrm{fn}(P) \cup \mathrm{fn}(\mathcal{R}) \cup \{\bullet\}$ *because* $y \neq x$.

(RStr): *All the three claims follow directly from the induction hypothesis applying Lemma 4.7.1.* ∎

The following proves that with a well formed rule set a well formed process can rewrite only to a well formed process. In this proposition we, of course, do not implicitly suppose that Q is well formed because it is the claim to be proved.

Proposition 6.4.2 (Well-formedness preservation). *Let* P *and* $\mathcal{R}$ *be well formed and let* $P \overset{\mathcal{R}}{\hookrightarrow} Q$. *Then* Q *is well formed.*

Proof. *Let* P *and* $\mathcal{R}$ *be well formed and let* $P \overset{\mathcal{R}}{\hookrightarrow} Q$. *Firstly, by Lemma 6.4.1 we obtain the following.*

$$\mathrm{itags}(Q) \subseteq \mathrm{itags}(P) \quad \mathrm{ntags}(Q) \subseteq \mathrm{ntags}(P) \quad \mathrm{ftags}(Q) \subseteq \mathrm{ftags}(P) \cup \mathrm{tags}(\mathcal{R}) \cup \{\bullet\}$$

Thus clearly W4 *is satisfied for* Q. *Let us prove* W1 *for* Q, *that is, that* $\mathrm{itags}(Q)$ *and* $\mathrm{ntags}(Q) \cup \mathrm{ftags}(Q)$ *are disjoint. By* W1 *for* P *we know that* $\mathrm{itags}(P)$ *and* $\mathrm{ntags}(P) \cup \mathrm{ftags}(P)$ *are disjoint. Thus it is enough to prove that* $\mathrm{tags}(\mathcal{R}) \cup \{\bullet\}$ *is disjoint with* $\mathrm{itags}(P)$. *But it is easy to see because* $\mathrm{fn}(\mathcal{R}) \cup \{\bullet\} \subseteq \mathrm{SpecialTag}$ *by well-formedness of* $\mathcal{R}$ *and* SpecialTag *is disjoint with* $\mathrm{itags}(P)$ *by* W4 *for* P. *Hence* W1 *holds for* Q.

Now let us prove W2 *and* W3 *for* Q *by induction on the derivation of* $P \overset{\mathcal{R}}{\hookrightarrow} Q$. *Let* $P \overset{\mathcal{R}}{\hookrightarrow} Q$ *be derived by*

RRw: *There is some* **rewrite**$\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *and there is some* $\mathbb{P}$ *such that* $P = \mathbb{P}[\![\mathring{P}]\!]$ *and* $Q = \mathbb{P}[\![\mathring{Q}]\!]$. *Firstly, it is easy to see that* W3 *for* Q *follows from* R7 *for* $\mathring{Q}$

and W3 *for* P*. Let us prove* W2 *for* Q*. Suppose that* Q *contains two nested input-binders such that the outer input-binder binds type tag* ι_0 *and the inner input-binder bounds* ι_1*. To prove that* W3 *holds for* Q *it is enough to prove that* $\iota_0 \neq \iota_1$*. We distinguish the following two possibilities.*

(1) The outer binder is introduced by a form template from $\mathring{Q}$*. Thus there is some* $\mathring{x} \in \mathrm{bv}(\mathring{Q})$ *such that* $\overline{\mathbb{P}(\mathring{x})} = \iota_0$*. Moreover there is some* $\mathring{z} \in \mathrm{var}(\mathring{Q})$ *such that* $\iota_1 \in \mathrm{itags}(\mathbb{P}(\mathring{z}))$*. This covers the following three cases when (a)* $\mathring{z}$ *is a bound name variable, (b)* $\mathring{z}$ *is a standalone process variable, or (c)* $\mathring{z}$ *is a process variable under substitution (because substitution application does not change input-binders by Lemma 4.6.2). In all the cases we have that* $\mathring{Q} \vdash_\exists \mathring{x} \gg \mathring{z}$*. Thus by* R5 *and* R6 *we obtain that* $\mathring{P} \vdash_\forall \mathring{x} \gg \mathring{z}$*. Both* $\mathring{x}$ *and* $\mathring{z}$ *have to occur in* $\mathring{P}$ *by* R2 *and* R3 *and thus we have* $\mathring{P} \vdash_\exists \mathring{x} \gg \mathring{z}$*. Hence the same two binders which bind* ι_0 *and* ι_1 *are nested in* P *as well and thus* $\iota_0 \neq \iota_1$*.*

(2) Both input binders come from value of $\mathbb{P}$ *for some process variable* $\mathring{p}$ *in* $\mathring{P}$*. This covers both cases when* $\mathring{p}$ *is a standalone process variable or when* $\mathring{p}$ *is under substitution application (as above we use Lemma 4.6.2 to prove that substitution application does not change input-binders). Hence the nested input binders occur in* $\mathbb{P}(\mathring{p})$ *for some* $\mathring{p} \in \mathrm{fv}(\mathring{Q})$*. By* R2 *we obtain* $\mathring{p} \in \mathrm{fv}(\mathring{P})$ *and thus* $\mathbb{P}(\mathring{p})$ *is a subprocess of* P *and hence well formed by Lemma 4.4.1. Thus clearly* $\iota_0 \neq \iota_1$*.*

RAct**:** *There is some* $\mathbf{active}\{\, \mathring{p} \,\mathbf{in}\, \mathring{P} \,\} \in \mathcal{R}$ *and there are some* P_0*,* Q_0*, and* $\mathbb{P}$ *such that* $P = (\mathbb{P}[\mathring{p} \mapsto P_0])[\![\mathring{P}]\!]$ *and* $Q = (\mathbb{P}[\mathring{p} \mapsto Q_0])[\![\mathring{P}]\!]$ *and* $P_0 \xrightarrow{\mathcal{R}} Q_0$*. Now* P *is well formed and thus* P_0 *is well formed by Lemma 4.4.1. By the induction hypothesis we have that* Q_0 *is well formed. Now* W2 *for* Q *is implied by* W2 *for* Q_0 *and by* W2 *for* P *together with* $\mathrm{itags}(Q_0) \subseteq \mathrm{itags}(P_0)$ *proved above. Finally,* W3 *for* Q *follows from* W3 *for* Q_0 *and* P *because* Q *does not contain any additional forms not contained in* Q_0 *and* P*.*

RPar**:** *Thus* $P = P_0 \mid R_0$ *and* $Q = Q_0 \mid R_0$ *for some* P_0*,* Q_0*, and* R_0 *such that* $P_0 \xrightarrow{\mathcal{R}} Q_0$*. Now* P_0 *is well scoped by Lemma 4.4.1 and thus* Q_0 *is well scoped by the induction hypothesis. Thus* W2 *and* W3 *for* Q *follows from* W2 *and* W3 *for* Q_0 *and* R_0 *because parallel composition can introduce neither additional nesting of input-binders nor any additional form ("additional" means "not present in* Q_0 *and* R_0*").*

RNu**:** *Follows directly from the induction hypothesis using Lemma 4.4.1.*

RStr**:** *Follows directly from the induction hypothesis using Lemma 4.7.2.* ∎

Chapter 7

The Generic Type System POLY✶

POLY✶ provides a generic notion of *shape predicates* which are rooted oriented graphs that represent sets of META✶ processes. A shape predicate Π describes possible shapes of process syntax trees. The set of processes described by Π is called the *meaning* of Π. All instantiations of POLY✶ use the same syntax of shape predicates and thus the meaning of a shape predicate is not necessarily closed under rewriting. In Section 7.6, we shall define *shape $\mathcal{R}$-types* for every rule description $\mathcal{R}$ to be a subset of shape predicates which is closed under rewritings with $\mathcal{R}$.

Many interesting properties of processes can be expressed as properties of shape $\mathcal{R}$-types. How to use shape types to reason about specific properties of processes in specific process calculi is demonstrated in Part III of this thesis which contains many examples.

7.1 Types of Basic META✶ Entities

For all kinds of basic (non-process) META✶ entities (sequences, messages, elements, and forms) we define corresponding types (sequence types, message types, element types, and form types). Each type represents a set of entities of the appropriate kind, for example, a message type represents a set of messages. Let ζ range over the above type entities. When ζ represents Z then we say that Z *matches* ζ or that Z *has type* ζ.

Type tags can be seen as types of names, ι represents all names of the shape a^{ι}. The syntax and semantics of other basic type entities is presented in the top part of Figure 7.1. The meaning of type entities is defined using the binary relation $\vdash Z : \zeta$ which expresses that Z has type ζ. The sequence type "$\iota_0 \ldots \iota_k$" describes any META✶ sequence of the length k whose i-th name has the type tag ι_i.

Sequence type sets and message types are both types of META✶ messages. The difference is that message types allow us to recognize single name messages and composed messages directly from their types. For example, we can see $\vdash \mathsf{x}^{\times} : \{\mathsf{x}\}$

Syntax of POLY✶ *basic type entities:*

$$\begin{array}{llll} \sigma & \in & \text{SequenceType} & ::= \quad \iota_0 \ldots \iota_k \\ \Sigma & \in & \text{SequenceTypeSet} & = \quad \text{power}_{\text{fin}}(\text{SequenceType}) \\ \mu & \in & \text{MessageType} & ::= \quad \Sigma\!* \mid \iota \\ \varepsilon & \in & \text{ElementType} & ::= \quad \iota \mid (\iota_1, \ldots, \iota_k) \mid \langle \mu_1, \ldots, \mu_k \rangle \\ \varphi & \in & \text{FormType} & ::= \quad \varepsilon_0 \ldots \varepsilon_k \end{array}$$

Matching of basic META✶ *entities against type entities:*

$$\frac{}{\vdash a^{\iota} : \iota}\ (\textsc{TName}) \qquad \frac{\forall i \leqslant k \quad \vdash x_i : \iota_i}{\vdash x_0 \ldots x_k : \iota_0 \ldots \iota_k}\ (\textsc{TSeq}) \qquad \frac{\vdash s : \sigma \quad \sigma \in \Sigma}{\vdash s : \Sigma}\ (\textsc{TSet})$$

$$\frac{}{\vdash \mathsf{0} : \Sigma}\ (\textsc{TEmp}) \qquad \frac{\vdash M_0 : \Sigma \quad \vdash M_1 : \Sigma}{\vdash M_0.M_1 : \Sigma}\ (\textsc{TCmp}) \qquad \frac{\vdash M : \Sigma \quad M \notin \text{Name}}{\vdash M : \Sigma\!*}\ (\textsc{TStar})$$

$$\frac{\forall i\colon 0 < i \leqslant k \quad \vdash x_i : \iota_i}{\vdash (x_1, \ldots, x_k) : (\iota_1, \ldots, \iota_k)}\ (\textsc{TIn}) \qquad \frac{\forall i\colon 0 < i \leqslant k \quad \vdash M_i : \mu_i}{\vdash \langle M_1, \ldots, M_k \rangle : \langle \mu_1, \ldots, \mu_k \rangle}\ (\textsc{TOut})$$

$$\frac{\forall i \leqslant k \quad \vdash E_i : \varepsilon_i}{\vdash E_0 \ldots E_k : \varepsilon_0 \ldots \varepsilon_k}\ (\textsc{TEls})$$

Figure 7.1: Syntax of POLY✶ shape predicates.

but $\not\vdash \mathsf{x}^{\mathsf{x}} : \{\mathsf{x}\}*$, and thus whenever $\vdash M : \Sigma*$ then M must be a composed message. This allows us to predict behavior of substitution application only from types of messages in its range. For example, a substitution which contain only single names in its range can not produce a syntactic error "•". A sequence type set Σ describes all messages whose sequence parts are described by some sequence type from Σ (allowing repetitions). For example, we have $\vdash (\mathsf{in\ a.in\ b^a}).\mathsf{x}{:}\{\mathsf{in\ a}, \mathsf{x}\}$ and $\vdash \mathsf{x}{:}\{\mathsf{in\ a}, \mathsf{x}\}$ and also $\vdash \mathsf{out^x.in^x}{:}\{\mathsf{in\ a}, \mathsf{x}\}$. On the other hand we obtain $\not\vdash \mathsf{x}{:}\{\mathsf{in\ a}, \mathsf{x}\}*$ as mentioned above and thus the only message type that describes x^{x} is x.

Element and form types simply resemble the syntax of elements and forms. All input- and output-element types and form types describe only entities of the same length as the type. Note that rule TSeq is a special case of rule TEls and thus TSeq could be omitted.

Let itags(φ) be the set of all input-bound type tags of φ, that is, those type tags that occur inside some input-element type $(\iota_1, \ldots, \iota_k)$. Let $[\![\zeta]\!]$ be the meaning of ζ, that is, the set of all META✶ entities that match ζ. Formal definitions of these two notions and description of their basic properties can be found in Section 8.1.

Application of a type substitution to sequence types:

$$\ddot{\mathfrak{o}}(\iota_0 \ldots \iota_k) = \begin{cases} \Sigma & \text{if } k = 0 \;\&\; \mathfrak{o}(\iota_0) = \Sigma* \\ \{(\bar{\mathfrak{o}}\iota_0 \ldots \bar{\mathfrak{o}}\iota_k)\} & \text{otherwise} \end{cases}$$

Application of a type substitution to message types:

$$\dot{\mathfrak{o}}\iota = \begin{cases} \mathfrak{o}(\iota) & \text{if } \iota \in \mathrm{dom}(\mathfrak{o}) \\ \iota & \text{otherwise} \end{cases} \qquad \dot{\mathfrak{o}}(\{\sigma_1, \ldots, \sigma_k\}*) = (\ddot{\mathfrak{o}}\sigma_1 \cup \cdots \cup \ddot{\mathfrak{o}}\sigma_k)*$$

Application of a type substitution to element types and form types:

$$\bar{\mathfrak{o}}\iota = \begin{cases} \mathfrak{o}(\iota) & \text{if } \mathfrak{o}(\iota) \in \mathrm{TypeTag} \\ \iota & \text{if } \iota \notin \mathrm{dom}(\mathfrak{o}) \\ \bullet & \text{otherwise} \end{cases} \qquad \begin{array}{l} \bar{\mathfrak{o}}\langle\mu_1, \ldots, \mu_k\rangle = \langle\dot{\mathfrak{o}}\mu_1, \ldots, \dot{\mathfrak{o}}\mu_k\rangle \\ \bar{\mathfrak{o}}\,(\iota_1, \ldots, \iota_k) = (\iota_1, \ldots, \iota_k) \\ \bar{\mathfrak{o}}(\varepsilon_0 \ldots \varepsilon_k) = (\bar{\mathfrak{o}}\varepsilon_0) \ldots (\bar{\mathfrak{o}}\varepsilon_k) \end{array}$$

Figure 7.2: Application of a type substitution to POLY✶ entities.

7.2 Type Substitutions

In this section we introduce type substitutions which are similar to ordinary META✶ substitutions but they apply to type entities. Type substitutions can be seen as types of META✶ substitutions, each type substitution representing a set of META✶ substitutions. Type substitutions are used in shape predicates as labels of *flow edges* which are in turn used to select shape types out of shape predicates. Flow edges are described below in Section 7.4.

DEFINITION 7.2.1. *A* **type substitution***, denoted* $\mathfrak{o}$*, is a finite function from type tags to message types.* ∎

Application of $\mathfrak{o}$ to various type entities, defined in Figure 7.2, is designed to correspond to applications of META✶ substitutions described by $\mathfrak{o}$. Application of $\mathfrak{o}$ to a sequence type σ is written $\ddot{\mathfrak{o}}\sigma$. It maps sequence types to sequence type sets. Application of $\mathfrak{o}$ to a message type μ is written $\dot{\mathfrak{o}}\mu$. It maps message types to message types. Finally, application of $\mathfrak{o}$ to an element or form type ζ is written $\bar{\mathfrak{o}}\zeta$. It maps element types and form types in turn to element types and form types. The result of $\bar{\mathfrak{o}}\iota$ is always a type tag. Three different substitution application operators $\ddot{\mathfrak{o}}$, $\dot{\mathfrak{o}}$, and $\bar{\mathfrak{o}}$ are necessary because a type substitution is applied in different ways to type tags inside sequence types, to single type tag message types, and to single type tag element types. For example, with $\mathfrak{o} = \{\mathsf{x} \mapsto \{\mathsf{in\ a}\}*\}$ we have $\ddot{\mathfrak{o}}\mathsf{x} = \{\mathsf{in\ a}\}$ and $\dot{\mathfrak{o}}\mathsf{x} = \{\mathsf{in\ a}\}*$ and $\bar{\mathfrak{o}}\mathsf{x} = \bullet$. To illustrate this further let us consider the following form type "x<x, {x, in x}*>" which contains x at four different positions. Application of the above $\mathfrak{o}$ to this form type is as follows.

$$\bar{\mathfrak{o}}(\mathsf{x{<}x, \{x, in\ x\}{*}{>}}) = \bullet\mathsf{{<}\{in\ a\}{*}, \dot{\mathfrak{o}}(\{x, in\ x\}*){>}} = \bullet\mathsf{{<}\{in\ a\}{*}, \{in\ a, in\ \bullet\}{*}{>}}$$

Finally, note that names inside input-element types are left intact, as in the case META✶ substitutions, and thus $\bar{\mathfrak{o}}(\mathsf{(x)}) = \mathsf{(x)}$.

The following defines META✶ substitutions described by a type substitution $\mathfrak{o}$, that is, the meaning of $\mathfrak{o}$.

DEFINITION 7.2.2. *Write* $\vdash \mathbb{S} : \mathfrak{o}$ *when*

(1) there is a bijection from $\mathrm{dom}(\mathbb{S})$ *to* $\mathrm{dom}(\mathfrak{o})$ *that maps* a^ι *to* ι*, and*

(2) for any $a^\iota \in \mathrm{dom}(\mathbb{S})$ *it holds that* $\vdash \mathbb{S}(a^\iota) : \mathfrak{o}(\iota)$. ∎

Point (*1*) can equivalently be stated as

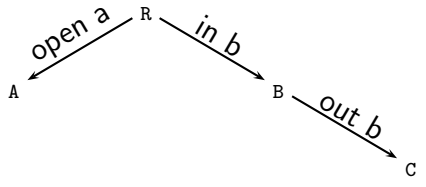
(1') $\overline{\mathrm{dom}(\mathbb{S})} = \mathrm{dom}(\sigma)$ and different names from $\mathrm{dom}(\mathbb{S})$ have different type tags.

Thus, for example, neither $\mathbb{S}_0 = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{a}^{\mathsf{a}}, \mathsf{y}^{\mathsf{x}} \mapsto \mathsf{b}^{\mathsf{b}}\}$ nor $\mathbb{S}_1 = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{a}^{\mathsf{a}}, \mathsf{y}^{\mathsf{x}} \mapsto \mathsf{b}^{\mathsf{a}}\}$ has the type $\mathfrak{o} = \{\mathsf{x} \mapsto \mathsf{a}\}$ even though $\mathbb{S}_1$ becomes $\mathfrak{o}$ when we forget basic name parts of names. In fact both $\mathbb{S}_0$ and $\mathbb{S}_1$ have no type at all because x^{x} and x^{y} are different names but have the same type tag. Hence for $\vdash \mathbb{S} : \mathfrak{o}$ to hold there has to be for every $\iota \in \mathrm{dom}(\mathfrak{o})$ exactly one $a^\iota \in \mathrm{dom}(\mathbb{S})$ (for some a). This gives us an unambiguous correspondence between assignments (pairs) in $\mathbb{S}$ and $\mathfrak{o}$. Point (*2*) says that messages in the range of $\mathbb{S}$ match the corresponding message types in the range of $\mathfrak{o}$.

Application of a type substitution to different type entities might seem complicated but it is carefully designed to reach the following property. When $\mathfrak{o}$ describes $\mathbb{S}$ ($\vdash \mathbb{S} : \mathfrak{o}$) and φ describes F ($\vdash F : \varphi$) then the application of $\mathfrak{o}$ to φ describes the application of $\mathbb{S}$ to F ($\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$). We call this property *type substitution correctness* and its proof and further discussions are found in Section 8.2

7.3 POLY✶ Shape Predicates

A *shape predicate* is a rooted finite oriented graph with edges labeled by form types. The formal syntax of shape predicates is presented in the top part of Figure 7.3. A shape predicate $\langle \Gamma, \chi \rangle$ is the *shape graph* Γ together with the *root* χ. A shape graph can contain loops and cycles. A shape predicate describes a set of process syntax trees. A process P matches a shape predicate Π when P's syntax tree is a "subgraph" of Π. Shape predicates look alike processes drawn as graphs where parallel composition ("|") corresponds to branching and prefixing (".") correspond to sequencing of edges. For example, the shape predicate $\Pi = \langle \Gamma, \mathtt{R} \rangle$ where

$\Gamma =$ 

Syntax of POLY✶ *shape predicates:*

$$\begin{array}{rcll} \chi & \in & \text{Node} & ::= \ \mathtt{X} \mid \mathtt{Y} \mid \mathtt{Z} \mid \cdots \\ \eta & \in & \text{Edge} & ::= \ \chi_0 \xrightarrow{\varphi} \chi_1 \mid \chi_0 \overset{\varpi}{\dashrightarrow} \chi_1 \\ \Gamma & \in & \text{ShapeGraph} & = \ \text{power}_{\text{fin}}(\text{Edge}) \\ \Pi & \in & \text{ShapePredicate} & ::= \ \langle \Gamma, \chi \rangle \end{array}$$

Matching META✶ *processes against shape predicates:*

$$\frac{}{\vdash 0 : \Pi}\ (\text{TNUL}) \qquad \frac{\vdash F : \varphi \quad (\chi \xrightarrow{\varphi} \chi_0) \in \Gamma \quad \vdash P_0 : \langle \Gamma, \chi_0 \rangle}{\vdash F.P_0 : \langle \Gamma, \chi \rangle}\ (\text{TFRM})$$

$$\frac{\vdash P : \Pi \quad \vdash Q : \Pi}{\vdash P \mid Q : \Pi}\ (\text{TPAR}) \qquad \frac{\vdash P : \Pi}{\vdash \nu x.P : \Pi}\ (\text{TNU}) \qquad \frac{\vdash P : \Pi}{\vdash !P : \Pi}\ (\text{TREP})$$

Figure 7.3: Syntax and Semantics of POLY✶ shape predicates.

represents the process

open a.0 | in b.out b.0

Names of nodes are just opaque identifiers and the meaning of a shape predicate is given by edge labels. The same edge in a shape graph can be used repeatedly or not at all when matching parallel processes and thus all the following processes match Π.

open a.0 (open a.0 | open a.0) in b.(out b.0 | out b.0)

On the other hand "in b.in b.0" does not match Π because the possibility to reuse edges applies only to parallel composition. When matching processes against shape predicates we ignore replication ("!"), name restriction ("ν"), and basic name parts of names. Thus the following processes are also represented by the above Π.

!open x^a.0 νa.(open a.0 | !open a.0) in x^b.(out y^b.0 | out z^b.0)

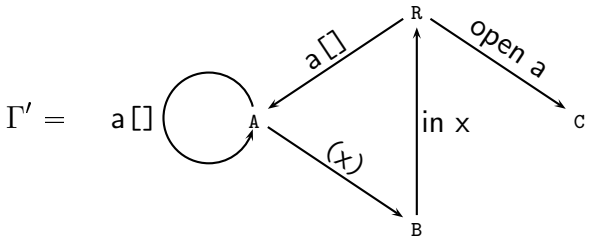
From the above we can see that when some P matches Π then also "$P \mid P$" matches Π. Thus it is reasonable to ignore replication because a replicated process "$!P$" behaves as finitely many copies of P in parallel. Name restriction can be ignored because type tags act as handles of bound names and are preserved under α-conversion. Because type entities are build only from type tags, different ν-bound names with the same type tag are handled as the same name by the type analysis. This can cause some over-approximation in types but it does not influence analysis correctness. Handling of name restriction in shape types is further discussed in Section 9.1.

The syntax of shape predicates in Figure 7.3 also defines a second kind of edges labeled by type substitution which are called *flow edges*. Edges labeled by form types are called *form edges*. Flow edges do not influence the meaning of a shape predicate and they are described in Section 7.4.

The typing relation $\vdash P : \Pi$ is the smallest relation that satisfies the rules from the bottom part of Figure 7.3. The null process `0` matches any shape predicate. Rule TFRM says that in order to match $F.P_0$ with $\langle \Gamma, \chi \rangle$ we need to find some edge $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ outgoing from χ such that F matches φ and then we need to match P_0 with $\langle \Gamma, \chi_0 \rangle$. Other typing rules are straightforward. The meaning of shape predicates and the subtyping relation are defined as follows.

DEFINITION 7.3.1. *The* **meaning** $[\![\Pi]\!]$ **of** Π *is defined as* $[\![\Pi]\!] = \{P : (\vdash P : \Pi)\}$. *Write* $\Pi_0 \preccurlyeq \Pi_1$ *when* $[\![\Pi_0]\!] \subseteq [\![\Pi_1]\!]$. ∎

Shape graphs can contain loops and cycles and thus a single shape predicate, which is a finite object, can describe processes of an arbitrary depth. Consider the shape predicate $\Pi' = \langle \Gamma', \mathtt{R} \rangle$ where Γ' is the following graph.



It can describe an arbitrarily deep nesting of the ambient `a` like "`a[a[a[···]]]`". The meaning of a shape predicate does not necessarily be closed under rewriting with rules $\mathcal{R}$. For an example, let us consider the rewriting rule set $\mathcal{A}_{\mathsf{mon}}$ from Section 5.3 which instantiates META✶ to monadic Mobile Ambients. Then the above Π' is not closed under rewriting with $\mathcal{A}_{\mathsf{mon}}$ because we have

$$\mathtt{a[(x).0]} \mid \mathtt{open\ a.0} \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \mathtt{(x).0}$$

It is easy to see that the left-hand side process matches Π' (recall that $x[P]$ stand for $x[].P$) but the right-hand side "`(x).0`" does not. Section 7.6 describes how to recognize shape predicates closed under rewriting with $\mathcal{R}$.

7.4 Flow Edges and Flow Closed Graphs

A shape graph also contains *flow edges* of the shape $\chi_0 \overset{\sigma}{\dashrightarrow} \chi_1$ which are used in the type inference algorithm and in recognizing shape predicates closed under rewriting. Flow edges are labeled by type substitutions and their presence in a shape graphs does not affect the meaning of a shape predicate. The intended meaning of a flow edge $(\chi_0 \overset{\sigma}{\dashrightarrow} \chi_1) \in \Gamma$ is to describe possible movements of processes that involve substitution application as follows. Let $(\chi_0 \overset{\sigma}{\dashrightarrow} \chi_1) \in \Gamma$. Then we want $\vdash \mathbb{S} : \sigma$ and $\vdash P : \langle \Gamma, \chi_0 \rangle$ to imply $\vdash \bar{\mathbb{S}}P : \langle \Gamma, \chi_1 \rangle$. This intended meaning is of course not satisfied for an arbitrary flow edge added to an arbitrary shape graph. The flow

edge $(\chi_0 \overset{\varpi}{\dashrightarrow} \chi_1) \in \Gamma$ can be seen as a request to copy the content of the node χ_0 (that is, the subgraph containing all the edges reachable from χ_0 by an oriented path) to node χ_1 and apply the substitution ϖ to the copied content. In this section we define the class of *flow-closed* shape predicates which satisfy the intended meaning of flow edges.

We could define flow-closed shape predicates simply to be the predicates which satisfy the above intended meaning of flow edges. But instead, we provide a constructive and easier to verify definition and we prove it to imply the above intended meaning.

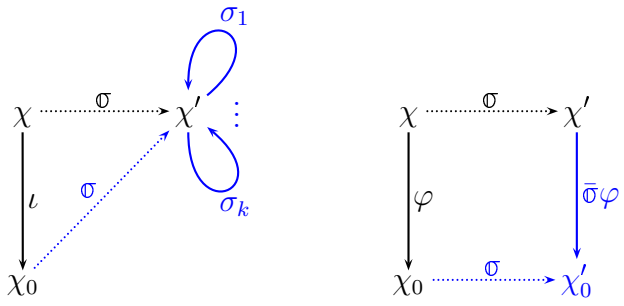
DEFINITION 7.4.1. *A shape graph Γ is said to be* **flow-closed** *iff whenever it contains $\chi \xrightarrow{\varphi} \chi_0$ and $\chi \overset{\varpi}{\dashrightarrow} \chi'$ such that* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\varpi) = \emptyset$ *then it holds that*

(F1) *if $\varphi = \iota$ for some ι and $\varpi(\iota) = \Sigma\ast$ then $\{\chi' \xrightarrow{\sigma} \chi' \colon \sigma \in \Sigma\} \cup \{\chi_0 \overset{\varpi}{\dashrightarrow} \chi'\} \subseteq \Gamma$,*

(F2) *otherwise there is χ_0' such that $\{\chi_0 \overset{\varpi}{\dashrightarrow} \chi_0', \chi' \xrightarrow{\bar{\varpi}\varphi} \chi_0'\} \subseteq \Gamma$.*

The shape predicate $\langle \Gamma, \chi \rangle$ is flow-closed iff Γ is. ∎

Conditions F1 and F2 can be visualized by the following diagrams where the blue edges are those whose existence is required by the corresponding condition. Let Σ from case F1 be $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$.



(F1) (F2)

To explain these conditions let us consider some Γ with $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ and $(\chi \overset{\varpi}{\dashrightarrow} \chi') \in \Gamma$ such that $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\varpi) = \emptyset$ as in the definition. Rule F1 applies when φ is some type tag ι and $\varpi(\iota) = \Sigma\ast$ for some Σ. How F1 works will be described shortly. Rule F2 applies when either (a) $\varphi = \varepsilon_1 \ldots \varepsilon_k$ with $k > 1$, or (b) $k = 1$ and ε_1 is some type tag ι such that and $\varpi(\iota)$ is not a starred message type (of the shape $\Sigma'\ast$). Case (b) covers two possibilities, (b1) that $\varpi(\iota)$ is a type tag and (b2) that $\iota \notin \mathrm{dom}(\varpi)$. Thus case (b) simply describes the situation when $\bar{\varpi}\iota \neq \bullet$.

Let us describe rule F2 first on the following example.

EXAMPLE 7.4.2. *Let "$\varphi =$ in x" and $\varpi = \{\mathsf{x} \mapsto \mathsf{a}\}$. Let Γ be the graph from the diagram for* F2 *above, that is, the following graph.*

$$\Gamma = \{\chi \xrightarrow{\mathsf{in\ x}} \chi_0, \chi \overset{\varpi}{\dashrightarrow} \chi', \chi' \xrightarrow{\mathsf{in\ a}} \chi_0', \chi_0 \overset{\varpi}{\dashrightarrow} \chi_0'\}$$

Now it is clear that, for example, the process "$P = \mathsf{in}\ \mathsf{x}.0$" matches $\langle \Gamma, \chi \rangle$. Moreover let us take $\mathbb{S} = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{a}^{\mathsf{a}}\}$ so that we have $\vdash \mathbb{S} : \sigma$. The intended meaning of the flow edge $(\chi \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$ says that "$\bar{\mathbb{S}}P = \mathsf{in}\ \mathsf{a}$" has to match $\langle \Gamma, \chi' \rangle$. That is why the existence of the first blue edge $(\chi' \xrightarrow{\bar{\sigma}\varphi} \chi'_0) \in \Gamma$ for some χ'_0 is required in case F2*. In this example the required edge is $(\chi' \xrightarrow{\mathsf{in\ a}} \chi'_0) \in \Gamma$ which ensures that "$\bar{\mathbb{S}}P = \mathsf{in}\ \mathsf{a}$" is in the meaning of $\langle \Gamma, \chi' \rangle$. The second blue edge $(\chi_0 \overset{\sigma}{\dashrightarrow} \chi'_0) \in \Gamma$ in case* F2 *is required to propagate the request for new edges throughout the graph so that the intended meaning of $(\chi \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$ is satisfied for all processes.* ▫

Now let us demonstrate F1 on a simple example.

Example 7.4.3. *Let $\varphi = \mathsf{x}$ and $\sigma = \{\mathsf{x} \mapsto \{\mathsf{in\ a}, \mathsf{out\ a}\}*\}$. Let Γ be the graph from the diagram for* F1 *above, that is, the following graph.*

$$\Gamma = \{\chi \xrightarrow{\mathsf{x}} \chi_0, \chi \overset{\sigma}{\dashrightarrow} \chi', \chi' \xrightarrow{\mathsf{in\ a}} \chi', \chi' \xrightarrow{\mathsf{out\ a}} \chi', \chi_0 \overset{\sigma}{\dashrightarrow} \chi'\}$$

Now $P = \mathsf{x}^{\mathsf{x}}.0$ matches $\langle \Gamma, \chi \rangle$. Let us take $\mathbb{S} = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{in\ a.out\ a.in\ a}\}$ so that we have $\vdash \mathbb{S} : \sigma$. The intended meaning of the flow edge $(\chi \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$ says that "$\bar{\mathbb{S}}(\mathsf{x}^{\mathsf{x}}.0) = \mathbb{S}(\mathsf{x}^{\mathsf{x}})_*0 = \mathsf{in\ a.out\ a.in\ a}.0$" has to match $\langle \Gamma, \chi' \rangle$. It is easy to check that it is true because the loops $\chi' \xrightarrow{\mathsf{in\ a}} \chi'$ and $\chi' \xrightarrow{\mathsf{out\ a}} \chi'$ are present in Γ. Recall that the message type $\{\mathsf{in\ a}, \mathsf{out\ a}\}*$ describes any message with arbitrary many repetitions of the sequence "$\mathsf{in\ a}$" and that is why the required form edges in case F1 *are loops. The second flow edge $(\chi_0 \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$ is again required to propagate the request for new edges so that the intended meaning of $(\chi \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$ is satisfied also for processes of the shape $a^{\mathsf{x}}.P_0$ with a non-null continuation P_0.* ▫

An important special case of flow edges is when $\sigma = \emptyset$. The intended meaning of $(\chi \overset{\emptyset}{\dashrightarrow} \chi') \in \Gamma$ then says that when $\vdash P : \langle \Gamma, \chi \rangle$ then it has to hold that $\vdash P : \langle \Gamma, \chi' \rangle$. In order words, it says that $\langle \Gamma, \chi \rangle \preccurlyeq \langle \Gamma, \chi' \rangle$. We therefore speak of $\chi \overset{\emptyset}{\dashrightarrow} \chi'$ as a *subtyping edge* and we write simply $\chi \dashrightarrow \chi'$.

Finally, note that Definition 7.4.1 is constructive in the sense that it provides the algorithm to check whether or not a shape predicate is flow-closed. It does not, however, give us an algorithm to compute the edges necessary to make an arbitrary graph flow-closed. Mere adding of edges which are requested by the definition would not give us a terminating algorithm which is further discussed in Section 11.4 and Section 11.5.

Further discussion of flow closure including some alternative implementation choices can be found in Section 8.4. The property that the intended meaning of flow edges is satisfied in flow-closed graphs is formulated and proved in Section 8.5.

7.5 Closed Shape Predicates

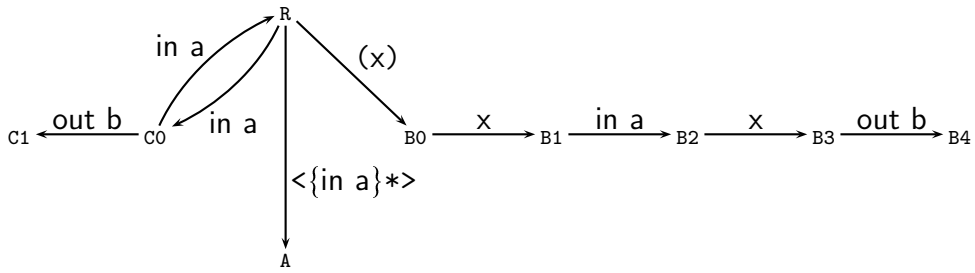
The meaning of a shape predicate is not necessarily closed under rewriting with arbitrary rewriting rules. Thus in order to use shape predicates as process types, it is desirable to find a decidable and efficient procedure to recognize shape predicates closed under rewriting. In this section we demonstrate the complexity of this task.

We call a shape predicate $\mathcal{R}$*-closed* when its meaning is closed under rewriting with $\mathcal{R}$. In the next section we shall define *shape* $\mathcal{R}$*-types* to be a subclass of all $\mathcal{R}$-closed shape predicates which can be recognized by a simple closure test.

DEFINITION 7.5.1. *Let* $\mathcal{R}$ *be a rule set. A shape predicate* Π *is called* **$\mathcal{R}$-closed**, *written* $\mathcal{R} \Vdash_{\mathsf{closed}} \Pi$, *iff* $\vdash P : \Pi$ *and* $P \overset{\mathcal{R}}{\hookrightarrow} Q$ *imply* $\vdash Q : \Pi$. ∎

It is not always easy to recognize $\mathcal{R}$-closed shape predicates as demonstrated by the following example. Those shape predicates that can not be proved $\mathcal{R}$-closed by the simple closure test from the next section will not be considered $\mathcal{R}$-types. That is to say that not every $\mathcal{R}$-closed shape predicate is necessarily an $\mathcal{R}$-type.

EXAMPLE 7.5.2. *Let us consider the shape predicate* $\Pi = \langle \Gamma, \mathtt{R} \rangle$ *where* Γ *is the following graph.*



Note that the following holds.

$$\vdash \mathsf{in\ a.in\ a.in\ a.out\ b.0} : \Pi \qquad \textit{but} \qquad \not\vdash \mathsf{in\ a.in\ a.out\ b.0} : \Pi$$

We can see that a process of the shape "$\mathsf{in\ a.} \cdots \mathsf{.in\ a.out\ b.0}$*" matches* Π *iff "*$\mathsf{in\ a}$*" is repeated odd number of times. When we consider the monadic Mobile Ambients rewriting rules* $\mathcal{A}_{\mathsf{mon}}$ *from Section 5.3 we can verify that* Π *is actually* $\mathcal{A}_{\mathsf{mon}}$*-closed. Only the Mobile Ambients communication rule can apply to a process* P *in the meaning of* Π *because no ambient boundaries are mentioned in* Π. *Now it is easy to see that whenever the communication rule introduces some process of the shape "*$\mathsf{in\ a.} \cdots \mathsf{.in\ a.out\ b.0}$*" then "*$\mathsf{in\ a}$*" has to be repeated odd number of times because there are two occurrences of "*x*" on the path from* $\mathtt{B0}$ *to* $\mathtt{B4}$. *It is not trivial, however, to find an efficient algorithmic way to recognize that* Π *and all examples of this kind are* $\mathcal{A}_{\mathsf{mon}}$*-closed. Later (see also the last paragraph of Section 10.4) we will see that* Π *is* $\mathcal{A}_{\mathsf{mon}}$*-closed but not an* $\mathcal{A}_{\mathsf{mon}}$*-type.* □

Instantitating basic templates to types:

$$\mathbb{\pi}(\!|a^{\iota}|\!) = \iota \qquad \mathbb{\pi}(\!|(\mathring{x}_1,\ldots,\mathring{x}_k)|\!) = (\mathbb{\pi}(\mathring{x}_1),\ldots,\mathbb{\pi}(\mathring{x}_k))$$
$$\mathbb{\pi}(\!|\mathring{x}|\!) = \mathbb{\pi}(\mathring{x}) \qquad \mathbb{\pi}(\!|<\mathring{m}_1,\ldots,\mathring{m}_k>|\!) = <\mathbb{\pi}(\mathring{m}_1),\ldots,\mathbb{\pi}(\mathring{m}_k)>$$
$$\mathbb{\pi}(\!|\mathring{m}|\!) = \mathbb{\pi}(\mathring{m}) \qquad \mathbb{\pi}(\!|\mathring{E}_0\ldots\mathring{E}_k|\!) = \mathbb{\pi}(\!|\mathring{E}_0|\!)\ldots\mathbb{\pi}(\!|\mathring{E}_k|\!)$$

Relating templates and shape graphs (s ranges over $\{\mathsf{L},\mathsf{R}\}$*):*

$$\frac{\mathbb{\pi}(\mathring{p}) = \chi}{\mathbb{\pi} \models_{\mathsf{L}} \mathring{p} : \langle\Gamma,\chi\rangle}\ (\textsc{CVar}) \qquad \frac{(\mathbb{\pi}(\mathring{p}) \overset{\emptyset}{\dashrightarrow} \chi) \in \Gamma}{\mathbb{\pi} \models_{\mathsf{R}} \mathring{p} : \langle\Gamma,\chi\rangle}\ (\textsc{CFlow})$$

$$\frac{(\mathbb{\pi}(\mathring{p}) \overset{\{\ldots,\mathbb{\pi}(\mathring{x}_i)\mapsto\mathbb{\pi}(\mathring{s}_i),\ldots\}}{\dashrightarrow} \chi) \in \Gamma}{\mathbb{\pi} \models_{\mathsf{R}} \{\ldots,\mathring{x}_i := \mathring{s}_i,\ldots\}\,\mathring{p} : \langle\Gamma,\chi\rangle}\ (\textsc{CSub}) \qquad \frac{}{\mathbb{\pi} \models_s 0 : \Pi}\ (\textsc{CNul})$$

$$\frac{\mathbb{\pi} \models_s \mathring{P}_0 : \Pi \quad \mathbb{\pi} \models_s \mathring{P}_1 : \Pi}{\mathbb{\pi} \models_s \mathring{P}_0 \mid \mathring{P}_1 : \Pi}\ (\textsc{CPar}) \qquad \frac{(\chi \xrightarrow{\mathbb{\pi}(\mathring{F})} \chi_0) \in \Gamma \quad \mathbb{\pi} \models_s \mathring{P}_0 : \langle\Gamma,\chi_0\rangle}{\mathbb{\pi} \models_s \mathring{F}.\mathring{P}_0 : \langle\Gamma,\chi\rangle}\ (\textsc{CFrm})$$

Figure 7.4: Instantiating templates to shape graphs.

7.6 Shape Types and Closure Test

In the previous section we have demonstrated that it is not always easy to recognize that a shape predicate is $\mathcal{R}$-closed. However, it is desirable that types can be effectively recognized. That is why in this section we define the class of $\mathcal{R}$*-types* to be an easier to recognize subclass of all $\mathcal{R}$-closed shape predicates.

We introduce a simple closure test which determines $\mathcal{R}$-types. The closure test can be briefly described as follows. *Apply the rewriting rules* $\mathcal{R}$ *directly to all active positions in a shape graph and check whether all the edges required by rules* $\mathcal{R}$ *are already present in the graph.* In the rest of this section we describe how rules are applied to a graph, what are active positions in a graph, and what are the edges required by the application of a rule to a graph.

In order to apply rewriting rules directly to graphs we need to establish a connection between process templates and shape graphs. For this purpose we define type instantiations $\mathbb{\pi}$ (the symbol $\mathbb{\pi}$ is a black board bold π) which connect templates with shape predicates just like process instantiations $\mathbb{P}$ connect templates with processes.

DEFINITION 7.6.1. *A* **type instantiation** $\mathbb{\pi}$ *is a finite function mapping* NameVar *to* TypeTag$\backslash\{\bullet\}$*,* MessageVar *to* MessageType*, and* ProcessVar *to* Node. ∎

Application of a template instantiation $\mathbb{\pi}$ to element and form templates, written $\mathbb{\pi}(\!|\mathring{Z}|\!)$, is defined in the top part of Figure 7.4. It fills in values of corresponding kinds for variables just like process instantiations do. Thus $\mathbb{\pi}$ instantiates element templates to element types and form templates to form types. Note that application of $\mathbb{\pi}$ to templates forgets the basic name part of names contained in templates

$(\pi[\![a^\iota]\!] = \iota)$. As a result only the type tags of special names mentioned in rewriting rules are relevant for the type analysis.

In the case of processes, we know that a process instantiation $\mathbb{P}$ and a process template $\mathring{P}$ uniquely determine the process $\mathbb{P}[\![\mathring{P}]\!]$. Note that a type instantiation π maps process variables to nodes but no graph to which these nodes belong has been mentioned yet. Instead of extending π so that π and $\mathring{P}$ uniquely determine a shape predicate we define the relation $\pi \models_{\mathsf{L}} \mathring{P} : \Pi$ which we read as "$\mathring{P}$ can be instantiated by π to Π". One π can instantiate the same $\mathring{P}$ to different shape predicates. The nodes which are values of π for process variables are supposed to be nodes of Π.

The following defines a straightforward relationship between process and type instantiations. A process instantiation $\mathbb{P}$ respects a type instantiation π when both are defined on the same variables and the values of $\mathbb{P}$ have appropriate types given by π.

DEFINITION 7.6.2. *A process instantiation* $\mathbb{P}$ **respects** *a type instantiation* π *on* Γ*, written* $\Gamma \vdash \mathbb{P} : \pi$*, iff all the following hold.*

(1) $\mathrm{dom}(\mathbb{P}) = \mathrm{dom}(\pi)$

(2) $\vdash \mathbb{P}(\mathring{x}) : \pi(\mathring{x})$ *for all* $\mathring{x} \in \mathrm{dom}(\mathbb{P})$

(3) $\vdash \mathbb{P}(\mathring{m}) : \pi(\mathring{m})$ *for all* $\mathring{m} \in \mathrm{dom}(\mathbb{P})$

(4) $\vdash \mathbb{P}(\mathring{p}) : \langle \Gamma, \pi(\mathring{p}) \rangle$ *for all* $\mathring{p} \in \mathrm{dom}(\mathbb{P})$ ∎

The relation $\pi \models_{\mathsf{L}} \mathring{P} : \Pi$, which is used for rule left-hand sides, is defined to be the smallest relation which satisfies the rules in the second part of Figure 7.4. The same figure defines also a similar relation $\pi \models_{\mathsf{R}} \mathring{P} : \Pi$ which is used for rule right-hand sides and will be described shortly. The relation $\pi \models_{\mathsf{L}} \mathring{P} : \Pi$ means there is a process instantiation $\mathbb{P}$ which respects π on the graph of Π such that $\mathbb{P}[\![\mathring{P}]\!]$ matches Π. In other words, $\pi \models_{\mathsf{L}} \mathring{P} : \Pi$ means that $\mathring{P}$ can be instantiated to some process P that matches Π. Let us demonstrate this on the following example.

EXAMPLE 7.6.3. *Let us consider the left-hand side "*$\mathring{P}$ = `ċ<å>.0 | ċ(ẋ).Q̊`*" of the communication rule from the asynchronous* π*-calculus*

$$\mathcal{P}_{\mathsf{async}} = \{\mathbf{rewrite}\{\, \mathring{c}\texttt{<}\mathring{a}\texttt{>.0 | }\mathring{c}\texttt{(}\mathring{x}\texttt{).}\mathring{Q} \hookrightarrow \{\mathring{x} := \mathring{a}\}\, \mathring{Q}\,\}\}$$

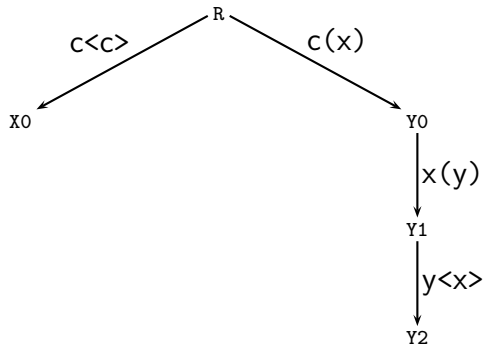
from Section 5.3. We can see that

$$\mathbb{P} = \{\mathring{c} \mapsto \mathsf{c}, \mathring{a} \mapsto \mathsf{c}, \mathring{x} \mapsto \mathsf{x}, \mathring{Q} \mapsto \mathsf{x(y).y<x>.0}\}$$

instantiates $\mathring{P}$ *to "*P = `c<c>.0 | c(x).x(y).y<x>.0`*" which matches the shape predicate*

$\Pi_0 = \langle \Gamma_0, \mathtt{R} \rangle$ *with the following shape graph.*

$\Gamma_0 =$



Now it is easy to check that $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi_0$ *where*

$$\mathbb{m} = \{\mathring{\mathtt{c}} \mapsto \mathtt{c}, \mathring{\mathtt{a}} \mapsto \mathtt{c}, \mathring{\mathtt{x}} \mapsto \mathtt{x}, \mathring{\mathtt{Q}} \mapsto \mathtt{Y0}\}$$

We can also directly see that $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi_0$ *holds just by comparing the syntax tree of* $\mathring{P}$ *with the shape graph* Π_0 *starting at the root node. Thus the relation* $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$ *allows us to recognize that* $\mathring{P}$ *can be instantiated to some process* P *of the type* Π *without constructing* P. □

When $\mathring{P}$ is a left-hand side of some rewriting rule then $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$ implies that the rewriting rule can be applied to at least one process matching Π. The relation $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$ does not hold for templates which contain the substitution application template $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}$ because the relation is supposed to be used only with left-hand sides of well formed rules.

The relation $\mathbb{m} \models_{\mathsf{R}} \mathring{Q} : \Pi$ is similar to $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$. Briefly, the design goal of the relation $\mathbb{m} \models_{\mathsf{R}} \mathring{Q} : \Pi$ is that when $\mathbb{P}$ respects $\mathbb{m}$ on the shape graph of Π then the instantiation $\mathbb{P}[\![\mathring{Q}]\!]$ of $\mathring{Q}$ has the type Π. This property holds provided that Π is flow-closed. The relation $\mathbb{m} \models_{\mathsf{R}} \mathring{Q} : \Pi$ holds also for templates which contain the substitution application template $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}$ and it is used with right-hand sides of rules. This will be described shortly.

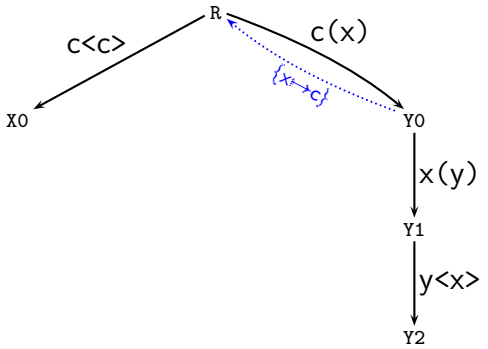
Technically, the difference between $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$ and $\mathbb{m} \models_{\mathsf{R}} \mathring{Q} : \Pi$ is in the handling of process variables $\mathring{p}$. In the case of the first relation, rule CVAR says that $\mathbb{m} \models_{\mathsf{L}} \mathring{p} : \langle \Gamma, \chi \rangle$ simply holds only when $\mathbb{m}(\mathring{p}) = \chi$. On the other hand, from rule CFLOW we see that for $\mathbb{m} \models_{\mathsf{R}} \mathring{p} : \langle \Gamma, \chi \rangle$ to hold there has to be a subtyping edge from $\mathbb{m}(\mathring{p})$ to χ. It suggests that some process originally matching Γ at node $\mathbb{m}(\mathring{p})$ was moved by a rewriting rule to the new position corresponding to χ. Rule CSUB is a generalized form of CFLOW and it describes process movements which additionally involve substitution application.

The following example demonstrates the use of the relation $\mathbb{m} \models_{\mathsf{R}} \mathring{Q} : \Pi$.

EXAMPLE 7.6.4. *Let us take the right-hand side of the* $\mathcal{P}_{\mathsf{async}}$ *communication rule "*$\mathring{Q} = \{\mathring{\mathtt{x}} := \mathring{\mathtt{a}}\}\mathring{\mathtt{Q}}$*". Let the type instantiation "*$\mathbb{m} = \{\mathring{\mathtt{c}} \mapsto \mathtt{c}, \mathring{\mathtt{a}} \mapsto \mathtt{c}, \mathring{\mathtt{x}} \mapsto \mathtt{x}, \mathring{\mathtt{Q}} \mapsto \mathtt{Y0}\}$*"*

and graph Γ_0 *be as in Example 7.6.3. We can see that* $\mathbb{T} \not\models_{\mathtt{R}} \mathring{Q} : \langle \Gamma_0, \mathtt{R} \rangle$ *because there is no flow edge from* `Y0` *to* `R`*. Let us construct the shape predicate* $\Pi_1 = \langle \Gamma_1, \mathtt{R} \rangle$ *with the required edge added, where* Γ_1 *is as follows.*

$\Gamma_1 =$



Now $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \Pi_1$ *holds. The new flow edge describes that the communication can apply a substitution of type* $\mathfrak{o} = \{\mathtt{x} \mapsto \mathtt{c}\}$ *to some process that matches* $\langle \Gamma_1, \mathtt{Y0} \rangle$ *and move the resulting process to the root position* `R`*. The design goal of* $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \Pi_1$*, that the process* $\mathbb{P}[\![\mathring{Q}]\!]$ *matches* Π_1 *when* $\mathbb{P}$ *respects* $\mathbb{T}$ *on* Γ_1*, is not satisfied because* Π_1 *is not flow-closed. With the process instantiation "*$\mathbb{P} = \{\mathring{\mathtt{c}} \mapsto \mathtt{c}, \mathring{\mathtt{a}} \mapsto \mathtt{c}, \mathring{\mathtt{x}} \mapsto \mathtt{x}, \mathring{\mathtt{Q}} \mapsto \mathtt{x(y).y<x>.0}\}$*" from Example 7.6.3 we can indeed see that* $\mathbb{P}[\![\mathring{Q}]\!] = \mathtt{c(y).y<c>.0}$ *does not match* Π_1*. We will conclude this example below in Example 7.6.10.* ▫

The relations $\mathbb{T} \models_{\mathtt{L}} \mathring{P} : \chi$ and $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \chi$ are used to apply a rewriting rule $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ directly to a shape graph Γ at node χ as follows. Firstly, we find all possible type instantiations $\mathbb{T}$ such that $\mathbb{T} \models_{\mathtt{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. There are only finitely many $\mathbb{T}$'s like that with $\mathrm{dom}(\mathbb{T}) = \mathrm{var}(\mathring{P})$ (see Section 11.4.1 for an effective algorithm). Secondly, we check that $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$. We have seen above that for $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$ to hold an existence of some additional edges which are not in Γ might be required. In this way application of a rule to a shape graph can insist on existence of new edges. We call a shape graph Γ *locally* $\mathcal{R}$*-closed at* χ when the edges required by application of any rewriting rule from $\mathcal{R}$ to $\langle \Gamma, \chi \rangle$ are already present in $\langle \Gamma, \chi \rangle$.

Some formal properties of type instantiations are formulated and proved in Section 8.6.

Definition 7.6.5. *The shape graph* Γ *is called* **locally** $\mathcal{R}$**-closed at** χ *iff for any* $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *it holds that* $\mathbb{T} \models_{\mathtt{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *implies* $\mathbb{T} \models_{\mathtt{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$*.* ▪

We will shortly define $\mathcal{R}$-types to be flow-closed shape predicates which are locally $\mathcal{R}$-closed at every node where rewriting rules can be applied. What remains is to determine this set of *active nodes* where rewriting rules need to be applied. This set is given by the **active** rules from $\mathcal{R}$. For the rule sets without any active rule, like for example $\mathcal{P}_{\mathsf{async}}$ above, there is only one active position in a shape predicate,

that is, the root node. For every node χ and Γ we define the set $\mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi)$ of active successors of χ in Γ with respect to the **active** rules in $\mathcal{R}$ as follows.

DEFINITION 7.6.6. *The set* $\mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi)$ *of* **active successors of** χ **in** Γ **w.r.t.** $\mathcal{R}$ *is defined to be*

$$\mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi) = \{\mathbb{m}(\mathring{p}) \colon \mathbf{active}\{\, \mathring{p} \;\mathbf{in}\; \mathring{P} \,\} \in \mathcal{R} \;\&\; \mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle\}.$$

Using the above definition it is easy to determine the set of nodes of a shape predicate Π where rewriting rules $\mathcal{R}$ need to be applied. It is the least node set that contains the root of Π and that is closed for active successors.

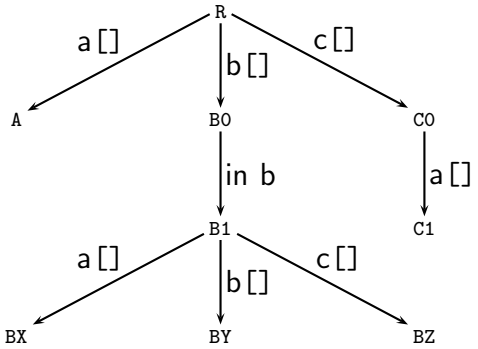
DEFINITION 7.6.7. *Let* $\Pi = \langle \Gamma, \chi \rangle$. *The set* $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ *of* **active nodes of** Π **w.r.t.** $\mathcal{R}$ *is the least node set* Ξ *such that*

(1) $\chi \in \Xi$, *and*
(2) for every $\chi_0 \in \Xi$ *it holds that* $\mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi_0) \subseteq \Xi$. ∎

The following example demonstrates the definition of active nodes.

EXAMPLE 7.6.8. *Let us consider the monadic Mobile Ambients rules* $\mathcal{A}_{\mathsf{mon}}$ *from Section 5.3 and the shape predicate* $\Pi = \langle \Gamma, \mathtt{R} \rangle$ *where* Γ *is the following graph.*

$\Gamma =$



The only rule important for this example is $\mathbf{active}\{\, \mathring{\mathtt{P}} \;\mathbf{in}\; \mathring{\mathtt{a}}[\mathring{\mathtt{P}}] \,\} \in \mathcal{A}_{\mathsf{mon}}$. *It is easy to compute the set* $\mathrm{ActiveNode}_{\mathcal{A}_{\mathsf{mon}}}(\Pi) = \{\mathtt{R}, \mathtt{A}, \mathtt{B0}, \mathtt{C0}, \mathtt{C1}\}$. *Note that it does not contain* BX, BY, *and* BZ *because node* B1 *is not active. On the other hand we can see that* $\mathrm{ActiveNode}_{\mathcal{A}_{\mathsf{mon}}}(\langle \Gamma, \mathtt{B1} \rangle) = \{\mathtt{B1}, \mathtt{BX}, \mathtt{BY}, \mathtt{BZ}\}$. ▫

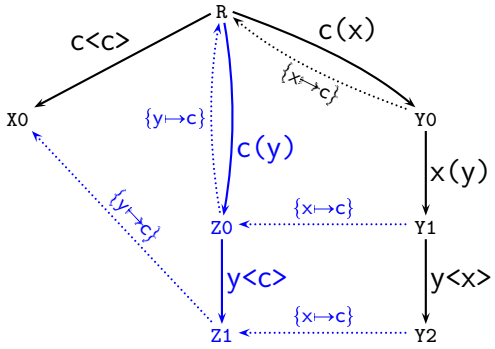
Now we can finally define $\mathcal{R}$-types to be flow-closed shape predicates locally $\mathcal{R}$-closed at all active nodes.

DEFINITION 7.6.9 ($\mathcal{R}$-TYPE). *A shape predicate* $\Pi = \langle \Gamma, \chi \rangle$ *is an* $\mathcal{R}$**-type**, *written* $\mathcal{R} \Vdash_{\mathsf{type}} \Pi$, *iff* Γ *is flow-closed and also locally* $\mathcal{R}$*-closed at every* $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. *When* $\mathcal{R} \Vdash_{\mathsf{type}} \Pi$ *and* $\vdash P : \Pi$ *we say that* Π *is an* $\mathcal{R}$*-type of* P. ∎

Now we can conclude the π-calculus examples 7.6.3 and 7.6.4 which demonstrated local closure.

Example 7.6.10. *Let us consider the shape predicate* $\Pi_2 = \langle \Gamma, \mathtt{R} \rangle$ *where* Γ_2 *is below.*

$\Gamma_2 =$



The shape predicate Π_2 *is obtained from* Π_1 *(from Example 7.6.4) by adding the blue edges. We can see that* Π_2 *is an* $\mathcal{P}_{\mathsf{async}}$*-type. It can be constructed from* Π_1 *as follows. At first we need to make* Π_1 *flow-closed and this can be done by adding four new edges* $\mathtt{R} \xrightarrow{\mathtt{c(y)}} \mathtt{Z0}$, $\mathtt{Z0} \xrightarrow{\mathtt{y<c>}} \mathtt{Z1}$, $\mathtt{Y1} \overset{\{\mathtt{x}\mapsto\mathtt{c}\}}{\cdots\!\!\rightarrow} \mathtt{Z0}$, *and* $\mathtt{Y2} \overset{\{\mathtt{x}\mapsto\mathtt{c}\}}{\cdots\!\!\rightarrow} \mathtt{Z1}$. *This makes* Π_1 *flow-closed but it is no longer locally* $\mathcal{P}_{\mathsf{async}}$*-closed at* R *because the new edge labeled with "*c(y)*" can interact with the one labeled with "*c<c>*". Hence the remaining edges are added to make the shape predicate an* $\mathcal{P}_{\mathsf{async}}$*-type. We can, for example, see that the meaning of* Π_2 *contains all the following processes.*

$$\mathtt{!c<c>.0 \mid c(x).x(y).y<x>.0} \xrightarrow{\mathcal{P}_{\mathsf{async}}} \mathtt{!c<c>.0 \mid c(y).y<c>.0} \xrightarrow{\mathcal{P}_{\mathsf{async}}} \mathtt{!c<c>.0}$$

The above definition gives us the algorithm to verify whether a given shape predicate is an $\mathcal{R}$-type. It does not give us, however, an algorithm to complete an arbitrary shape predicate to an $\mathcal{R}$-type. Chapter 11 describes the type inference algorithm which computes an $\mathcal{R}$-type of P for any $\mathcal{R}$ and P.

Subject reduction says that the meaning of an $\mathcal{R}$-type is closed under rewriting with $\mathcal{R}$, that is, that an $\mathcal{R}$-type is $\mathcal{R}$-closed. The proof is found in Section 8.7.

Theorem 7.6.11 (Subject reduction). *Every* $\mathcal{R}$*-type is* $\mathcal{R}$*-closed.* ∎

7.7 Spatial Polymorphism

In this section we describe *spatial polymorphism* which was briefly mentioned in Section 1.5. Spatial polymorphism, first described in the PolyA system, can be encountered in analysis systems for process calculi which place processes in a spatial structure like for example the ambient hierarchy in Mobile Ambients. Type systems for Mobile Ambients usually restrict the possible content that an ambient can hold, for example, they restrict communication actions or capabilities that can be executed inside the ambient. Sometimes it is reasonable to allow the same ambient to hold different content when it is found on different positions in the spatial hierarchy. Spatial polymorphism is the ability of an analysis system to describe this behavior.

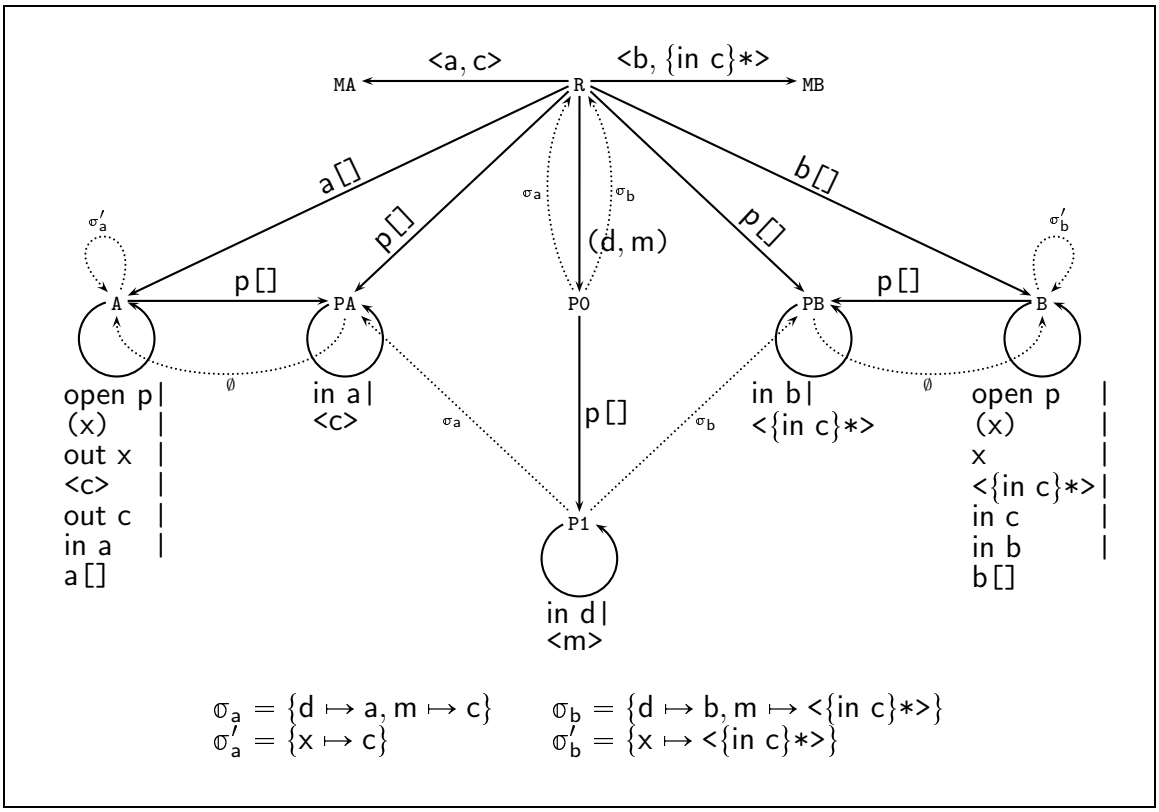


Figure 7.5: Spatial polymorphism on the example of a messenger ambient.

A typical example in Mobile Ambients is a *messenger* ambient which delivers a message to a given destination ambient. Let us consider the following Mobile Ambients process.

$$(\mathsf{d}, \mathsf{m}).\mathsf{p}[\text{in d.<m>.0}]$$

This process receives the name d of a destination ambient and a message m to be delivered, and creates the packet ambient p which will deliver and transmit the message. As the message is re-transmitted inside the ambient p the destination ambient d is expected to open it using "open p". Now the content allowed inside ambient p depends on its destination d. The delivered message m should have the type that is expected by the destination d and thus we can not possibly restrict the content of ambient p before we know the destination d.

The following example shows the above messenger process in action. The messenger process delivers the message c to ambient a and also the message "in c" to ambient b.

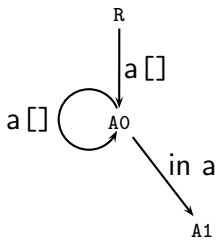
$$\begin{array}{l} !(\mathsf{d}, \mathsf{m}).\mathsf{p}[\text{in d.<m>.0}] \;\; | \\ \quad \langle \mathsf{a}, \mathsf{c} \rangle.0 \quad | \;\; \mathsf{a}[\text{open p.(x).out x.0}] \;\; | \\ \quad \langle \mathsf{b}, \text{in c} \rangle.0 \;\; | \;\; \mathsf{b}[\text{open p.(x).x.0}] \end{array}$$

An $\mathcal{A}_{\mathsf{mon}}$-type of the above process is shown in Figure 7.5. Edges with the same source and destination are shown as one edge with a label that joins the labels of individual edges using "|". Some flow edges which are not crucial are omitted in the graph in order to improve readability. We can see that the type proves, for example,

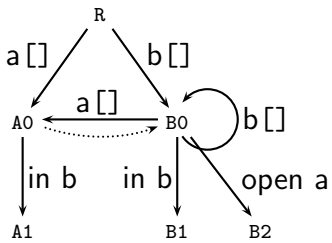
that the ambient p can never transmit the message <c> when p is inside b but it can transmit this message when p is inside a. Thus we can use the type to prove that the delivered messages have the right type, that is, the type expected by the destination ambient (the ambient a expects a name while the ambient b expects a capability sequence).

7.8 The in/open Anomaly

By the name "in/open anomaly" we refer to an over-approximation that happens in shape types for Mobile Ambients and similar systems which contain the in and open capabilities. The problem can be simply described on the process "a[in a.0]" considered together with $\mathcal{A}_{\mathsf{mon}}$ Mobile Ambients rules. This process is inert in Mobile Ambients and we would like to describe it simply by the shape predicate "$\Pi_0 = \langle\{\mathtt{R} \xrightarrow{\mathtt{a[]}} \mathtt{A0}, \mathtt{A0} \xrightarrow{\mathtt{in\ a}} \mathtt{A1}\}, \mathtt{R}\rangle$". Unfortunately it turns out that Π_0 is not an $\mathcal{A}_{\mathsf{mon}}$-type because Π_0 has to also describe the process "!a[!in a.0]" because replication is described implicitly in shape types. This replicated process is, however, no longer inert and it can evolve to a process with arbitrarily many nested copies of a like "a[a[a[$\cdots$]]]". Thus any shape type of "a[in a.0]" has to contain the loop or cycle of edges labeled with "a[]". One possible $\mathcal{A}_{\mathsf{mon}}$-type of the above process is the following (with the root R).



The same kind of over-approximation is encountered in a very common situation where an ambient a enters ambient b and in the next step b opens a. For example, the best shape $\mathcal{A}_{\mathsf{mon}}$-type for the process[1] "a[in b.0] | b[open a.0]" looks as follows (unimportant flow edges are omitted).



Basically, no shape $\mathcal{A}_{\mathsf{mon}}$-type of "a[in b.0] | b[open a.0]" can avoid the over-approximation caused by the loop $\mathtt{B0} \xrightarrow{\mathtt{b[]}} \mathtt{B0}$ from the same reason as above. That

[1] This example process motivates the name of the "in/open" anomaly.

is, because the type has to describe also the process "!a[!in b.0] | !b[open a.0]".

The "in/open" anomaly was firstly described by Amtoft and Wells [AW02] in the system which motivates POLY✶. One possible solution to deal with the anomaly would be to equip edges in shape graphs with natural numbers (extended with ω for infinity) which would determine how many times a single edge in a type can be used when matching a process. Although this *counting* would eliminate over-approximation in the case of the above process "a[in a.0]", it would not completely eliminate over-approximation in the case of "a[in b.0] | b[open a.0]". To demonstrate this let us consider even a simpler process "a[in b.0] | b[0]". Let us equip edges in shape graphs with natural number superscripts with the meaning described above and let us try to close the shape predicate with root R and the following shape graph.

$$\{(\mathtt{R} \xrightarrow{\mathtt{a[]}} \mathtt{A1})^1, (\mathtt{A1} \xrightarrow{\mathtt{in\ b}} \mathtt{A2})^1, (\mathtt{R} \xrightarrow{\mathtt{b[]}} \mathtt{B1})^1\}$$

The above is the smallest shape predicate that matches "a[in b.0] | b[0]". Now the in rule can be applied and it moves ambient a into b. The shape graph changes to the following.

$$\{(\mathtt{R} \xrightarrow{\mathtt{a[]}} \mathtt{A1})^1, (\mathtt{A1} \xrightarrow{\mathtt{in\ b}} \mathtt{A2})^1, (\mathtt{R} \xrightarrow{\mathtt{b[]}} \mathtt{B1})^1, (\mathtt{B1} \xrightarrow{\mathtt{a[]}} \mathtt{B2})^1\}$$

At this point counting helped us to recognize that the "in b" capability has been consumed by the rule application and thus there is no edge labeled by "in b" outgoing from B2. However there is no indication in the shape graph that the in rule has already been applied. The new shape predicate can also match the process "a[in b.0] | b[a[0]]" and thus we need to apply the in rule again which increases the index of $(\mathtt{B1} \xrightarrow{\mathtt{a[]}} \mathtt{B2})$ to 2. The situation repeats again and finally we end up with the shape predicate with root R and the following shape graph.

$$\{(\mathtt{R} \xrightarrow{\mathtt{a[]}} \mathtt{A1})^1, (\mathtt{A1} \xrightarrow{\mathtt{in\ b}} \mathtt{A2})^1, (\mathtt{R} \xrightarrow{\mathtt{b[]}} \mathtt{B1})^1, (\mathtt{B1} \xrightarrow{\mathtt{a[]}} \mathtt{B2})^\omega\}$$

This shape predicate is the best shape type of "a[in b.0] | b[0]" and thus we see that even counting has not eliminated the over-approximation because of the lack of temporal information in shape graphs.

Chapter 8

Technical Details on Poly✶ and Subject Reduction

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later,

either the whole chapter or just some particular part.

8.1 Properties of Basic Poly✶ Types

Figure 8.1 defines sets of free and input-bound type tags of Poly✶ type entities. Next we define meaning of basic Poly✶ type entities. Moreover we define a subtyping relation on type entities and we prove its basic properties.

Definition 8.1.1. *The* **meaning** $[\![\cdot]\!]$ *of* Poly✶ *type entities is defined as follows.*

$$[\![\iota]\!] = \{x\colon (\vdash x : \iota)\} \qquad [\![\sigma]\!] = \{s\colon (\vdash s : \sigma)\} \qquad [\![\Sigma]\!] = \{M\colon (\vdash M : \Sigma)\}$$
$$[\![\mu]\!] = \{M\colon (\vdash M : \mu)\} \quad [\![\varepsilon]\!] = \{E\colon (\vdash E : \varepsilon)\} \quad [\![\varphi]\!] = \{F\colon (\vdash F : \varphi)\}$$

Definition 8.1.2. *The* **subtyping relation** *on* Poly✶ *type entities* ζ *is defined as* $\zeta_0 \preccurlyeq \zeta_1$ *iff* $[\![\zeta_0]\!] \subseteq [\![\zeta_1]\!]$ *where* ζ *range over* ι, σ, Σ, μ, ε, *and* φ. ∎

The ordering $\preccurlyeq$ is well founded for all basic type entities, that is, for any ζ_1 there is only finitely many types ζ_0 such that $\zeta_0 \preccurlyeq \zeta_1$. This will not be true for subtyping on shape predicates which is introduced later. For any σ_0 and σ_1 we can see that $\sigma_0 \preccurlyeq \sigma_1$ holds if and only if $\sigma_0 = \sigma_1$. The same holds for any basic type entities that do not contain any sequence type set Σ (including the empty set).

Now we prove basic properties of the subtyping relation on basic Poly✶ type entities. Additional properties about the correspondence of the subtyping relation and type substitutions are formulated and proved in Section 8.3. At first we prove a close correspondence between subtyping relation on sequence type sets and starred message types of the shape $\Sigma*$.

$$
\begin{array}{ll}
\mathrm{ftags}(\iota_0 \ldots \iota_k) = \{\iota_0, \ldots, \iota_k\} & \mathrm{itags}(\iota_0 \ldots \iota_k) = \emptyset \\
\mathrm{ftags}(\Sigma) = \bigcup_{\sigma \in \Sigma} \mathrm{ftags}(\sigma) & \mathrm{itags}(\Sigma) = \emptyset \\
\mathrm{ftags}(\iota) = \{\iota\} & \mathrm{itags}(\iota) = \emptyset \\
\mathrm{ftags}(\Sigma\ast) = \bigcup_{\sigma \in \Sigma} \mathrm{ftags}(\sigma) & \mathrm{itags}(\Sigma\ast) = \emptyset \\
\mathrm{ftags}(\mathsf{(}\iota_1, \ldots, \iota_k\mathsf{)}) = \emptyset & \mathrm{itags}(\mathsf{(}\iota_1, \ldots, \iota_k\mathsf{)}) = \{\iota_1, \ldots, \iota_k\} \\
\mathrm{ftags}(\mathsf{<}\mu_1, \ldots, \mu_k\mathsf{>}) = \bigcup_{i=1}^{k} \mathrm{ftags}(\mu_i) & \mathrm{itags}(\mathsf{<}\mu_1, \ldots, \mu_k\mathsf{>}) = \emptyset \\
\mathrm{ftags}(\varepsilon_0 \ldots \varepsilon_k) = \bigcup_{i=0}^{k} \mathrm{ftags}(\varepsilon_i) & \mathrm{itags}(\varepsilon_0 \ldots \varepsilon_k) = \bigcup_{i=0}^{k} \mathrm{itags}(\varepsilon_i) \\
\mathrm{ftags}(\chi_0 \xrightarrow{\varphi} \chi_1) = \mathrm{ftags}(\varphi) & \mathrm{itags}(\chi_0 \xrightarrow{\varphi} \chi_1) = \mathrm{itags}(\varphi) \\
\mathrm{ftags}(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_1) = \mathrm{dom}(\mathfrak{o}) \cup \bigcup_{\mu \in \mathrm{rng}(\mathfrak{o})} \mathrm{ftags}(\mu) & \mathrm{itags}(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_1) = \emptyset \\
\mathrm{ftags}(\Gamma) = \bigcup_{\eta \in \Gamma} \mathrm{ftags}(\eta) & \mathrm{itags}(\Gamma) = \bigcup_{\eta \in \Gamma} \mathrm{itags}(\eta) \\
\mathrm{ftags}(\langle \Gamma, \chi \rangle) = \mathrm{ftags}(\Gamma) & \mathrm{itags}(\langle \Gamma, \chi \rangle) = \mathrm{itags}(\Gamma)
\end{array}
$$

Figure 8.1: Free and (input-) bound type tags of POLY✶ type entities.

LEMMA 8.1.3. *The following holds.*

$$\Sigma \preccurlyeq \Sigma' \quad \textit{iff} \quad \Sigma\ast \preccurlyeq \Sigma'\ast \quad \textit{iff} \quad \Sigma \subseteq \Sigma'$$

PROOF. *We know that* $[\![\Sigma_0\ast]\!] = ([\![\Sigma_0]\!] \backslash \mathrm{Name})$ *for any* Σ_0 *and thus it is clear that* $\Sigma \preccurlyeq \Sigma'$ *implies* $\Sigma\ast \preccurlyeq \Sigma'\ast$. *Let us prove the opposite implication. Let* $\Sigma\ast \preccurlyeq \Sigma'\ast$ *and let* $\vdash M : \Sigma$. *We need to prove* $\vdash M : \Sigma'$. *When* $M \notin \mathrm{Name}$ *then* $\vdash M : \Sigma\ast$ *and by the assumption* $\vdash M : \Sigma'\ast$ *and thus* $\vdash M : \Sigma'$. *When* $M = x \in \mathrm{Name}$ *we know that* $\vdash x.x : \Sigma$ *and thus* $\vdash x.x : \Sigma'\ast$ *and* $\vdash x.x : \Sigma'$ *as above. But this implies the claim* $\vdash x : \Sigma'$.

Now it is enough to prove that $\Sigma \preccurlyeq \Sigma'$ *iff* $\Sigma \subseteq \Sigma'$. *Let us prove the "*$\Rightarrow$*" implication. Let* $\Sigma \preccurlyeq \Sigma'$ *and let* $\sigma \in \Sigma$. *There is* s *such that* $\vdash s : \sigma$ *and thus* $\vdash s : \Sigma$. *By the assumption we obtain* $\vdash s : \Sigma'$ *which implies that there is some* σ' *such that* $\vdash s : \sigma'$ *and* $\sigma' \in \Sigma'$. *It is easy to see that* $\vdash s : \sigma$ *and* $\vdash s : \sigma'$ *imply that* $\sigma = \sigma'$ *and thus* $\sigma \in \Sigma'$. *To prove the opposite "*$\Leftarrow$*" implication let* $\Sigma \subseteq \Sigma'$ *and* $\vdash M : \Sigma$. *By an easy induction on the structure of* M *we prove* $\vdash M : \Sigma'$. *Hence the claim.* ∎

The following lemma expresses the property that when one of the message types in $\mu \preccurlyeq \mu'$ is a type tag then the second message type is the very same type tag.

LEMMA 8.1.4. *Let* $\mu \preccurlyeq \mu'$. *Then*

(1) $\mu \in \mathrm{TypeTag}$ *iff* $\mu' \in \mathrm{TypeTag}$, *and*

(2) when $\mu = \iota \in \mathrm{TypeTag}$ *then* $\mu' = \iota = \mu$.

PROOF. *Let* $\mu \preccurlyeq \mu'$. *Firstly let us prove the "*$\Rightarrow$*" direction of (1) and (2). Let* $\mu = \iota \in \mathrm{TypeTag}$. *We know* $\vdash \mathsf{a}^{\iota} : \iota$ *and thus* $\vdash \mathsf{a}^{\iota} : \mu'$. *Clearly* μ' *can not have the*

shape $\Sigma*$ *because a single name can not have a* $\Sigma*$ *type. Thus* μ' *has to be a type tag hence it has to be* ι.

Secondly let us prove the "$\Leftarrow$*" direction of (1). Let* $\mu' = \iota \in \mathrm{TypeTag}$. *We know that there is some* M *such that* $\vdash M : \mu$ *because* $[\![\mu]\!] \neq \emptyset$. *Thus* $\vdash M : \mu'$ *which implies that* $M = a^\iota$ *for some* a *because* $\mu' = \iota$. *Thus* $\vdash a^\iota : \mu$ *and hence* $\mu = \iota \in \mathrm{TypeTag}$ *because a single name can not have a* $\Sigma*$ *type.* ∎

Similar lemma as the previous one holds also for form types.

LEMMA 8.1.5. *Let* $\varphi \preccurlyeq \varphi'$. *Then*

(1) $\varphi \in \mathrm{TypeTag}$ *iff* $\varphi' \in \mathrm{TypeTag}$, *and*

(2) when $\varphi = \iota \in \mathrm{TypeTag}$ *then* $\varphi' = \iota = \varphi$.

PROOF. *Let* $\varphi \preccurlyeq \varphi'$. *Firstly let us prove the "*$\Rightarrow$*" direction of (1) and (2). Let* $\varphi = \iota \in \mathrm{TypeTag}$. *We know* $\vdash \mathsf{a}^\iota : \iota$ *and thus* $\vdash \mathsf{a}^\iota : \varphi'$. *Thus* φ' *has to be a single element type and hence it has to be* ι.

Secondly let us prove the "$\Leftarrow$*" direction of (2). Let* $\varphi' = \iota \in \mathrm{TypeTag}$. *We know that there is some* F *such that* $\vdash F : \varphi$ *because* $[\![\varphi]\!] \neq \emptyset$. *Thus* $\vdash F : \varphi'$ *which implies that* $F = a^\iota$ *for some* a. *Thus* $\vdash a^\iota : \varphi$ *and hence* $\varphi = \iota \in \mathrm{TypeTag}$. ∎

The following lemma says that only form types with the same sets of input-bound tags are related by the subtyping relation.

LEMMA 8.1.6. *When* $\varphi \preccurlyeq \varphi'$ *then* $\mathrm{itags}(\varphi) = \mathrm{itags}(\varphi')$.

PROOF. *It is easy to see that* $\vdash E : \varepsilon$ *implies* $\mathrm{itags}(E) = \mathrm{itags}(\varepsilon)$ *for any* E *and* ε. *Thus* $\vdash F_0 : \varphi_0$ *implies* $\mathrm{itags}(F_0) = \mathrm{itags}(\varphi_0)$ *for any* F_0 *and* φ_0. *Let* $\varphi \preccurlyeq \varphi'$. *We know that* $[\![\varphi]\!] \neq \emptyset$ *and thus there is some* F *such that* $\vdash F : \varphi$. *The assumption implies that* $\vdash F : \varphi'$. *Thus* $\mathrm{itags}(\varphi) = \mathrm{itags}(F) = \mathrm{itags}(\varphi')$. ∎

8.2 Type Substitution Correctness

As stated above we want to prove the property that $\vdash \mathbb{S} : \mathfrak{o}$ and $\vdash F : \varphi$ implies $\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$. The following remark discusses additional conditions required for this to hold and that these conditions will always be satisfied when we work with well formed entities only.

REMARK 8.2.1. Let $\vdash \mathbb{S} : \mathfrak{o}$ and $\vdash F : \varphi$. To demonstrate the additional conditions required for $\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$ to hold let us take $\mathbb{S} = \{\mathsf{x}^\mathsf{x} \mapsto \mathsf{a}^\mathsf{a}\}$ and $\mathfrak{o} = \{\mathsf{x} \mapsto \mathsf{a}\}$. We see $\vdash \mathbb{S} : \mathfrak{o}$ and we also know that $\vdash \mathsf{y}^\mathsf{x} : \mathsf{x}$. But now $\bar{\mathbb{S}}\mathsf{y}^\mathsf{x} = \mathsf{y}^\mathsf{x}$ and $\bar{\mathfrak{o}}\mathsf{x} = \mathsf{a}$ and thus we obtain $\not\vdash \bar{\mathbb{S}}\mathsf{y}^\mathsf{x} : \bar{\mathfrak{o}}\mathsf{x}$. The problem here is that y^x which is not contained in $\mathrm{dom}(\mathbb{S})$ has

type tag x which is contained in $\mathrm{dom}(\mathfrak{o})$. From this we can formulate a necessary condition for $\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$ to hold to be

$$\text{for any } x \in \mathrm{dom}(\mathbb{S}) \text{ and } y \in \mathrm{fn}(F) : \overline{x} = \overline{y} \text{ implies } x = y.$$

For any $a^{\iota} \in \mathrm{dom}(\mathbb{S})$, it ensures that a^{ι} is the only name with type tag ι that can occur free in F. In practice this condition will be implied by well-formedness of processes for all substitution applications executed by rewriting rules. The only situation in META✶ that results in the application of a substitution is when the right-hand side $\mathring{Q}$ of some rewriting rule **rewrite**$\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\}$ contains a substitution application template, for example "$\{\mathring{x} := \mathring{s}\}\ \mathring{p}$". Then we see that it holds $\mathring{Q} \vdash_{\exists} \mathring{x} \gtrdot \mathring{p}$ and thus in well formed rules it has to hold that $\mathring{P} \vdash_{\exists} \mathring{x} \gtrdot \mathring{p}$. The substitution at the right-hand side of some rule is applied only to (the instantiation of) $\mathring{p}$. Thus the required condition for any F from (the instantiation of) $\mathring{p}$ is implied by Lemma 6.3.4 which was discussed in Remark 6.3.3. This issue is important for subject reduction and it is further discussed in Remark 8.5.2 ∎

Now we are ready to prove the following lemma which proves the property discussed above that applications of type substitutions to type entities faithfully describes applications of META✶ substitutions to process entities. We call this property "type substitution correctness" because it proves that application of type substitutions to various type entities is defined as expected.

PROPOSITION 8.2.2 (TYPE SUBSTITUTION CORRECTNESS). *Let* $\vdash \mathbb{S} : \mathfrak{o}$. *Let* Z *range over* $\{x, M, E, F\}$ *and let for any* $x_0 \in \mathrm{fn}(Z)$ *and* $y_0 \in \mathrm{dom}(\mathbb{S})$, $\overline{x_0} = \overline{y_0}$ *imply* $x_0 = y_0$. *Then all the following hold.*

(1) $\vdash x : \iota$ *implies* $\vdash \bar{\mathbb{S}}x : \bar{\mathfrak{o}}\iota$

(2) $\vdash x : \iota$ *implies* $\vdash \dot{\mathbb{S}}x : \ddot{\mathfrak{o}}\iota$

(3) $\vdash M : \mu$ *implies* $\vdash \dot{\mathbb{S}}M : \dot{\mathfrak{o}}\mu$

(4) $\vdash E : \varepsilon$ *implies* $\vdash \bar{\mathbb{S}}E : \bar{\mathfrak{o}}\varepsilon$

(5) $\vdash F : \varphi$ *implies* $\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$

PROOF. *Let* $\vdash \mathbb{S} : \mathfrak{o}$. *Let for any* $x_0 \in \mathrm{fn}(Z)$ *and* $y_0 \in \mathrm{dom}(\mathbb{S})$, $\overline{x_0} = \overline{y_0}$ *imply* $x_0 = y_0$ *where* Z *is* x *in cases (1) & (2) below,* Z *is* M *in case (3),* Z *is* E *in case (4), and* Z *is* F *in case (5).*

(1) Let $\vdash x : \iota$. *Thus* $\overline{x} = \iota$. *Distinguish the following three cases. When*

$\mathbb{S}(x) \in \mathrm{Name}$**:** *Then* $\bar{\mathbb{S}}x = \mathbb{S}(x)$. *Now* $\iota \in \mathrm{dom}(\mathfrak{o})$ *because* $\vdash \mathbb{S} : \mathfrak{o}$ *and* $x \in \mathrm{dom}(\mathbb{S})$. *Moreover* $\mathfrak{o}(\iota) \in \mathrm{TypeTag}$ *because* $\vdash \mathbb{S}(x) : \mathfrak{o}(\iota)$. *Thus* $\bar{\mathfrak{o}}\iota = \mathfrak{o}(\iota)$. *Hence the claim* $\vdash \bar{\mathbb{S}}x : \bar{\mathfrak{o}}\iota$.

$x \notin \mathrm{dom}(\mathbb{S})$**:** *Let us prove* $\iota \notin \mathrm{dom}(\mathfrak{o})$ *by contradiction. When* $\iota \in \mathrm{dom}(\mathfrak{o})$ *then there is some* $y = a^{\iota} \in \mathrm{dom}(\mathbb{S})$ *(because* $\vdash \mathbb{S} : \mathfrak{o}$*). Hence* $\overline{x} = \overline{y}$ *and thus* $x = y$ *by the assumption. But then* $x \in \mathrm{dom}(\mathbb{S})$ *and hence contradiction. Thus it has to hold* $\iota \notin \mathrm{dom}(\mathfrak{o})$*. Then* $\bar{\mathbb{S}}x = x$ *and* $\bar{\mathfrak{o}}\iota = \iota$*. Hence the claim.*

otherwise: *We know that* $x \in \mathrm{dom}(\mathbb{S})$ *but* $\mathbb{S}(x) \notin \mathrm{Name}$*. Thus also* $\iota \in \mathrm{dom}(\mathfrak{o})$ *and obviously* $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$ *because we know that* $\vdash \mathbb{S}(x) : \mathfrak{o}(\iota)$ *holds. Thus* $\bar{\mathbb{S}}x = \bullet$ *(more precisely* $\bar{\mathbb{S}}x = \bullet^{\bullet}$*) and* $\bar{\mathfrak{o}}\iota = \bullet$*. Hence the claim.*

(2) Let $\vdash x : \iota$*. Thus* $\overline{x} = \iota$*. Distinguish the following three cases. Let*

$\mathfrak{o}(\iota) \in \mathrm{TypeTag}$**:** *Thus* $\iota \in \mathrm{dom}(\mathfrak{o})$*. There is some* $y = a^{\iota} \in \mathrm{dom}(\mathbb{S})$ *(because* $\vdash \mathbb{S} : \mathfrak{o}$ *and* $\iota \in \mathrm{dom}(\mathfrak{o})$*). Now* $\overline{x} = \overline{y}$ *and thus* $x = y$ *by the assumption of this lemma. Hence* $x \in \mathrm{dom}(\mathbb{S})$*. Thus* $\dot{\mathbb{S}}x = \mathbb{S}(x)$ *and* $\ddot{\mathfrak{o}}\iota = \{\bar{\mathfrak{o}}\iota\} = \{\mathfrak{o}(\iota)\}$*. We know that* $\vdash \mathbb{S}(x) : \mathfrak{o}(\iota)$ *and thus the claim is derived by the message typing rule with the premise* $\mathfrak{o}(\iota) \in \{\mathfrak{o}(\iota)\}$*.*

$\iota \in \mathrm{dom}(\mathfrak{o})$ & $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$**:** *There is some* Σ *such that* $\mathfrak{o}(\iota) = \Sigma\ast$*. We also know that* $x \in \mathrm{dom}(\mathbb{S})$*. Thus* $\dot{\mathbb{S}}x = \mathbb{S}(x)$*. But now* $\ddot{\mathfrak{o}}\iota = \Sigma$*. We know that* $\vdash \mathbb{S}(x) : \Sigma\ast$ *and thus* $\vdash \mathbb{S}(x) : \Sigma$*. Hence the claim.*

$\iota \notin \mathrm{dom}(\mathfrak{o})$**:** *Thus* $x \notin \mathrm{dom}(\mathbb{S})$*. Here we have that* $\dot{\mathbb{S}}x = x$ *and* $\ddot{\mathfrak{o}}\iota = \{\iota\}$*. Thus the claim is derived by the typing rule with the premise* $\iota \in \{\iota\}$*.*

(3) Let $\vdash M : \mu$*. Prove the claim by induction on the structure of* M*.*

$M = \mathtt{0}$**:** *Thus* $\dot{\mathbb{S}}M = \mathtt{0}$*. Because* $\vdash \mathtt{0} : \mu$ *we know that* $\mu = \Sigma\ast$ *for some* Σ*. Obviously* $\vdash \mathtt{0} : \dot{\mathfrak{o}}(\Sigma\ast)$ *as well because* $\dot{\mathfrak{o}}(\Sigma\ast)$ *is a starred message type.*

$M = s$**:** *Here* $s = x_0 \ldots x_k$*. Let*

$k = 0$**:** *Thus* $s = x$*. It means that* $\mu = \iota$ *for some* ι *and we have* $\vdash x : \iota$ *which means* $\overline{x} = \iota$*. When* $x \notin \mathrm{dom}(\mathbb{S})$ *then* $\iota \notin \mathrm{dom}(\mathfrak{o})$ *as well (proved as in the subcase of (1) where* $x \notin \mathrm{dom}(\mathbb{S})$*). Thus* $x \notin \mathrm{dom}(\mathbb{S})$ *implies* $\dot{\mathbb{S}}x = x$ *and* $\dot{\mathfrak{o}}\iota = \iota$ *which implies the claim. Now suppose that* $x \in \mathrm{dom}(\mathbb{S})$*. Hence* $\iota \in \mathrm{dom}(\mathfrak{o})$*. Thus* $\dot{\mathbb{S}}x = \mathbb{S}(x)$ *and* $\dot{\mathfrak{o}}\iota = \mathfrak{o}(\iota)$*. Hence the claim because* $\vdash \mathbb{S}(x) : \mathfrak{o}(\iota)$ *by* $\vdash \mathbb{S} : \mathfrak{o}$*.*

$k > 0$**:** *Here we know that* $\mu = \Sigma\ast$ *for some* $\Sigma = \{\sigma_1, \ldots, \sigma_l\}$*. Moreover there is some* $\sigma \in \Sigma$ *such that* $\vdash s : \sigma$*. We also know that* $s = x_0 \ldots x_k$ *and thus we can see that* $\sigma = \iota_1 \ldots \iota_k$ *where* $\vdash x_i : \iota_i$ *and thus* $\iota_i = \overline{x_i}$ *for* $i \in \{0, \ldots, k\}$*. Now we can see*

$$\dot{\mathbb{S}}M = \dot{\mathbb{S}}(x_0 \ldots x_k) = \bar{\mathbb{S}}(x_0 \ldots x_k) = (\bar{\mathbb{S}}x_0) \ldots (\bar{\mathbb{S}}x_k)$$
$$\dot{\mathfrak{o}}\mu = \dot{\mathfrak{o}}(\{\sigma_1, \ldots, \sigma_l\}\ast) = (\ddot{\mathfrak{o}}\sigma_1 \cup \cdots \cup \ddot{\mathfrak{o}}\sigma_l)\ast$$

For the above $\sigma = \iota_0 \ldots \iota_k \in \Sigma$ *we have that* $\ddot{\mathfrak{o}}\sigma = \{(\bar{\mathfrak{o}}\iota_0 \ldots \bar{\mathfrak{o}}\iota_k)\}$ *which is a singleton set. By the already proved point (1) of this lemma we*

have that $\vdash \bar{\mathbb{S}}x_i : \bar{\mathfrak{o}}\iota_i$ *for all* $i \in \{0, \ldots, k\}$ *(because* $\vdash x_i : \iota_i$*). Thus* $\vdash \bar{\mathbb{S}}x_0 \ldots \bar{\mathbb{S}}x_k : \bar{\mathfrak{o}}\iota_0 \ldots \bar{\mathfrak{o}}\iota_k$ *which gives us* $\vdash \dot{\mathbb{S}}M : \dot{\mathfrak{o}}\mu$ *as required because* $\bar{\mathfrak{o}}\iota_0 \ldots \bar{\mathfrak{o}}\iota_k \in \dot{\mathfrak{o}}\mu$*. Hence the claim.*

$M = M_0.M_1$**:** *Here we know that* $\mu = \Sigma*$ *for some* $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ *and also* $\vdash M : \Sigma$*. From the typing rules we know that it has to hold both* $\vdash M_0 : \Sigma$ *and* $\vdash M_1 : \Sigma$*. We have that* $\dot{\mathfrak{o}}\mu = (\ddot{\mathfrak{o}}\sigma_1 \cup \cdots \cup \ddot{\mathfrak{o}}\sigma_k)*$*. Let* $\Sigma' = (\ddot{\mathfrak{o}}\sigma_1 \cup \cdots \cup \ddot{\mathfrak{o}}\sigma_k)$*. Let us prove that* $\vdash \dot{\mathbb{S}}M_0 : \Sigma'$*.*

When $M_0 \notin$ Name *then* $\vdash M_0 : \mu$ *and thus by induction hypothesis we obtain* $\vdash \dot{\mathbb{S}}M_0 : \dot{\mathfrak{o}}\mu$ *which proves that* $\vdash \dot{\mathbb{S}}M_0 : \Sigma'$*. So now suppose that* $M_0 \in$ Name *and note that we can not use the induction hypothesis in this case. Let* $x = M_0$ *and* $\iota = \overline{x}$*. Thus we have* $\vdash x : \iota$*. Now because* $\vdash x : \Sigma$ *it has to hold that* $\iota \in \Sigma$*. But now we can see that* $\ddot{\mathfrak{o}}\iota \subseteq \Sigma'$ *because* $\ddot{\mathfrak{o}}\iota = \ddot{\mathfrak{o}}\sigma_i$ *for some* i*. Moreover from* $\vdash x : \iota$ *we obtain by the already proved point (2) of this lemma that* $\vdash \dot{\mathbb{S}}x : \ddot{\mathfrak{o}}\iota$*. And thus it has to hold also that* $\vdash \dot{\mathbb{S}}x : \Sigma'$ *which we wanted to prove.*

Thus we have $\vdash \dot{\mathbb{S}}M_0 : \Sigma'$*. Analogously we prove that* $\vdash \dot{\mathbb{S}}M_1 : \Sigma'$*. Thus* $\vdash \dot{\mathbb{S}}M_0.\dot{\mathbb{S}}M_1 : \Sigma'$ *and* $\vdash \dot{\mathbb{S}}(M_0.M_1) : \Sigma'*$*. Hence the claim.*

(4) Let $\vdash E : \varepsilon$*. Distinguish the following case by the structure of* E*. Let*

$E = x$**:** *We know that* $\vdash x : \varepsilon$ *and* $\varepsilon = \overline{x}$*. Thus the claim holds by the already proved point (1) of this lemma.*

$E = (x_1, \ldots, x_k)$**:** *Hence* $\varepsilon = (\iota_1, \ldots, \iota_k)$ *where we have* $\iota_i = \overline{x_i}$ *for all* $i \in \{1, \ldots, k\}$*. Now we see* $\bar{\mathbb{S}}E = E$ *and* $\bar{\mathfrak{o}}\varepsilon = \varepsilon$*. Hence the claim.*

$E = \langle M_1, \ldots, M_k \rangle$**:** *Here we have* $\varepsilon = \langle \mu_1, \ldots, \mu_k \rangle$ *and* $\vdash M_i : \mu_i$ *for all* $i \in \{1, \ldots, k\}$*. By the already proved point (2) of this lemma we have that* $\vdash \dot{\mathbb{S}}M_i : \dot{\mathfrak{o}}\mu_i$ *for the related* i*'s. Also we see that* $\bar{\mathbb{S}}M = \langle \dot{\mathbb{S}}M_1, \ldots, \dot{\mathbb{S}}M_k \rangle$ *and* $\bar{\mathfrak{o}}\mu = \langle \dot{\mathfrak{o}}\mu_1, \ldots, \dot{\mathfrak{o}}\mu_k \rangle$*. Hence the claim* $\vdash \bar{\mathbb{S}}M : \bar{\mathfrak{o}}\mu$*.*

(5) Let $\vdash F : \varphi$*. We have that* $F = E_0 \ldots E_k$*. Thus because* $\vdash F : \varphi$ *it has to hold that* $\varphi = \varepsilon_0 \ldots \varepsilon_k$ *and* $\vdash E_i : \varepsilon_i$ *for all* $i \in \{0, \ldots, k\}$*. Thus the previously proved point (4) of this lemma proves the claim.* ∎

8.3 Preservation of Subtyping Relation

In this section we prove another important property of type substitutions, namely that application of a type substitution (to POLY✶ type entities) preserves subtyping. This property will be necessary to prove existence of principal typings and correctness of the type inference algorithm. Firstly, we define subtyping on type substitutions.

DEFINITION 8.3.1. *Write* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$ *when*

(1) $\mathrm{dom}(\mathfrak{o}) = \mathrm{dom}(\mathfrak{o}')$*, and*

(2) $\mathfrak{o}(\iota) \preccurlyeq \mathfrak{o}'(\iota)$ *for all* $\iota \in \mathrm{dom}(\mathfrak{o})$*.* ∎

Now we prove that application of type substitutions $\mathfrak{o}$ and $\mathfrak{o}'$ such that $\mathfrak{o} \preccurlyeq \mathfrak{o}'$ preserves subtyping of various type entities.

LEMMA 8.3.2. *Let* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$*. Then all the following hold.*

(1) $\bar{\mathfrak{o}}\iota = \bar{\mathfrak{o}}'\iota$ *for any* ι

(2) $\ddot{\mathfrak{o}}\sigma \subseteq \ddot{\mathfrak{o}}'\sigma$ *for any* σ

(3) $\mu \preccurlyeq \mu'$ *implies* $\dot{\mathfrak{o}}\mu \preccurlyeq \dot{\mathfrak{o}}'\mu'$

(4) $\varepsilon \preccurlyeq \varepsilon'$ *implies* $\bar{\mathfrak{o}}\varepsilon \preccurlyeq \bar{\mathfrak{o}}'\varepsilon'$

(5) $\varphi \preccurlyeq \varphi'$ *implies* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$

PROOF. *Let* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$*.*

(1) We distinguish the following three cases.

$\iota \notin \mathrm{dom}(\mathfrak{o})$**:** *Then* $\iota \notin \mathrm{dom}(\mathfrak{o}')$ *and thus* $\bar{\mathfrak{o}}\iota = \iota = \bar{\mathfrak{o}}'\iota$*.*

$\iota \in \mathrm{dom}(\mathfrak{o})$ **and** $\mathfrak{o}(\iota) = \iota' \in \mathrm{TypeTag}$**:** *Then also* $\iota \in \mathrm{dom}(\mathfrak{o}')$*. We know that* $\mathfrak{o}(\iota) \preccurlyeq \mathfrak{o}'(\iota)$ *and thus by Lemma 8.1.4 we obtain that* $\mathfrak{o}'(\iota) \in \mathrm{TypeTag}$ *and* $\mathfrak{o}'(\iota) = \mathfrak{o}(\iota) = \iota'$*. Thus* $\bar{\mathfrak{o}}\iota = \iota' = \bar{\mathfrak{o}}'\iota$*.*

$\iota \in \mathrm{dom}(\mathfrak{o})$ **but** $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$**:** *Then also* $\iota \in \mathrm{dom}(\mathfrak{o}')$*. By Lemma 8.1.4 as above we obtain* $\mathfrak{o}'(\iota) \notin \mathrm{TypeTag}$*. Thus* $\bar{\mathfrak{o}}\iota = \bullet = \bar{\mathfrak{o}}'\iota$*.*

(2) Let $\sigma = \iota_0 \dots \iota_k$*. Let us distinguish the following cases.*

$k = 0$ **and** $\bar{\mathfrak{o}}\iota_0 = \Sigma\ast$**:** *Then* $\ddot{\mathfrak{o}}\sigma = \Sigma$*. We have* $\iota_0 \in \mathrm{dom}(\mathfrak{o})$ *and* $\iota_0 \in \mathrm{dom}(\mathfrak{o}')$*. Let* $\mu' = \mathfrak{o}'(\iota_0)$*. We know that* $\Sigma\ast \preccurlyeq \mu'$ *and thus by Lemma 8.1.4 we obtain that* μ' *can not be a type tag and thus* $\mu' = \Sigma'\ast$ *for some* Σ'*. From* $\Sigma\ast \preccurlyeq \Sigma'\ast$ *we obtain* $\Sigma \subseteq \Sigma'$ *by Lemma 8.1.3. Moreover we see that* $\bar{\mathfrak{o}}'\sigma = \ddot{\mathfrak{o}}'\iota_0 = \Sigma'$*. Hence* $\ddot{\mathfrak{o}}\sigma \subseteq \ddot{\mathfrak{o}}'\sigma$*.*

otherwise: *That is* $k > 0$ *or* $\bar{\mathfrak{o}}\iota_0 \in \mathrm{TypeTag}$*. Then* $\bar{\mathfrak{o}}\sigma = \{(\bar{\mathfrak{o}}\iota_0 \dots \bar{\mathfrak{o}}\iota_k)\}$*. Using Lemma 8.1.3 in the case when* $k = 0$ *we see that also* $\ddot{\mathfrak{o}}'\sigma = \{(\bar{\mathfrak{o}}'\iota_0 \dots \bar{\mathfrak{o}}'\iota_k)\}$*. The already proved case (1) of this lemma says that* $\bar{\mathfrak{o}}\iota_i = \bar{\mathfrak{o}}'\iota_i$ *for all* $i \in \{0, \dots, k\}$*. Hence* $\ddot{\mathfrak{o}}\sigma \subseteq \ddot{\mathfrak{o}}'\sigma$*.*

(3) Let $\mu \preccurlyeq \mu'$*. We distinguish the following two cases.*

$\mu = \iota$ **for some** ι**:** *Lemma 8.1.3 gives us that* $\mu' = \iota$*. When* $\iota \in \mathrm{dom}(\sigma)$ *then* $\iota \in \mathrm{dom}(\mathfrak{o}')$ *and we see that* $\dot{\mathfrak{o}}\mu = \mathfrak{o}(\iota)$ *and* $\dot{\mathfrak{o}}'\mu' = \mathfrak{o}'(\iota)$ *and the claim* $\dot{\mathfrak{o}}\mu \preccurlyeq \dot{\mathfrak{o}}'\mu'$ *follows from the assumption. When* $\iota \notin \mathrm{dom}(\mathfrak{o})$ *then* $\iota \notin \mathrm{dom}(\mathfrak{o}')$ *and we see that* $\dot{\mathfrak{o}}\mu = \iota = \dot{\mathfrak{o}}'\mu'$ *hence* $\dot{\mathfrak{o}}\mu \preccurlyeq \dot{\mathfrak{o}}'\mu'$*.*

$\mu = \Sigma*$ **for some** Σ**:** *By Lemma 8.1.3 we obtain that* $\mu' = \Sigma'*$ *for some* Σ' *and Lemma 8.1.3 gives us that* $\Sigma \subseteq \Sigma'$*. Let*

$$\Sigma_0 = \bigcup_{\sigma \in \Sigma} \ddot{\mathfrak{o}}\sigma \qquad \text{and} \qquad \Sigma'_0 = \bigcup_{\sigma \in \Sigma'} \ddot{\mathfrak{o}}'\sigma$$

We see that $\bar{\mathfrak{o}}\mu = (\Sigma_0)*$ *and* $\bar{\mathfrak{o}}'\mu' = (\Sigma'_0)*$*. From* $\Sigma \subseteq \Sigma'$ *using the already proved point (2) we obtain that* $\Sigma_0 \subseteq \Sigma'_0$*. By Lemma 8.1.3 we obtain the claim* $\dot{\mathfrak{o}}\mu \preccurlyeq \dot{\mathfrak{o}}'\mu'$*.*

(4) Let $\varepsilon \preccurlyeq \varepsilon'$*. Let us distinguish the following cases by the structure of* ε*.*

- $\varepsilon = \iota$**:** *Clearly* $\vdash \mathsf{a}^\iota : \iota$ *and thus* $\vdash \mathsf{a}^\iota : \varepsilon'$ *and thus* $\varepsilon' = \iota$*. Thus* $\bar{\mathfrak{o}}\varepsilon = \bar{\mathfrak{o}}\iota$ *and* $\bar{\mathfrak{o}}'\varepsilon' = \bar{\mathfrak{o}}'\iota$ *and the claim follows from the already proved point (1) of this lemma.*
- $\varepsilon = (\iota_1, \ldots, \iota_k)$**:** *It is easy to see that* $[\![\varepsilon]\!] \neq \emptyset$ *and thus* $\varepsilon' = (\iota_1, \ldots, \iota_k)$*. Clearly* $\bar{\mathfrak{o}}\varepsilon = (\iota_1, \ldots, \iota_k) = \bar{\mathfrak{o}}'\varepsilon'$ *and hence the claim.*
- $\varepsilon = \langle \mu_1, \ldots, \mu_k \rangle$**:** *It is easy to see that* $[\![\mu]\!] \neq \emptyset$ *and thus* $\varepsilon' = \langle \mu'_1, \ldots, \mu'_k \rangle$ *for some* μ'_1*, …,* μ'_k*. Let us prove that* $\mu_i \preccurlyeq \mu'_i$ *for all* $i \in \{1, \ldots, k\}$*. Let us fix* i *and let* $\vdash M_i : \mu_i$*. We need to prove that* $\vdash M_i : \mu'_i$*. We know that there are some* M_1*, …,* M_{i-1}*,* M_{i+1}*, …,* M_k *such that* $\vdash M_j : \mu_j$ *for all* $j \in \{1, 2, \ldots, i-1, i+1, \ldots, k-1, k\}$*. Hence* $\vdash \langle M_1, \ldots, M_k \rangle : \mu$ *and thus* $\vdash \langle M_1, \ldots, M_k \rangle : \mu'$*. Clearly* $\vdash M_i : \mu'_i$ *and thus* $\mu_i \preccurlyeq \mu'_i$*. By the already proved point (3) of this lemma we obtain that* $\dot{\mathfrak{o}}\mu_i \preccurlyeq \dot{\mathfrak{o}}'\mu'_i$ *for all* $i \in \{1, \ldots, k\}$*. Now we see that* $\bar{\mathfrak{o}}\varepsilon = \langle \dot{\mathfrak{o}}\mu_1, \ldots, \dot{\mathfrak{o}}\mu_k \rangle$ *and* $\bar{\mathfrak{o}}'\varepsilon' = \langle \dot{\mathfrak{o}}'\mu'_1, \ldots, \dot{\mathfrak{o}}'\mu'_k \rangle$*. Hence the claim* $\bar{\mathfrak{o}}\varepsilon \preccurlyeq \bar{\mathfrak{o}}'\varepsilon'$*.*

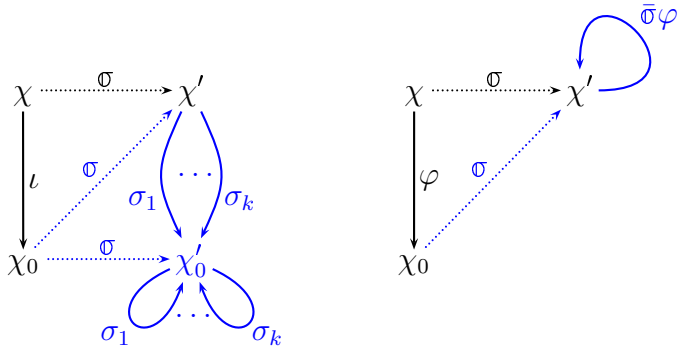
(5) Let $\varphi \preccurlyeq \varphi'$*. Let* $\varphi = \varepsilon_0 \ldots \varepsilon_k$*. Now* $[\![\varphi]\!] \neq \emptyset$ *and thus there is some* F *such that both* $\vdash F : \varphi$ *and* $\vdash F : \varphi'$*. It implies that* $\varphi' = \varepsilon'_0 \ldots \varepsilon'_k$ *for some* ε'_0*, …,* ε'_k *(that is,* φ *and* φ' *have the same length). We want to prove that* $\varepsilon_i \preccurlyeq \varepsilon'_i$ *for all* $i \in \{1, \ldots, k\}$*. Let us fix* i *and let* $\vdash E_i : \varepsilon_i$*. Now for any other* ε_j *where* $j \neq i$ *there always exists* E_j *such that* $\vdash E_j : \varepsilon_j$*. Thus we have* $\vdash \varepsilon_0 \ldots \varepsilon_k : \varepsilon$ *and hence* $\vdash \varepsilon_0 \ldots \varepsilon_k : \varepsilon'$ *and thus* $\vdash E_i : \varepsilon'_i$*. Thus* $\varepsilon_i \preccurlyeq \varepsilon'_i$ *for all* $i \in \{1, \ldots, k\}$ *and the already proved point (4) of this lemma gives us that* $\bar{\mathfrak{o}}\varepsilon_i \preccurlyeq \bar{\mathfrak{o}}'\varepsilon'_i$ *for all* $i \in \{1, \ldots, k\}$*. Now we see that* $\bar{\mathfrak{o}}\varphi = (\bar{\mathfrak{o}}\varepsilon_0) \ldots (\bar{\mathfrak{o}}\varepsilon_k)$ *and* $\bar{\mathfrak{o}}'\varphi' = (\bar{\mathfrak{o}}'\varepsilon'_1) \ldots (\bar{\mathfrak{o}}'\varepsilon'_k)$*. Hence the claim* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$*.* ∎

8.4 Details on Flow Closure

The example from Section 7.4 also shows why in the case of F1 the required blue edges have to be loops and not, for example, a sequence of edges. The reason is

that the starred message type $\Sigma*$ can match arbitrarily depth messages where a single sequence can repeat several times. Thus we can not construct all possible sequences of edges that would match all possible sequences paths $\mathbb{S}(\mathsf{x}^\iota)_*\mathsf{0}$ because there is no limit on the number of different paths. These loops result in some over-approximation in types which can be illustrated on the example from the previous paragraph as follows. Suppose that some other edge $\chi' \xrightarrow{\text{open a}} \chi'$ was present in Γ before the addition of the two blue loops and that the existence of this edge is not connected with the intended meaning of $(\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$ (perhaps it is required by some other flow edge). When we add the two blue loops then also the process "in a.open a.0" is added to the meaning of $\langle \Gamma, \chi' \rangle$. But it is clear that the process "in a.open a.0" does not need to be added to the meaning of $\langle \Gamma, \chi' \rangle$ at this step because no application of any substitution $\mathbb{S}$ such that $\vdash \mathbb{S} : \mathfrak{o}$ to $a^\iota.\mathsf{0}$ can result in "in a.open a.0".

Over-approximation in types can not be totally eliminated but we can make a modification F1' of rule F1 which reduces over-approximation and thus improves expressiveness of types as follows. To further illustrate this we also show alternative version F2' of F2 which adds a loop instead of an ordinary edge.



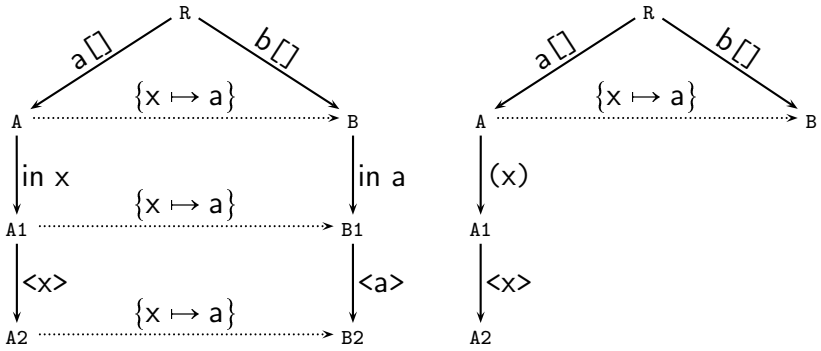
(F1') (F2')

We can see that the over-approximation described in the previous paragraph does not happen with the modified F1'. On the other hand this modification would make all flow-closed shape predicates approximately two times bigger because of the increased number of the new edges added. That is why we prefer F1 over F1'. Note that it is still necessary to add the flow edge $\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi'$ because $\mathfrak{o}(\iota) = \Sigma*$ can match the empty message $\mathsf{0}$. The alternative rule F2' would cause unnecessary over-approximation. Note that the number of new edges required by F2' is the same as in the case F2 and thus F2 does not result in bigger shape graphs. That is why we prefer rule F2. Here we just state that the intuitive meaning of flow edges would be implied when the definition of flow-closed graphs used the alternative rules F1' or F2'.

The condition from Definition 7.4.1 that $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ says that some edges can be excluded from the flow closure test. The presence of $\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi'$ in a shape graph Γ describes a possibility that some substitution $\mathbb{S}$ of the type $\mathfrak{o}$ can be applied

to some process P of the type $\langle \Gamma, \chi \rangle$. Now when some edge $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ is actually used to match P against $\langle \Gamma, \chi \rangle$ then it has to hold that $\mathrm{itags}(\varphi) \subseteq \mathrm{itags}(P)$. For example when $\varphi = (\mathsf{x})$ then we can deduce that P has the shape "$(a^{\mathsf{x}}).P_0$" for some a and P_0 and thus clearly $\mathsf{x} \in \mathrm{itags}(P)$. When $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\sigma) \neq \emptyset$ then the above $\mathbb{S}$ of type σ also have some name b^{x} in its domain $\mathrm{dom}(\mathbb{S})$. Thus the above possibility described by the presence of $\chi \overset{\sigma}{\dashrightarrow} \chi'$, that is, that $\mathbb{S}$ of the type σ is applied to P of the type $\langle \Gamma, \chi \rangle$, can not actually happen when well formed rules are applied to well formed processes. This is for the same reason as discussed in Remark 8.2.1, that is, that this situation could occur only after application of some rewriting rule to a process of the shape like "$(b^{\mathsf{x}}).\cdots.(a^{\mathsf{x}}).P_0$" which is not well formed.

To further illustrate this issue let us consider the following two shape graphs which are both flow closed. Note that the condition discussed in the previous paragraph applies in the case of the right shape graph.



8.5 Flow Closure Correctness

By "flow closure correctness" we mean that the intended meaning of the flow edges is satisfied in flow closed shape graphs. This is expressed by the following proposition Proposition 8.5.1. We prove that the intended meaning meaning is valid only for processes P with $\mathrm{itags}(P) \cap \mathrm{dom}(\sigma) = \emptyset$. This condition is related to the similar condition from the definition of flow-closed graphs discussed above. This condition will be implied by well-formedness in all applications of Proposition 8.5.1. Another condition *(5)* below, which will be satisfied in all required applications as well, is further discussed in Remark 8.5.2.

PROPOSITION 8.5.1 (FLOW CLOSURE CORRECTNESS). *Let the following hold.*

(1) Γ *is a flow closed shape graph*

(2) $(\chi \overset{\sigma}{\dashrightarrow} \chi') \in \Gamma$

(3) $\vdash \mathbb{S} : \sigma$

(4) $\mathrm{itags}(P) \cap \mathrm{dom}(\sigma) = \emptyset$

(5) for all $x \in \mathrm{fn}(P)$ *and* $y \in \mathrm{dom}(\mathbb{S})$, $\overline{x} = \overline{y}$ *implies* $x = y$

Then $\vdash P : \langle \Gamma, \chi \rangle$ *implies* $\vdash \bar{\mathbb{S}}P : \langle \Gamma, \chi' \rangle$.

PROOF. *By induction on the structure of* P. *The only non-trivial case is when* $P = F.P_0$. *Then let* $\vdash P : \langle \Gamma, \chi \rangle$. *Thus there are some* φ *and* χ_0 *such that* $\vdash F : \varphi$, *and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$, *and* $\vdash P_0 : \langle \Gamma, \chi_0 \rangle$. *We distinguish the following two cases which correspond to the rules* F1 *and* F2 *from Definition 7.4.1. When*

$\varphi = \iota$ & $\mathfrak{o}(\iota) = \{\sigma_1, \ldots, \sigma_k\}$✶**:** *Thus* $\vdash F : \iota$ *and there has to be some* x *such that* $F = x$ *and* $\overline{x} = \iota$. *Obviously* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ *and because* Γ *is flow-closed we know that condition* F1 *is satisfied for* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *and* $(\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$. *It implies that there is* $(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$. *Clearly condition (5) is satisfied for* P_0 *because* $\mathrm{fn}(P) = \mathrm{fn}(P_0) \cup \{x\}$. *Thus we can use the induction hypothesis for* P_0 *and* $(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$ *by which we obtain that* $\vdash \bar{\mathbb{S}}P_0 : \langle \Gamma, \chi' \rangle$.

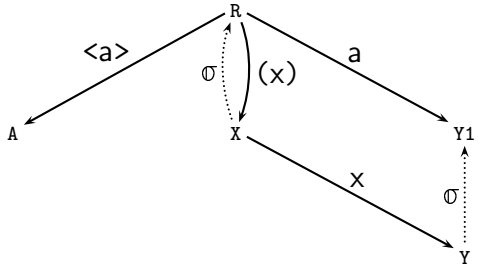
Now we also know that $\vdash \mathbb{S} : \mathfrak{o}$ *and thus it has to hold that* $x \in \mathrm{dom}(\mathbb{S})$ *and* $\vdash \mathbb{S}(x) : \{\sigma_1, \ldots, \sigma_k\}$✶. *The later implies that* $\mathbb{S}(x)$ *is not a single name. Thus* $\bar{\mathbb{S}}(x.P_0) = \mathbb{S}(x)_*\bar{\mathbb{S}}P_0$. *Now it is easy to see that there is some* l *and some forms* $s_1, \ldots, s_l$ *such that* $\bar{\mathbb{S}}(x.P_0) = s_1.\cdots.s_l.\bar{\mathbb{S}}P_0$. *Moreover for every* s_j *there is some* $i \in \{1, \ldots, k\}$ *such that* $\vdash s_j : \sigma_i$. *We have already showed that condition* F1 *is satisfied and thus there is the edge* $(\chi' \xrightarrow{\sigma_i} \chi') \in \Gamma$ *for every* $i \in \{1, \ldots, k\}$. *Previously we have obtained* $\vdash \bar{\mathbb{S}}P_0 : \langle \Gamma, \chi' \rangle$ *by the induction hypothesis. Hence the claim* $\vdash \bar{\mathbb{S}}F.P_0 : \langle \Gamma, \chi' \rangle$ *is proved by* l *applications of rule* TFRM.

otherwise: *It holds that* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ *because* $\mathrm{itags}(P) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ *and thus condition* F2 *is satisfied for* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *and* $(\chi \overset{\mathbb{S}}{\dashrightarrow} \chi') \in \Gamma$ *because* Γ *is flow-closed. Thus there are some* χ_0' *and edges* $(\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0') \in \Gamma$ *and* $(\chi_0 \overset{\mathbb{S}}{\dashrightarrow} \chi_0') \in \Gamma$. *Obviously* $\mathrm{itags}(P_0) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ *and thus the assumptions of the induction step for* $(\chi_0 \overset{\mathbb{S}}{\dashrightarrow} \chi_0') \in \Gamma$ *and* $\vdash P_0 : \langle \Gamma, \chi_0 \rangle$ *are satisfied. To use the induction hypothesis we need to verify assumption (5) for* P_0. *Let* $x_0 \in \mathrm{fn}(P_0)$. *When* $x_0 \in \mathrm{fn}(P)$ *then the condition is already satisfied. When* $x_0 \in \mathrm{fn}(P_0) \backslash \mathrm{fn}(P)$ *then* $x_0 \in \mathrm{bn}(F)$ *and thus* $\overline{x_0} \in \mathrm{itags}(P)$. *It means that* $\overline{x_0} \notin \mathrm{dom}(\mathfrak{o})$ *by (4). Now for any* $y_0 \in \mathrm{dom}(\mathbb{S})$ *we have* $\overline{y_0} \in \mathrm{dom}(\mathfrak{o})$ *and thus* $\overline{x_0} \neq \overline{y_0}$. *Thus we can use the induction hypothesis for* P_0 *and* $(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$ *by which we obtain that* $\vdash \bar{\mathbb{S}}P_0 : \langle \Gamma, \chi' \rangle$.

From $\vdash F : \varphi$ *we obtain* $\vdash \bar{\mathbb{S}}F : \bar{\mathfrak{o}}\varphi$ *by Proposition 8.2.2 case (5). Thus* $\vdash \bar{\mathbb{S}}F.\bar{\mathbb{S}}P_0 : \langle \Gamma, \chi' \rangle$ *because* $(\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0') \in \Gamma$. *By the definition we have that* $\bar{\mathbb{S}}P = \bar{\mathbb{S}}F.\bar{\mathbb{S}}P_0$ *if* $F \notin \mathrm{dom}(\mathbb{S})$. *Now when* $F \in \mathrm{dom}(\mathbb{S})$, *that is* $F = x$ *for some* x, *we have that* $\iota \in \mathrm{dom}(\mathfrak{o})$ *for* $\iota = \overline{x}$. *We see that* $\varphi = \iota$ *and thus* $\mathfrak{o}(\iota) \in$ TypeTag *because starred message types are covered by the previous case. But it means that* $\mathbb{S}(x) \in$ Name. *Thus* $\mathbb{S}(x) = \bar{\mathbb{S}}x = \bar{\mathbb{S}}F$ *and thus* $\bar{\mathbb{S}}P = \mathbb{S}(x)_*\bar{\mathbb{S}}P_0 = \bar{\mathbb{S}}F.\bar{\mathbb{S}}P_0$ *as well. Hence the claim is proved.* ∎

REMARK 8.5.2. The condition (*5*) from Proposition 8.5.1 is closely related the similar condition from Proposition 8.2.2 previously discussed in Remark 6.3.3 and Remark 8.2.1. This condition will be implied by well-formedness in all possible applications of Proposition 8.5.1. To illustrate this let us consider the following shape graph where $\mathfrak{o} = \{\mathsf{x} \mapsto \mathsf{a}\}$ as in Remark 8.2.1.

$\Gamma =$



This shape graph is flow-closed. Let us take $\mathbb{S} = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{a}^{\mathsf{a}}\}$ so that we have $\vdash \mathbb{S} : \mathfrak{o}$. Now let us consider the process $P = \mathsf{y}^{\mathsf{x}}.\mathsf{0}$ which does not satisfy (*5*) with $\mathbb{S}$. We see that $\vdash P : \langle \Gamma, \mathtt{X} \rangle$ but $\not\vdash \bar{\mathbb{S}}P : \langle \Gamma, \mathtt{R} \rangle$ even though there is a flow edge $(\mathtt{X} \overset{\mathfrak{o}}{\dashrightarrow} \mathtt{R}) \in \Gamma$ and $\vdash \mathbb{S} : \mathfrak{o}$. It means that the intended meaning of $(\mathtt{X} \overset{\mathfrak{o}}{\dashrightarrow} \mathtt{R}) \in \Gamma$ is not satisfied for P and $\bar{\mathbb{S}}P$. This is, however, not a problem because we have already discussed in Remark 8.2.1 that application of well formed rules to well formed processes can never lead to application of $\mathbb{S}$ to P in this particular example.

We have mentioned above that $(\mathtt{X} \overset{\mathfrak{o}}{\dashrightarrow} \mathtt{R}) \in \Gamma$ describes a possibility that a substitution $\mathbb{S}'$ of the type $\mathfrak{o}$ is applied to some process P of the type $\langle \Gamma, \mathtt{X} \rangle$. As an example how this can happen let us consider the following Mobile Ambients process.

$$Q = \mathsf{<a>.0} \mid \mathsf{(y^x).y^x.0} = \mathsf{<a>.0} \mid \mathsf{(y^x)}.P$$

It is easy to check that $\vdash Q : \langle \Gamma, \mathtt{R} \rangle$ and $\vdash P : \langle \Gamma, \mathtt{X} \rangle$. Application of the monadic Mobile Ambients rewriting rules $\mathcal{A}_{\mathsf{mon}}$ from Section 5.3 gives us the following rewriting.

$$\mathsf{<a>.0} \mid \mathsf{(y^x).y^x.0} \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \mathsf{a.0} \qquad = \bar{\mathbb{S}}'P \text{ where } \mathbb{S}' = \{\mathsf{y}^{\mathsf{x}} \mapsto \mathsf{a}\}$$

Here a substitution $\mathbb{S}'$ of type $\mathfrak{o}$ was applied to process P of type $\langle \Gamma, \mathtt{X} \rangle$. We can see that the names from the domain of $\mathbb{S}'$ are always constructed from some input-binder above P. Thus the well-formedness rules W1 and W2 ensure that (*5*) is always satisfied. The discussion of this issue will be concluded in Remark 8.7.1. ∎

8.6 Properties of Type Instantiations

Here we prove some properties of type instantiations. Firstly we prove weakening and strengthening lemmas which allow us to add or remove variables from the domain of $\mathbb{T}$ while preserving the $\models_s$ relations.

LEMMA 8.6.1 (TYPE INSTANTIATION WEAKENING). *Let type instantiations* $\mathbb{\pi}$ *and* $\mathbb{\pi}_0$ *such that* $\mathbb{\pi}_0 \subseteq \mathbb{\pi}$ *be given. Then* $\mathbb{\pi}_0 \models_s \mathring{P} : \Pi$ *implies* $\mathbb{\pi} \models_s \mathring{P} : \Pi$ *where* $s \in \{\mathsf{L}, \mathsf{R}\}$.

PROOF. *By induction of the structure of* $\mathring{P}$ *prove that for any* Π', $\mathbb{\pi}_0 \models_s \mathring{P} : \Pi'$ *implies* $\mathbb{\pi} \models_s \mathring{P} : \Pi'$. ∎

LEMMA 8.6.2 (TYPE INSTANTIATION STRENGTHENING). *Let type instantiations* $\mathbb{\pi}$ *and* $\mathbb{\pi}_0$ *such that* $\mathbb{\pi}_0 \subseteq \mathbb{\pi}$ *and* $\mathrm{dom}(\mathbb{\pi}_0) \subseteq \mathrm{var}(\mathring{P})$ *be given. Then* $\mathbb{\pi} \models_s \mathring{P} : \Pi$ *implies* $\mathbb{\pi}_0 \models_s \mathring{P} : \Pi$ *where* $s \in \{\mathsf{L}, \mathsf{R}\}$.

PROOF. *By induction of the structure of* $\mathring{P}$ *prove that for any* Π', $\mathbb{\pi}_0 \models_s \mathring{P} : \Pi'$ *implies* $\mathbb{\pi} \models_s \mathring{P} : \Pi'$. ∎

The following lemma is similar to the type instantiation weakening Lemma 8.6.1 above but it extends the shape graph rather then the type instantiation.

LEMMA 8.6.3. $\mathbb{\pi} \models_s \mathring{P} : \langle \Gamma, \chi \rangle$ *implies* $\mathbb{\pi} \models_s \mathring{P} : \langle \Gamma \cup \Gamma', \chi \rangle$ *where* $s \in \{\mathsf{L}, \mathsf{R}\}$.

PROOF. *By induction of the structure of* $\mathring{P}$ *prove that for any* χ', $\mathbb{\pi} \models_s \mathring{P} : \langle \Gamma, \chi' \rangle$ *implies* $\mathbb{\pi} \models_s \mathring{P} : \langle \Gamma \cup \Gamma', \chi' \rangle$. ∎

The following is a simple implication of the strengthening lemma. It says that it is enough to check the local closure condition Definition 7.6.5 on type instantiations which mention only variables from the left-hand side of some rule. Thus this lemma implies that it is enough to check the local closure only for finitely many type instantiations instead of for all type instantiations (which are infinite in number).

LEMMA 8.6.4. *Let* $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *and let* $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\langle \Gamma, \chi_{\mathsf{r}} \rangle)$. *Let* $\mathbb{\pi}_0 \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *imply* $\mathbb{\pi}_0 \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$ *for all* $\mathbb{\pi}_0$ *such that* $\mathrm{dom}(\mathbb{\pi}_0) = \mathrm{var}(\mathring{P})$. *Then* $\mathbb{\pi} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *imply* $\mathbb{\pi} \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$ *for all* $\mathbb{\pi}$.

PROOF. *Let* $\mathbb{\pi}$ *be such a type instantiation and let* $\mathbb{\pi} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. *Take* $\mathbb{\pi}_0 = \mathrm{var}(\mathring{P}) \lhd \mathbb{\pi}$. *It holds that* $\mathrm{dom}(\mathbb{\pi}_0) = \mathrm{var}(\mathring{P})$ *(because* $\mathbb{\pi} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *is defined). We have* $\mathbb{\pi}_0 \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *by the type instantiation strengthening Lemma 8.6.2. By the assumption we have* $\mathbb{\pi}_0 \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$. *Thus the claim holds by the type instantiation weakening Lemma 8.6.1.* ∎

The following says that when $\mathbb{P}$ has the type $\mathbb{\pi}$ then $\mathbb{P}$ instantiates $\mathring{F}$ to the process $\mathbb{P}[\![\mathring{F}]\!]$ of the type $\mathbb{P}[\![\mathring{F}]\!]$. In this lemma the value of Γ is irrelevant because $\mathring{F}$ contains no process variables.

LEMMA 8.6.5. *Let* $\Gamma \vdash \mathbb{P} : \mathbb{\pi}$ *and* $\mathrm{var}(\mathring{F}) \subseteq \mathrm{dom}(\mathbb{P})$. *Then* $\vdash \mathbb{P}[\![\mathring{F}]\!] : \mathbb{\pi}(\!|\mathring{F}|\!)$.

PROOF. *Follows directly from Definition 7.6.2* ∎

The following is an extension of the previous lemma to process templates. When $\mathring{P}$ can be instantiated by $\mathbb{T}$ to Π then $\mathbb{P}$ of the type $\mathbb{T}$ instantiates $\mathring{P}$ to a process matching Π.

LEMMA 8.6.6. *When* $\mathbb{T} \models_{\text{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *and* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *then* $\vdash \mathbb{P}[\![\mathring{P}]\!] : \langle \Gamma, \chi \rangle$.

PROOF. *By induction on the structure of* $\mathring{P}$ *using Lemma 8.6.5 for* $\mathring{P} = \mathring{F}.\mathring{P}_0$. ∎

The following lemma says that when we have the process instantiation $\mathbb{P}$ which instantiates $\mathring{E}$ to a element of type ε then we can construct a type instantiation $\mathbb{T}$ which is a type of $\mathbb{P}$ and which instantiates $\mathring{E}$ to ε directly. The value of Γ is again irrelevant in this case.

LEMMA 8.6.7. *Let* $\mathring{E}$ *be a well lhs-formed element template. When* $\vdash \mathbb{P}[\![\mathring{E}]\!] : \varepsilon$ *and* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{E})$ *then there is a type instantiation* $\mathbb{T}$ *such that for any* Γ *it holds that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *and* $\mathbb{T}(\!|\mathring{E}|\!) = \varepsilon$.

PROOF. *Let us distinguish the following cases by the structure of* $\mathring{E}$. *Let*

$\mathring{E} = x$**:** *Take* $\mathbb{T} = \mathbb{P} = \emptyset$ *to prove the claim.*

$\mathring{E} = \mathring{x}$**:** *Let* $x = \mathbb{P}(\mathring{x})$. *Clearly* $\mathbb{P}[\![\mathring{E}]\!] = x$ *and* $\varepsilon = \overline{x}$. *Take* $\mathbb{T} = \{\mathring{x} \mapsto \overline{x}\}$. *Hence the claim.*

$\mathring{E} = (\mathring{x}_1, \ldots, \mathring{x}_k)$**:** *Let* $x_i = \mathbb{P}(\mathring{x}_i)$ *for* $i \in \{1, \ldots, k\}$. *Clearly* $\mathbb{P}[\![\mathring{E}]\!] = (x_1, \ldots, x_k)$ *and* $\varepsilon = (\overline{x_1}, \ldots, \overline{x_k})$. *Take* $\mathbb{T} = \{\mathring{x}_1 \mapsto \overline{x_1}, \ldots, \mathring{x}_k \mapsto \overline{x_k}\}$. *Hence the claim.*

$\mathring{E} = \mathord{<}\mathring{m}_1, \ldots, \mathring{m}_k\mathord{>}$**:** *Let* $M_i = \mathbb{P}(\mathring{m}_i)$ *for* $i \in \{1, \ldots, k\}$. *We see that there are* μ_1, …, μ_k *such that* $\varepsilon = \mathord{<}\mu_1, \ldots, \mu_k\mathord{>}$ *and* $\vdash M_i : \mu_i$ *for all* $i \in \{1, \ldots, k\}$. *Now let us take* $\mathbb{T} = \{\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k\}$. *Hence the claim.* ∎

The following is the version of the previous lemma which works with form templates instead of element templates. Once again the value of Γ is irrelevant.

LEMMA 8.6.8. *Let* $\mathring{F}$ *be a well lhs-formed element template. When* $\vdash \mathbb{P}[\![\mathring{F}]\!] : \varphi$ *and* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{F})$ *then there is a type instantiation* $\mathbb{T}$ *such that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *(for any* Γ*) and* $\mathbb{T}(\!|\mathring{F}|\!) = \varphi$.

PROOF. *Let* $\mathring{F}$ *be a well lhs-formed element template. Let* $\vdash \mathbb{P}[\![\mathring{F}]\!] : \varphi$ *and* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{F})$. *We see that there is* k *such that* $\mathring{F} = \mathring{E}_0 \ldots \mathring{E}_k$ *and* $\varphi = \varepsilon_0 \ldots \varepsilon_k$. *Let* $\mathbb{P}_i = \mathrm{var}(\mathring{E}_i) \lhd \mathbb{T}$ *for all* $i \in \{0, \ldots, k\}$. *Clearly* $\mathbb{T}_i[\![\mathring{E}_i]\!] = \varepsilon_i$. *Let* Γ *be arbitrary. Using Lemma 8.6.7 we obtain that for every* i *there is* $\mathbb{T}_i$ *such that* $\Gamma \vdash \mathbb{P}_i : \mathbb{T}_i$ *and* $\mathbb{T}_i(\!|\mathring{E}_i|\!) = \varepsilon_i$. *Let us take* $\mathbb{T} = \mathbb{T}_0 \cup \cdots \cup \mathbb{T}_k$.

First of all we need $\mathbb{T}$ *proved to be a function. Let* $\mathring{z} \in \mathrm{dom}(\mathbb{T}_i)$ *and* $\mathring{z} \in \mathrm{dom}(\mathbb{T}_j)$ *for some* $i \neq j$. *We need to prove that* $\mathbb{T}_i(\mathring{z}) = \mathbb{T}_j(\mathring{z})$. *We see that* $\mathrm{dom}(\mathbb{T}_i) = \mathrm{var}(\mathring{E}_i)$ *and that each* ε_i *is a well lhs-formed element template because* φ *is a well lhs-formed*

form template. Hence $\mathring{z}$ *has to be name variables, that is,* $\mathring{z} = \mathring{x}$ *for some* $\mathring{x}$*. From* $\Gamma \vdash \mathbb{P}_i : \mathbb{T}_i$ *and* $\Gamma \vdash \mathbb{P}_i : \mathbb{T}_i$ *we obtain that* $\vdash \mathbb{P}_i(\mathring{x}) : \mathbb{T}_i(\mathring{x})$ *and* $\vdash \mathbb{P}_j(\mathring{x}) : \mathbb{T}_j(\mathring{x})$*. It is easy to see that* $\mathbb{P}_i(\mathring{x}) = \mathbb{P}_j(\mathring{x})$ *which thus implies that* $\mathbb{T}$ *is a function. Now* $\Gamma \vdash \mathbb{P}_i : \mathbb{T}_i$ *is clear. Hence the claim.* ∎

Finally the following is the extension of previous two lemmas to process templates. When $\mathbb{P}$ instantiates $\mathring{P}$ to a process of the type Π then we can find a type instantiation $\mathbb{T}$ which is a type of $\mathbb{P}$ and which instantiates $\mathring{P}$ to Π directly. This lemma is used in the proof of subject reduction.

LEMMA 8.6.9. *Let the shape predicate* $\Pi = \langle \Gamma, \chi \rangle$*, a well formed lhs-template* $\mathring{P}$*, and the process instantiation* $\mathbb{P}$ *with* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{P})$ *be given. When* $\vdash \mathbb{P}[\![\mathring{P}]\!] : \Pi$ *then there exists some type instantiation* $\mathbb{T}$ *such that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *and* $\mathbb{T} \models_{\mathsf{L}} \mathring{P} : \Pi$*.*

PROOF. *Let the assumptions be satisfied. Let* $\vdash \mathbb{P}[\![\mathring{P}]\!] : \Pi$*. Prove the claim by induction of the structure of* $\mathring{P}$*. Let*

$\mathring{P} = \mathsf{0}$**:** *Take* $\mathbb{T} = \emptyset$*. Obviously* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{P}) = \emptyset$ *and thus* $\Gamma \vdash \mathbb{P} : \mathbb{T}$*. Moreover* $\mathbb{T} \models_{\mathsf{L}} \mathsf{0} : \Pi$ *holds as well.*

$\mathring{P} = \mathring{p}$**:** *Take* $\mathbb{T} = \{\mathring{p} \mapsto \chi\}$ *where* χ *is the root of* Π*. Obviously* $\mathrm{dom}(\mathbb{P}) = \mathrm{var}(\mathring{P}) = \{\mathring{p}\}$*. We see that* $\langle \Gamma, \mathbb{T}(\mathring{p}) \rangle = \Pi$ *and thus* $\Gamma \vdash \mathbb{P} : \mathbb{T}$*. Moreover* $\mathbb{T} \models_{\mathsf{L}} \mathring{p} : \Pi$ *holds as well.*

$\mathring{P} = \mathring{F}.\mathring{P}_0$**:** *We know that* $\vdash \mathbb{P}[\![\mathring{F}]\!].\mathbb{P}[\![\mathring{P}_0]\!] : \langle \Gamma, \chi \rangle$*. Thus there are some* F *and* χ_0 *such that* $\vdash \mathbb{P}[\![\mathring{F}]\!] : \varphi$*, and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$*, and* $\vdash \mathbb{P}[\![\mathring{P}_0]\!] : \langle \Gamma, \chi_0 \rangle$*. Let us take* $\mathbb{P}' = \mathrm{var}(\mathring{F}) \lhd \mathbb{P}$ *and* $\mathbb{P}_0 = \mathrm{var}(\mathring{P}_0) \lhd \mathbb{P}$*. By 8.6.8 we obtain* $\mathbb{T}'$ *such that with* $\Gamma \vdash \mathbb{P}' : \mathbb{T}'$ *and* $\mathbb{T}'(\!|\mathring{F}|\!) = \varphi$*. Moreover by the induction hypothesis for* $\langle \Gamma, \chi_0 \rangle$*,* $\mathring{P}_0$*, and* $\mathbb{P}_0$ *we obtain* $\mathbb{T}_0$ *such that* $\Gamma \vdash \mathbb{P}_0 : \mathbb{T}_0$ *and* $\mathbb{T}_0 \models_{\mathsf{L}} \mathring{P}_0 : \langle \Gamma, \chi_0 \rangle$*.*

Let us take $\mathbb{T} = \mathbb{T}' \cup \mathbb{T}_0$ *and prove that* $\mathbb{T}$ *is a type instantiation, that is, that values of* $\mathbb{T}'$ *and* $\mathbb{T}_0$ *for variables in both* $\mathrm{dom}(\mathbb{T}')$ *and* $\mathrm{dom}(\mathbb{T}_0)$ *do not differ. We know that* $\mathring{P}$ *is a well formed lhs-template and thus by* L3 *we obtain that any variable which is in both* $\mathrm{dom}(\mathbb{T}')$ *and* $\mathrm{dom}(\mathbb{T}_0)$ *has to be a name variable. Thus, let us take* $\mathring{x} \in \mathrm{dom}(\mathbb{T}') \cap \mathrm{dom}(\mathbb{T}_0)$*. By the definition of* $\mathbb{P}'$ *and* $\mathbb{P}_0$ *we know that* $\mathbb{P}'(\mathring{x}) = \mathbb{P}(\mathring{x}) = \mathbb{P}_0(\mathring{x})$*. From* $\Gamma \vdash \mathbb{P}' : \mathbb{T}'$ *and* $\Gamma \vdash \mathbb{P}_0 : \mathbb{T}_0$ *we also know that* $\vdash \mathbb{P}'[\![\mathring{x}]\!] : \mathbb{T}'(\!|\mathring{x}|\!)$ *and* $\vdash \mathbb{P}_0[\![\mathring{x}]\!] : \mathbb{T}_0(\!|\mathring{x}|\!)$ *hold. But this means that* $\mathbb{T}'(\mathring{x}) = \overline{\mathbb{P}'(\mathring{x})} = \overline{\mathbb{P}_0(\mathring{x})} = \mathbb{T}_0(\mathring{x})$*. Thus* $\mathbb{T}$ *is a function and we directly obtain that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$*.*

By Lemma 8.6.1 and $\mathbb{T}_0 \models_{\mathsf{L}} \mathring{P}_0 : \langle \Gamma, \chi_0 \rangle$ *we obtain that* $\mathbb{T} \models_{\mathsf{L}} \mathring{P}_0 : \langle \Gamma, \chi_0 \rangle$*. Now* $\mathbb{T}(\!|\mathring{F}|\!) = \varphi$ *and thus* $(\chi \xrightarrow{\mathbb{T}(\!|\mathring{F}|\!)} \chi_0) \in \Gamma$*. Hence the claim* $\mathbb{T} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *holds.*

$\mathring{P} = \mathring{P}_0 \mid \mathring{P}_1$**:** *We have that* $\mathbb{P}[\![\mathring{P}]\!] = \mathbb{P}[\![\mathring{P}_0]\!] \mid \mathbb{P}[\![\mathring{P}_1]\!]$ *and thus* $\vdash \mathbb{P}[\![\mathring{P}_0]\!]{:}\Pi$ *and* $\vdash \mathbb{P}[\![\mathring{P}_1]\!]{:}\Pi$*. Let us take* $\mathbb{P}_0 = \mathrm{var}(\mathring{P}_0) \lhd \mathbb{P}$ *and* $\mathbb{P}_1 = \mathrm{var}(\mathring{P}_1) \lhd \mathbb{P}$*. Thus the assumptions of the induction step for* $\mathring{P}_0$ *are satisfied. By the induction hypothesis we obtain*

$\mathbb{m}_0$ *and* $\mathbb{m}_1$ *such that* $\Gamma \vdash \mathbb{P}_0 : \mathbb{m}_0$ *and* $\mathbb{m}_0 \models_{\text{L}} \mathring{P}_0 : \Pi$ *as well as* $\Gamma \vdash \mathbb{P}_1 : \mathbb{m}_1$ *and* $\mathbb{m}_1 \models_{\text{L}} \mathring{P}_1 : \Pi$.

Let us take $\mathbb{m} = \mathbb{m}_0 \cup \mathbb{m}_1$. *Now* $\mathbb{m}$ *is a function from the same reasons as* $\mathbb{m}$ *from the previous case for* $\mathring{P} = \mathring{F}.\mathring{P}_0$. *Thus we directly obtain that* $\Gamma \vdash \mathbb{P} : \mathbb{m}$. *Now we obtain* $\mathbb{m} \models_{\text{L}} \mathring{P}_0 : \Pi$ *and* $\mathbb{m} \models_{\text{L}} \mathring{P}_1 : \Pi$ *from above by Lemma 8.6.1. Hence the claim.*

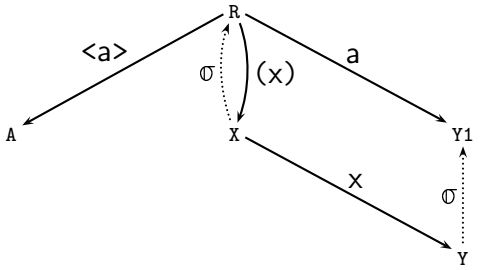
otherwise: *Condition* L6 *ensures that the above cases cover all possibilities.*

8.7 Subject Reduction

Here we discuss two issues related to subject reduction and we prove it. The first issue is the purpose of well-formedness conditions W1 and W2. The second issue is a problem with subject reduction in the previously published version of POLY✶ [MW04a]. Finally we provide the proof of subject reduction for the POLY✶ system first time presented in this thesis.

REMARK 8.7.1. Now it is easy to conclude the discussion from Remark 8.5.2 about the purpose of well-formedness conditions W1 & W2 and about their relationship to subject reduction. Let us again consider the following graph $\Pi_0 = \langle \Gamma_0, \mathtt{R} \rangle$ from Remark 8.5.2 together with the monadic Mobile Ambients rewriting rules $\mathcal{A}_{\mathsf{mon}}$ from Section 5.3. Let $\mathfrak{o} = \{\mathsf{x} \mapsto \mathsf{a}\}$ as before.

$\Gamma_0 =$



We see that Π_0 is an $\mathcal{A}_{\mathsf{mon}}$-type. Now let us consider the process

$$P_0 = \texttt{<a>.0} \mid (\mathsf{x}^{\mathsf{x}}).\mathsf{y}^{\mathsf{x}}.0$$

which violates W1. Let us for a brief moment ignore the fact that P_0 is not well formed in order to show what goes wrong without W1. All the META✶ definitions work correctly without W1 and thus we can prove the following expected rewriting.

$$\texttt{<a>.0} \mid (\mathsf{x}^{\mathsf{x}}).\mathsf{y}^{\mathsf{x}}.0 \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \mathsf{y}^{\mathsf{x}}.0$$

But now we see that $\not\vdash \mathsf{y}^{\mathsf{x}}.0 : \Pi_0$ which means that Π_0 is not $\mathcal{A}_{\mathsf{mon}}$-closed. Hence subject reduction does not hold without W1. The problem is that the type substitution

$\varpi = \{\mathsf{x} \mapsto \mathsf{a}\}$ changed x to a in the graph but the corresponding process substitution $\mathbb{S} = \{\mathsf{x}^{\mathsf{x}} \mapsto \mathsf{a}^{\mathsf{a}}\}$ left y^{x} in process P_0 unchanged. Well-formedness condition W1 ensures that this does not happen.

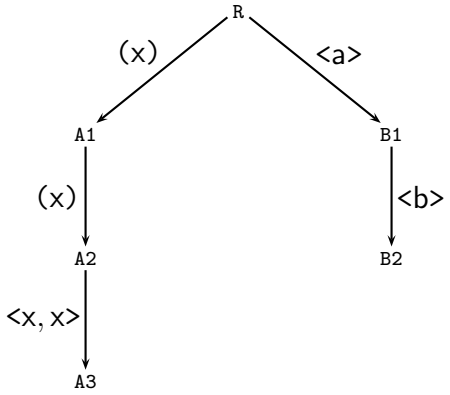
The above example can be adapted to demonstrate that W2 is required from the very same reason. However, W2 (which prevents nesting of input-binders binding the same type tag) has also another purpose which we reveal now. Let us consider the process

$$P_1 = (\mathsf{x}^{\mathsf{x}}).(\mathsf{y}^{\mathsf{x}}).\langle \mathsf{x}^{\mathsf{x}}, \mathsf{y}^{\mathsf{x}} \rangle.0 \mid \langle \mathsf{a} \rangle.\langle \mathsf{b} \rangle.0$$

which violates W2. However, META✶ again works correctly even for processes violating W2 and we obtain the following expected rewriting.

$$P_1 \xrightarrow{\mathcal{A}_{\mathsf{mon}}} (\mathsf{y}^{\mathsf{x}}).\langle \mathsf{a}, \mathsf{y}^{\mathsf{x}} \rangle.0 \mid \langle \mathsf{b} \rangle.0 \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \langle \mathsf{a}, \mathsf{b} \rangle.0$$

The question of a type of P_1 in POLY✶ is, however, more complicated. Let us consider the following shape predicate which directly corresponds to the syntax tree of P_1.

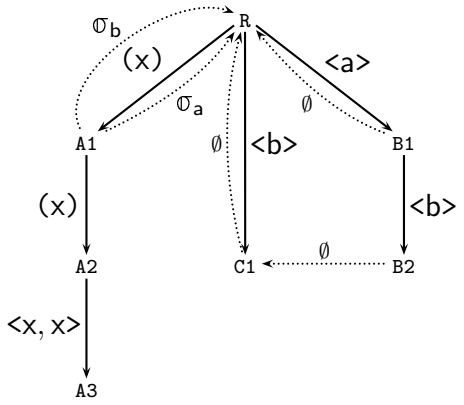


We see that the distinction between x^{x} and y^{x} inside $\langle \mathsf{x}^{\mathsf{x}}, \mathsf{y}^{\mathsf{x}} \rangle$ from P_1 is lost in the form type $\langle \mathsf{x}, \mathsf{x} \rangle$ from the shape graph. Hence any type substitution would yield the same result when applied to arguments of $\langle \mathsf{x}, \mathsf{x} \rangle$. It means that no application of a type substitution would introduce the form type $\langle \mathsf{a}, \mathsf{b} \rangle$ which has to be necessarily present in any shape predicate that matches P_1 and is $\mathcal{A}_{\mathsf{mon}}$-closed. Note that all the form types whose existence is required by closure conditions are constructed by application of type substitutions to form types already present in the graph. As a consequence of this we can deduce that the closure conditions can not require the presence of $\langle \mathsf{a}, \mathsf{b} \rangle$. Hence $\mathcal{A}_{\mathsf{mon}}$-types do not need to be $\mathcal{A}_{\mathsf{mon}}$-closed without W2.

To further demonstrate this issue let us consider the following $\mathcal{A}_{\mathsf{mon}}$-type $\Pi_1 =$

$\langle \Gamma_1, \mathtt{R} \rangle$ where $\varpi_\mathsf{a} = \{\mathsf{x} \mapsto \mathsf{a}\}$ and $\varpi_\mathsf{b} = \{\mathsf{x} \mapsto \mathsf{b}\}$.

$\Gamma_1 =$



We can see that $\vdash P_1 : \Pi_1$ but $\not\vdash$ $\mathsf{<a, b>.0} : \Pi_1$. In this case this is, however, mainly because of the condition from the Definition 7.4.1 which allows us not to propagate edges when $\mathrm{dom}(\varpi) \cap \mathrm{itags}(\varphi) \neq \emptyset$. This condition mainly decrease over-approximation in types. But even the propagation of the unpropagated edges $\{\mathtt{A1} \xrightarrow{\mathsf{(x)}} \mathtt{A2} \xrightarrow{\mathsf{<x,x>}} \mathtt{A3}\}$ would not give us a shape predicate that matches $\mathsf{<a, b>.0}$.

In order to redefine the flow closure to deal with nested input binders (that is, to have subject reduction without W2) it would be necessary for every type substitution ϖ to consider both possibilities that ϖ does or does not apply to any type tag ι from the graph. It would, however, dramatically increase the number edges in the graph and cause undesired over-approximation. ∎

Now we discuss issues with subject reduction in the previously published version of POLY✶.

REMARK 8.7.2. As already noted at several places the extension of POLY✶ which handles name restriction from the 2004 technical report [MW04a, Section 5.3] is broken and has no subject reduction. This is mainly because special names mentioned in rewriting rules were not prevented from being bound in a process as mentioned in the last paragraph of Section 4.3. Thus for example, the following rule description

$$\mathcal{R} = \{\mathbf{rewrite}\{\, \mathsf{a.0} \hookrightarrow \mathsf{b.0} \,\}\}$$

can be used to prove $\nu\mathsf{a.a.0} \overset{\mathcal{R}}{\hookrightarrow} \nu\mathsf{a.b.0}$. As briefly described later in Section 9.1, the 2004 extension defines *guarded shape predicates* to handle name restriction. A guarded shape predicate Π/X is a pair of a shape predicate Π and a set of names X. The set X can be seen as a set of names which are ν-bound in the shape graph of Π. When matching a process against a guarded shape predicate, a ν-bound name from the process can match any ν-bound name from the graph. To demonstrate let us take $\Pi = \langle \{\mathtt{R} \xrightarrow{\mathsf{c}} \mathtt{Y}\}, \mathtt{R} \rangle$ and let $\Pi/\{\mathsf{c}\}$ be a guarded shape predicate. Then in the 2004 POLY✶ extension it holds both that $\vdash \nu\mathsf{a.a.0} : \Pi/\{\mathsf{c}\}$ and $\vdash \nu\mathsf{c.c.0} : \Pi/\{\mathsf{c}\}$. Now the 2004 extension states that Π/X is a type when Π is a type. Hence Π/X is an

$\mathcal{R}$-type in the 2004 extension. However, we do not have $\vdash \nu\mathsf{a}.\mathsf{b}.0 : \Pi/\{\mathsf{c}\}$ which is necessary because of the rewriting proved above and hence we obtain contradiction with the subject reduction property.

A similar counterexample can be constructed for the instantiation $\mathcal{A}_{\mathsf{mon}}$ of POLY✶ to a type system for monadic Mobile Ambients. Clearly we can prove that

$$\mathsf{a}[\mathsf{in}\ \mathsf{b}.0] \mid \mathsf{b}[0] \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \mathsf{a}[\mathsf{b}[0]]$$

from which can obtain the following

$$\nu\mathsf{in}.\mathsf{a}[\mathsf{in}\ \mathsf{b}.0] \mid \mathsf{b}[0] \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \nu\mathsf{in}.\mathsf{a}[\mathsf{b}[0]]$$

and by α-conversion also accidentally

$$\nu\mathsf{out}.\mathsf{a}[\mathsf{out}\ \mathsf{b}.0] \mid \mathsf{b}[0] \xrightarrow{\mathcal{A}_{\mathsf{mon}}} \nu\mathsf{out}.\mathsf{a}[\mathsf{b}[0]].$$

Let us take $\Pi = \langle\{\mathtt{R} \xrightarrow{\mathtt{a[]}} \mathtt{A0}, \mathtt{A0} \xrightarrow{\mathtt{out\ b}} \mathtt{A1}, \mathtt{R} \xrightarrow{\mathtt{b[]}} \mathtt{B0}\}, \mathtt{R}\rangle$ and the guarded shape predicate $\Pi/\{\mathsf{out}\}$. Clearly Π is an $\mathcal{A}_{\mathsf{mon}}$-type so $\Pi/\{\mathsf{out}\}$ is an $\mathcal{A}_{\mathsf{mon}}$-type in the 2004 POLY✶ extension. However, we have

$$\vdash \nu\mathsf{out}.\mathsf{a}[\mathsf{out}\ \mathsf{b}.0] \mid \mathsf{b}[0] : \Pi/\{\mathsf{out}\} \quad \text{but} \quad \not\vdash \nu\mathsf{out}.\mathsf{a}[\mathsf{b}[0]] : \Pi/\{\mathsf{out}\}$$

which contradicts subject reduction because of the rewriting proved above. ∎

The following lemma is the first step towards the subject reduction. It says that structural equivalence preserves types. This lemma holds for all shape predicates not only for $\mathcal{R}$-types.

PROPOSITION 8.7.3. *Let $P \equiv Q$. Then $\vdash P : \Pi$ iff $\vdash Q : \Pi$.*

PROOF. *Proof by induction on the derivation of $P \equiv Q$.* ∎

The following is main part of the subject reduction proof for rule RRW. It is also closely related to Lemma 8.6.9 from the previous section. Basically the lemma says that $\Gamma \vdash \mathbb{P} : \mathbb{\pi}$ and $\mathbb{\pi} \models_{\mathsf{R}} \mathring{Q} : \Pi$ imply $\vdash \mathbb{P}[\![\mathring{Q}]\!] : \Pi$. However, this does not hold for all right-hand side templates $\mathring{Q}$ but only for those which are well formed w.r.t. some left-hand side template. Thus the left-hand side template must be involved in the formulation of the above property. Moreover we prove the lemma for all subtemplates $\mathring{Q}'$ in order to be able to do the induction step. The proof of the subject reduction will, however, use this lemma for $\mathring{Q}' = \mathring{Q}$.

LEMMA 8.7.4. *Let* **rewrite**$\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$*, let Γ be flow-closed, and let $\Gamma \vdash \mathbb{P} : \mathbb{\pi}$. Let $\mathbb{P}[\![\mathring{P}]\!]$ be defined and well formed. Then for a subtemplate $\mathring{Q}'$ of $\mathring{Q}$ it holds that $\mathbb{\pi} \models_{\mathsf{R}} \mathring{Q}' : \langle\Gamma, \chi\rangle$ implies $\vdash \mathbb{P}[\![\mathring{Q}']\!] : \langle\Gamma, \chi\rangle$.*

PROOF. *Let* $\mathcal{R}$, $\mathring{P}$, $\mathring{Q}$, Γ, $\mathbb{P}$, *and* $\mathbb{T}$ *be as in the assumptions. Let* $\mathring{Q}'$ *be a subtemplate of* $\mathring{Q}$ *and let* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}' : \langle \Gamma, \chi \rangle$. *Prove the claim* $\vdash \mathbb{P}[\![\mathring{Q}']\!] : \langle \Gamma, \chi \rangle$ *by induction on the structure of* $\mathring{Q}'$. *Let* $\Pi = \langle \Gamma, \chi \rangle$. *Let*

$\mathring{Q}' = \mathtt{0}$**:** *Clear.*

$\mathring{Q}' = \mathring{p}$**:** *From* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *we obtain* $\vdash \mathbb{P}[\![\mathring{p}]\!] : \langle \Gamma, \mathbb{T}(\mathring{p}) \rangle$. *We know that* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{p} : \Pi$ *and thus* $(\mathbb{T}(\mathring{p}) \overset{\emptyset}{\dashrightarrow} \chi) \in \Gamma$. *Thus by Proposition 8.5.1 for* $\mathbb{S} = \emptyset$ *we obtain the claim* $\vdash \mathbb{P}[\![\mathring{p}]\!] : \Pi$ *because* $\mathrm{dom}(\sigma) = \mathrm{dom}(\mathbb{S}) = \emptyset$.

$\mathring{Q}' = \{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$**:** *From* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}' : \langle \Gamma, \chi \rangle$ *we have that*

$$(\mathbb{T}(\mathring{p}) \overset{\{\ldots, \mathbb{T}(\mathring{x}_i) \mapsto \mathbb{T}(\mathring{s}_i), \ldots\}}{\dashrightarrow} \chi) \in \Gamma$$

Thus $\sigma = \{\mathbb{T}(\!|\mathring{x}_1|\!) \mapsto \mathbb{T}(\!|\mathring{s}_1|\!), \ldots, \mathbb{T}(\!|\mathring{x}_k|\!) \mapsto \mathbb{T}(\!|\mathring{s}_k|\!)\}$ *is a correctly defined type substitution, that is,* $\mathbb{T}(\!|\mathring{x}_i|\!) \neq \mathbb{T}(\!|\mathring{x}_j|\!)$ *whenever* $i \neq j$ *and all required values are defined. From this it follows that also* $\mathbb{S} = \{\mathbb{P}[\![\mathring{x}_1]\!] \mapsto \mathbb{P}[\![\mathring{s}_1]\!], \ldots, \mathbb{P}[\![\mathring{x}_k]\!] \mapsto \mathbb{P}[\![\mathring{s}_k]\!]\}$ *is a correctly defined substitution. It is easy to see that* $\vdash \mathbb{S} : \sigma$.

Now $\mathring{p} \in \mathrm{var}(\mathring{Q}')$ *and thus also* $\mathring{p} \in \mathrm{var}(\mathring{Q})$. *By* R2 *we have that* $\mathring{p} \in \mathrm{var}(\mathring{P})$. *Thus* $\mathring{p} \in \mathrm{dom}(\mathbb{P})$ *because we know that* $\mathbb{P}[\![\mathring{P}]\!]$ *is defined. Thus from* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *it follows that* $\vdash \mathbb{P}[\![\mathring{p}]\!] : \langle \Gamma, \mathbb{T}(\mathring{p}) \rangle$.

We also see that $\mathring{Q}' \vdash_{\exists} \mathring{x}_i \succcurlyeq \mathring{p}$ *for any* $0 \leqslant i \leqslant k$. *Thus also* $\mathring{Q} \vdash_{\exists} \mathring{x}_i \succcurlyeq \mathring{p}$ *and by* R2-2 *and* R5-6 *we obtain that* $\mathring{P} \vdash_{\exists} \mathring{x}_i \succcurlyeq \mathring{p}$ *holds as well. Thus by Lemma 6.3.2 we have that* $\overline{\mathbb{P}[\![\mathring{x}_i]\!]} \notin \mathrm{itags}(\mathbb{P}[\![\mathring{p}]\!])$. *Now because* $\overline{\mathbb{P}[\![\mathring{x}_i]\!]} = \mathbb{T}(\!|\mathring{x}_i|\!)$ *and because* i *was chosen arbitrarily we obtain that* $\mathrm{itags}(\mathbb{P}[\![\mathring{p}]\!]) \cap \mathrm{dom}(\sigma) = \emptyset$ *(because every type tag from* $\mathrm{dom}(\sigma)$ *is equal to* $\mathbb{T}(\!|\mathring{x}_i|\!)$ *for some* i*). This proves assumption (3) of Proposition 8.5.1. Now let us verify its assumption (4). Let* $x \in \mathrm{fn}(\mathbb{P}[\![\mathring{p}]\!])$ *and* $y \in \mathrm{dom}(\mathbb{S})$, *that is* $y = \mathbb{P}[\![\mathring{x}_i]\!]$ *for some* i. *But now* $\overline{x} = \overline{y}$ *implies* $x = y$ *by Lemma 6.3.4.*

Thus all the assumption of Proposition 8.5.1 (for $P = \mathbb{P}[\![\mathring{p}]\!]$*) are satisfied and we can use it to obtain* $\vdash \bar{\mathbb{S}}(\mathbb{P}[\![\mathring{p}]\!]) : \Pi$. *Now by the definition of application of a process instantiation we have that* $\bar{\mathbb{S}}(\mathbb{P}[\![\mathring{p}]\!]) = \mathbb{P}[\![\mathring{Q}']\!]$ *and hence the claim.*

$\mathring{Q}' = \mathring{F}.\mathring{P}_0$**:** *From* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}' : \langle \Gamma, \chi \rangle$ *we have that there exists some* χ_0 *such that* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma, \chi_0 \rangle$ *and* $(\chi \xrightarrow{\mathbb{T}(\mathring{F})} \chi_0) \in \Gamma$. *Obviously* $\mathring{Q}_0$ *is a subtemplate of* $\mathring{Q}$ *because* $\mathring{Q}'$ *is so. Now by the induction hypothesis (for* $\mathring{Q}_0$ *as* $\mathring{Q}'$*) we have that* $\vdash \mathbb{P}[\![\mathring{Q}_0]\!] : \langle \Gamma, \chi_0 \rangle$. *It is clear that* $\mathbb{T}(\!|\mathring{F}|\!)$ *is defined and thus* $\mathrm{var}(\mathring{F}) \subseteq \mathrm{dom}(\mathbb{T})$. *We know that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *and thus by Lemma 8.6.5 we obtain that* $\vdash \mathbb{P}[\![\mathring{F}]\!] : \mathbb{T}(\!|\mathring{F}|\!)$. *That is why* $\vdash \mathbb{P}[\![\mathring{F}]\!].\mathbb{P}[\![\mathring{Q}_0]\!] : \Pi$. *Hence the claim.*

$\mathring{Q}' = \mathring{Q}_0 \mid \mathring{Q}_1$**:** *From* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}' : \langle \Gamma, \chi \rangle$ *we have* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma, \chi \rangle$. *Obviously* $\mathring{Q}_0$ *is a subtemplate of* $\mathring{Q}$ *because* $\mathring{Q}'$ *is so. Now by the induction hypothesis (for* $\mathring{Q}_0$ *as* $\mathring{Q}'$*) we have that* $\vdash \mathbb{P}[\![\mathring{Q}_0]\!] : \Pi$. *Similarly for* $\mathring{Q}_1$ *we obtain* $\vdash \mathbb{P}[\![\mathring{Q}_1]\!] : \Pi$. *Hence the claim.* ∎

The following proves the subject reduction. We implicitly suppose that $\mathcal{R}$ is well formed.

THEOREM (PROOF OF THEOREM 7.6.11). *For every* Π *and* $\mathcal{R}$*, it holds that* $\mathcal{R} \Vdash_{\mathsf{type}} \Pi$ *implies* $\mathcal{R} \Vdash_{\mathsf{closed}} \Pi$.

PROOF. *Let* Π *be an* $\mathcal{R}$*-type of* P *and let* $P \xhookrightarrow{\mathcal{R}} Q$*. We need to prove that* $\vdash Q : \Pi$*. Prove this claim by induction on the derivation of* $P \xhookrightarrow{\mathcal{R}} Q$*. Recall that we implicitly suppose* $\mathcal{R}$ *to be well formed. Let* $\Pi = \langle \Gamma, \chi \rangle$*. Let* $P \xhookrightarrow{\mathcal{R}} Q$ *be derived by*

RRW**:** *Then we know that there are some well formed lhs- and rhs-templates* $\mathring{P}$ *and* $\mathring{Q}$ *with* $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$*, and that there is a process instantiation* $\mathbb{P}$ *such that* $P = \mathbb{P}[\![\mathring{P}]\!]$ *and* $Q = \mathbb{P}[\![\mathring{Q}]\!]$*. We can suppose* $\mathrm{var}(\mathring{P}) = \mathrm{dom}(\mathbb{P})$ *also because* R2 *and* R3*. We know that* $\vdash \mathbb{P}[\![\mathring{P}]\!] : \Pi$ *and thus by Lemma 8.6.9 we obtain that there is some type instantiation* $\mathbb{T}$ *such that* $\Gamma \vdash \mathbb{P} : \mathbb{T}$ *and* $\mathbb{T} \vDash_{\mathsf{L}} \mathring{P} : \Pi$*. Now* Γ *is an* $\mathcal{R}$*-type and thus* Γ *is locally closed at its root node* χ*. Thus also* $\mathbb{T} \vDash_{\mathsf{R}} \mathring{Q} : \Pi$*. Obviously* $\mathring{Q}$ *is a subtemplate of itself and thus by Lemma 8.7.4 we have that* $\vdash \mathbb{P}[\![\mathring{Q}]\!] : \Pi$ *holds. Hence the claim.*

RACT**:** *Then there are a process variable* $\mathring{p}$ *and a well formed lhs-template* $\mathring{P}$ *such that* $\mathbf{active}\{\, \mathring{p} \ \mathbf{in}\ \mathring{P} \,\} \in \mathcal{R}$*. Moreover there are a process instantiation* $\mathbb{P}$ *and processes* P_0 *and* Q_0 *such that* $P = (\mathbb{P}[\mathring{p} \mapsto P_0])[\![\mathring{P}]\!]$ *and* $Q = (\mathbb{P}[\mathring{p} \mapsto Q_0])[\![\mathring{P}]\!]$ *and also* $P_0 \xhookrightarrow{\mathcal{R}} Q_0$*. When* $\mathring{p} \notin \mathrm{var}(\mathring{P})$ *then* $P = Q$ *and the claim holds. Otherwise let* $\mathbb{P}_P = \mathbb{P}[\mathring{p} \mapsto P_0]$*. We can suppose that* $\mathrm{var}(\mathring{P}) = \mathrm{dom}(\mathbb{P}_P)$ *because* P *is defined and the values of* $\mathbb{P}_P$ *for variables not in* $\mathring{P}$ *are irrelevant. Thus by Lemma 8.6.9 we obtain that there is some type instantiation* $\mathbb{T}$ *such that* $\Gamma \vdash \mathbb{P}_P : \mathbb{T}$ *and* $\mathbb{T} \vDash_{\mathsf{L}} \mathring{P} : \Pi$*. Let* $\chi_0 = \mathbb{T}(\mathring{p})$ *and* $\Pi_0 = \langle \Gamma, \chi_0 \rangle$*. It holds that* $\vdash P_0 : \Pi_0$ *because* $\Gamma \vdash \mathbb{P}_P : \mathbb{T}$*. Also we know that* $\mathbf{active}\{\, \mathring{p} \ \mathbf{in}\ \mathring{P} \,\} \in \mathcal{R}$ *and thus* $\chi_0 \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$*. That is why* Π_0 *is an* $\mathcal{R}$*-type. Thus by the induction hypothesis we obtain that* $\vdash Q_0 : \Pi_0$*. Let* $\mathbb{P}_Q = \mathbb{P}[\mathring{p} \mapsto Q_0]$*. We see that* $\Gamma \vdash \mathbb{P}_Q : \mathbb{T}$ *holds. Hence the claim by Lemma 8.6.6.*

RPAR**:** *Then there are some* P_0*,* Q_0*, and* R_0 *such that* $P = P_0 \mid R_0$ *and* $Q = Q_0 \mid R_0$ *and* $P_0 \xhookrightarrow{\mathcal{R}} Q_0$*. We see that* $\vdash P_0 : \Pi$ *and* $\vdash R_0 : \Pi$*. By the induction hypothesis we have that* $\vdash Q_0 : \Pi$*. Hence the claim.*

RNU**:** *Then there are some name* x *and processes* P_0 *and* Q_0 *such that* $P = \nu x.P_0$ *and* $Q = \nu x.Q_0$ *and* $P_0 \xhookrightarrow{\mathcal{R}} Q_0$*. We see that* $\vdash P_0 : \Pi$*. By the induction hypothesis we obtain that* $\vdash Q_0 : \Pi$*. Hence the claim.*

RSTR**:** *Holds by induction hypothesis and Proposition 8.7.3.* ∎

Chapter 9

Changes and Extensions of Poly✶

9.1 Name Restriction

The main extension of the Poly✶ system in this thesis is the handling of name restriction. The name restriction handling presented here is based on a recent work of Jakubův and Wells [JW09, JW10]. However, the presentation style in this thesis is slightly different and this thesis contains additional details and proofs.

The difficulty with name restriction is that a shape type represents a syntactic structure of a process, and thus presence of bound names in a process has to be somehow reflected in a shape type. Because bound names can be α-renamed, Poly✶ needs to establish a connection between positions in a process and a shape type which is preserved by α-conversion. This connection is provided by type tags which are the key concept of name restriction handling in this thesis.

Let us suppose that some process P contains the form "a<a>". Then there has to be the corresponding form type "a<a>" in any shape type of P. When the name a in P were ν-bound and α-renamed to some other name then the correspondence between the form in the process and the form type would be lost. This problem is solved by building shape types from type tags which are preserved under α-conversion.

The handling of input-bound names in the previous Poly✶ was reached by disabling their α-conversion which is possible under the circumstances discussed in detail in Section 4.3. But α-conversion of ν-bound names can not be avoided and thus a different solution presented here has been developed. This solution allows us to handle α-conversion of ν-bound names and input-bound names uniformly which is much more intuitive than α-conversion in the previous Poly✶. Moreover, bound names are handled in the same way as free names when matching processes against shape predicates.

There are several alternative ways to support name restriction in shape types. We could, for example, allow α-renaming of bound names in types. In order to do

this it would be necessary to introduce the notion of the scope of a bound name in a shape type. Unfortunately, to introduce scopes into graphs is not straightforward because graphs can contain cycles which can make scopes of different binders (possibly binding the same name) overlap.

One possible solution to the above problem was given in an extension of POLY✶ from the 2004 technical report [MW04a, Section 5.3] which was designed to handle name restriction. In this 2004 extension, type tags are not used and both processes and shape types are built from the same atomic names. As noted above in Section 4.3, α-renaming of input-bound names is not allowed. The 2004 extension which handles name restriction defines *guarded shape predicates* (see also Remark 8.7.2) which are pairs of shape predicates and name sets. The name set of a guarded shape predicate can be seen as the set of names which are ν-bound in the shape graph. The scope of these bound names is simply the whole graph. This simple introduction of scopes into graphs is possible because of the definition of well formed processes and other restrictions. When matching a process against a guarded shape predicate, a ν-bound name from the process can match any ν-bound name from the graph.

Unfortunately, the 2004 POLY✶ extension was found broken as described in Remark 8.7.2. Furthermore, the solution presented in this thesis has other advantages. It allows us to handle α-renaming of bound names uniformly. More importantly, it is more expressive in the sense that it allows a simple embedding of type system with explicit types in POLY✶. We can now, for example, construct the embedding of explicitly typed Mobile Ambients [CG99] which is presented in Chapter 16. The difference in expressiveness can be demonstrated on the following simple example. The 2004 POLY✶ extension can not distinguish between the META✶ processes "$\nu x.x.\mathsf{0}$" and "$\nu y.y.\mathsf{0}$". They both have the same types under any circumstances. The distinction between the above two processes can be made in the POLY✶ from this thesis as long as x and y have different type tags. The ability to distinguish between the two processes becomes important when we want to embed a type system with explicit types in POLY✶. This is briefly described in the next paragraph.

In explicitly typed systems, bound names in processes are annotated with their types, for example, $(\nu x{:}\omega)P$ where ω is the type of x in P. When encoding processes with explicit type annotations as META✶ processes, it is desirable to simply drop these type annotations because they can not be straightforwardly encoded in META✶ syntax. A process encoding which drops type annotations gives us a faithful META✶ encoding of the original process calculus because the original rewriting relation does not usually inspect type annotations and it just copies them around. Nevertheless, a processes which differ only by type annotations can have different types. Let us consider some explicitly typed system where "$(\nu x : \omega_0)x.\mathsf{0}$" has some type ρ but "$(\nu y : \omega_1)y.\mathsf{0}$" has no type. These two processes are translated to the META✶

processes "$\nu x.x.0$" and "$\nu y.y.0$" from the previous paragraph. In order to faithfully embed the original explicitly typed system in POLY✶ we need POLY✶ to be able to recognize typability of the original processes on their META✶ equivalents. Thus to construct an embedding of this explicitly typed system, we need POLY✶ to be able to distinguish the above two processes. The POLY✶ from this thesis allows the embedding of explicitly typed systems by translating bound names with different type annotations to META✶ names with different type tags.

An alternative solution to embed explicitly type systems in POLY✶ would be to use some more complicated encoding of processes in META✶ which would not completely forget type annotations. We have thoroughly investigated this possibility but we have not found any satisfactory encoding. Any encoding of processes which tries to remember type annotations becomes opaque and hard to comprehend. The aspiration to simplify these encodings led us to the design of the POLY✶ version from this thesis which introduces type tags.

9.2 Changes from the Original POLY✶

This section summarizes changes between the previously published POLY✶ system [MW05, MW04a] and POLY✶ presented in Part I of this thesis.

Name Restriction. The main extension of the version presented in this thesis name is the support of name restriction. The previous POLY✶ version presented in the ESOP 2005 paper [MW05] does not support name restriction at all. There is, however, an extension presented in the 2004 technical report [MW04a, Section 5.3] which supports name restriction. Nevertheless this extension was found inconsistent. Details related to the support of name restriction in POLY✶ were presented in Section 9.1.

Fixes. This thesis fixes some mistakes from previous POLY✶ [MW05, MW04a]. The mistake in the definition of well formed processes and in application of substitutions is described in details in Section 4.3 and Section 4.5. This mistake breaks subject reduction which is discussed in details in Remark 8.7.2. Furthermore, requirements on the rewriting rules defined in the previous POLY✶ [MW04a, Section 5.1] did not ensure preservation of well formedness as described in Section 6.2 and Remark 8.7.2. These requirements were previously stated only informally and several points can be interpreted ambiguously. The requirements are formalized and described in Section 6.2 of this thesis. Last but not least least, references to undefined operations (for example "$[x := y]$" in [MW05, Figure 2]) are fixed in this thesis.

Clarifications. Introduction of type tags allows a uniform handling of ν-bound and input-bound names. Previously, input-bound names were not allowed to be α-

converted (see Section 4.3) and ν-bound names were not part on the set BN(P) of the bound names of a process (see Section 4.5). This unintuitive behavior was probably the main cause of the mistake in the definition of well formed processes mentioned above. Another change is that an implicit definition of α-equivalence from previous POLY✶ was in this thesis replaced by a formal definition (Section 4.2). Substitution now guards against name captures which makes the behavior of POLY✶ more predictable (see Section 4.5). Introduction of type tags also allows slightly simpler definition of type substitution application.

Proofs. Previous POLY✶ publications [MW05, MW04a] contain no proofs except of a very short (1 page) proof sketch of subject reduction. In this thesis we prove crucial properties of the system including preservation of well-formedness (Proposition 6.4.2), substitution correctness (Proposition 8.2.2), flow closure correctness (Proposition 8.5.1), subject reduction (Theorem 7.6.11), and existence of principal typings (Theorem 12.10.3.

9.3 Possible Extensions and Future Work

In this section we describe some possible extensions of the POLY✶ system. They include (1) additional process operators which are found in other calculi (recursion "μ" and the choice "+"), (2) possibilities to increase shape type expressiveness, and (3) a way to produce smaller shape graphs with the equivalent meaning by the type inference algorithm.

9.3.1 Recursion and the μ Operator

The current version of POLY✶ provides the replication operator "!" which can be used to implement recursive behavior of processes. Several other constructions used to implement recursive behavior are found in the literature [PV05]. These include the μ operator (also called **rec**), **let** expressions, and constant or parametric definitions. Replication can always be expressed by one of the above recursive constructions. In general, however, the expressive power of different constructions varies among different calculi [PV05]. For example in the π-calculus, replication can be used to encode the μ operator as well as parametric definitions [Par01, Section 3.4]. On the other hand, the same encoding can not be straightforwardly adapted to work in Mobile Ambients and other calculi which contain ambient boundaries.

Now we show how the μ operator can be simply emulated in many instantiations of META✶ using additional rewriting rules. The μ operator is of interest because in many calculi it is more expressive than replication as well as it is more convenient

to express recursive behavior. For example, it is very often used in biologically inspired calculi. The processes calculi with the μ operator usually defines a set of process variables. Let X range over process variables. The process syntax is then defined so that every process variable X and every construction of the shape "$\mu X.P$" are processes. Then a process substitution to substitute a process Q for process variable X in process P, written $P\{X \mapsto Q\}$, is defined. The process variable X is (μ-)bound in $\mu X.P$ and application of a process substitution has to guard against name and variable captures. The process "$\mu X.P$" is supposed to behave as the process "$P\{X \mapsto \mu X.P\}$". A replicated process "$!P$" can be expressed using the μ operator as "$\mu X.(P \mid X)$".

The semantics of the μ operator can be defined in structural equivalence by the axiom

$$\mu X.P \equiv P\{X \mapsto \mu X.P\}$$

or in the rewriting relation, either as a separate rewriting step by the axiom

$$\mu X.P \to P\{X \mapsto \mu X.P\}$$

or incorporated into other rewriting steps by the inference rule

$$\frac{P\{X \mapsto \mu X.P\} \mid Q \to R}{(\mu X.P) \mid Q \to R}$$

We can translate META✶ processes with the μ operator to standard μ-free META✶ processes as follows. For simplicity, let us suppose that process variables are taken from the set of names. The translation encoding that removes μ works as follows.

$$(\!|X|\!) = \mathsf{call}\ X.\mathsf{0}$$
$$(\!|\mu X.P|\!) = \nu X.(\mathsf{call}\ X.\mathsf{0} \mid !\mathsf{rec}\ X.(\!|P|\!))$$

Purely structural cases like $(\!|P \mid Q|\!) = (\!|P|\!) \mid (\!|Q|\!)$ are omitted. The names "rec" and "call" are ordinary META✶ names (more precisely type tags) and we suppose that they are in SpecialTag. The basic idea is to store the body P of a recursively defined process $\mu X.P$ using the replicated process "$!\mathsf{rec}\ X.(\!|P|\!)$". The process variable X as a process is encoded by the process "$\mathsf{call}\ X.\mathsf{0}$". The replacement of the variable by the process body, that is, the unfolding of the definition, is implemented by the following rewriting rule which is to be added to the set of rules.

$$\mathbf{rewrite}\{\,\mathsf{call}\ \mathring{\mathsf{x}}.\mathsf{0} \mid \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{P}} \hookrightarrow \mathring{\mathsf{P}}\,\}$$

This translation works without any problems in process calculi without any

active rules like the π-calculus. However, in process calculi with more active positions it can happen the body "rec $X.([P])$" of a definition and the request "call $X.0$" to unfold the definition appear at different active positions. For example in Mobile Ambients, they can be present in different ambients and thus the above rewriting rule can not be applied to unfold the definition properly. Consider, for example, "$\mu X.\mathsf{a}[X \mid \mathsf{out}\ \mathsf{a}.0]$" where the above rule can be used to unfold the definition at the top-level location but not inside any ambient.

A possible solution of this problem is to make the process "rec $X.([P])$" with the definition body appear inside all active ambients. This can be achieved by adding the following two rewriting rules which distributes the definition body both in and out of any ambient.

$$\textbf{rewrite}\{\ \mathring{\mathsf{a}}[\mathring{\mathsf{P}}]\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}} \hookrightarrow \mathring{\mathsf{a}}[\mathring{\mathsf{P}}\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}}]\ \}$$
$$\textbf{rewrite}\{\ \mathring{\mathsf{a}}[\mathring{\mathsf{P}}\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}}] \hookrightarrow \mathring{\mathsf{a}}[\mathring{\mathsf{P}}]\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}}\ \}$$

In META✶ instantiations which use user defined **active** rules, two rules similar the above have to be added for every active rule. This can not, however, be done for an active rule which mentions input-bound variable, like **active**$\{\ \mathring{\mathsf{P}}$ **in** $(\mathring{\mathsf{x}}).\mathring{\mathsf{P}}\ \}$. The reason is that the rule corresponding to this active rule

$$\textbf{rewrite}\{\ (\mathring{\mathsf{y}}).\mathring{\mathsf{P}}\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}} \hookrightarrow (\mathring{\mathsf{y}}).(\mathring{\mathsf{P}}\ \mid\ \mathsf{rec}\ \mathring{\mathsf{x}}.\mathring{\mathsf{Q}})\ \}$$

is not allowed by the META✶ syntax because it is not well formed (see Section 6.2). It is not allowed because it can create a nested input-binders which bind the same type tag which is not allowed by well-formedness condition W2. Nevertheless, we do not know of any process calculi in the literature that would require active rules with input-bound names. Thus this encoding of the μ operator seems to be sufficient.

Based on the above discussion we can claim that it would be possible to support the μ operator in META✶ directly as a built-in operator. This might be preferable in some situations because the above encoding can lead to more complicated shape types as they need to contain auxiliary edges introduced by the encoding. The μ operator can be directly supported in POLY✶ under the following circumstances. The **active** rules can not contain any input-bound variables. Furthermore, a recursive processes $\mu X.P$ can be unfolded to $P\{X \mapsto \mu X.P\}$ only when it appears at active position, that is, only one level a time. The above encoding satisfies this property which is again necessary to ensure that no nested input-binders created by the unfolding bind the same type tag.

9.3.2 The Choice Operator

Another process operator that is commonly used by process calculi is the *choice operator* "+" also called *alternative composition*. The process "$P + Q$" describes a process which behaves like P or like Q but not both of them. That is, at some point a decision is made whether the process "$P + Q$" will behave like the part P or Q, and the other part is discarded. The choice operator is commonly used in biologically inspired calculi and in the π-calculus.

In META✶ we can simply choose a special name "ch" and use it to encode "$P+Q$" as "ch.$(P \mid Q)$". Then we can adapt the rewriting rules to respect this encoding appropriately. For example, the monadic π-calculus with choice can be expressed by the following rule description.

$$\mathcal{P}_{\mathsf{choice}} = \{\textbf{rewrite}\{\, \mathsf{ch}.(\mathring{\mathsf{C}} \mid \mathring{\mathsf{c}}\texttt{<}\mathring{\mathsf{a}}\texttt{>}.\mathring{\mathsf{P}}) \mid \mathsf{ch}.(\mathring{\mathsf{D}} \mid \mathring{\mathsf{c}}(\mathring{\mathsf{x}}).\mathring{\mathsf{Q}}) \hookrightarrow \mathring{\mathsf{P}} \mid \{\mathring{\mathsf{x}} := \mathring{\mathsf{a}}\}\mathring{\mathsf{Q}}\,\}\}$$

The choice is associative and we assume that "$(P+Q)+R$" is encoded as "ch.$(P \mid Q \mid R)$" and not as "ch.$(\mathsf{ch}.(P \mid Q) \mid R)$". Also a standalone process with not alternatives like "c<a>.0" has to be encoded as "ch.c<a>.0". We can see the encoding of choice with the special name "ch" results in the presence of additional edges in shape types and thus it makes shape types more complicated.

Chapter 10 introduces additional restrictions on shape types which are required for type inference. These restrictions, namely the depth restriction (Section 10.2), would cause unnecessary over-approximation in shape types. It would be helpful to change the width restriction so that it handles the name "ch" in a special way. However, an advanced handling of the choice operator that would not unnecessarily complicate shape types and that would take the specific behavior of "+" into account is left for future research.

9.3.3 Other Extensions

There are several possibilities to improve expressiveness of shape types. One of them is to extend the syntax of META✶ sequences and messages (and related POLY✶ types) so that it allows more convenient encoding of calculi which communicate structural messages like the spi calculus [AG99]. Currently, we could use META✶ messages to encode the spi calculus structured messages but the problem is that the message structure can not be described by POLY✶ message types. For example, the message type "$\{\mathsf{a}, \mathsf{b}\}*$" is a type of "(a.b).(a.b)" as well as of "a.(b.a).b". Thus POLY✶ shape types for processes from calculi which heavily rely on structural messages would not be very precise. A possible solution would be to extend the syntax of shape graphs to allow more expressive message types. This solution would probably require to represent message types themselves by graphs and not just by

linear structures.

An improved version of message types was proposed in the 2004 technical report [MW04a, Section 5.2] under the name "sequenced message types". A sequenced message type has the shape "$\sigma_0.\cdots.\sigma_k$" for some sequence types σ_0, ..., σ_k. The sequenced message type "$\sigma_0.\cdots.\sigma_k$" matches all the messages which have exactly $k+1$ non-null sequence parts s_0, ..., s_k such that $\vdash s_i:\sigma_i$ where the parts are sorted as they occur in the message from left to right. Sequenced message types provide information about the count and the order of META✶ sequences in the message but not about its spatial "tree" structure. For example, the sequenced message type "a.b.c" matches both "(a.b).c" and "a.(b.c)" but not "b.a.c" or "a.b.b.c". In order to obtain the principal typing property (see Chapter 10) it is necessary that no sequenced message type contain two identical sequence types. For example, "a.a" is banned as a sequenced message type. It would be probably easy to extend the POLY✶ from this thesis to work with the sequenced message types from the 2004 technical report. The proofs of the subject reduction and the correctness of the type inference algorithm would need to be extended as well.

Another two extensions of POLY✶ are proposed in the 2004 technical report. The first one, target borrowing [MW04a, Section 5.5], is and optimization of a type inference algorithm which reduces the size of shape graphs by sharing edges. Only edges whose sharing does not change the meaning of the shape predicate are shared.

The second extension [MW04a, Section 5.6], increases the precision of shape types using atomic labels called *marks*. Marks are used to recover precision which is lost by additional restrictions on shape graphs which are necessary to achieve the principal typing property (see Chapter 10).

Both the two above extensions are briefly described in the 2004 technical report and implemented in the testing type inference algorithm implementation which belongs to the 2004 report. However, the POLY✶ 2004 theory was not extended to work with these extensions and no correctness results were proved. The integration of these extensions into the POLY✶ theory and extensions of the proofs from this thesis is left for the future research.

Part II

Type Inference

Chapter 10

Principal Typings

10.1 Principal Typings and Types

A principal type of a process P is the type which is some sense the "most general" among all the types of P. Wells [Wel02] provides a general definition of principal typings that works for many type systems. Wells distinguishes between *types* and *typings*. A typing is a collection of all the information other than the process (term) that appear in type statements. Usually it is the process (term) type and the environment or context which determines types of free names (variables). Shape type statements $\vdash P{:}\Pi$ do not use any environment or context and thus typings in Poly$\bigstar$ become equivalent with shape types. Henceforth, we use the notions "principal types" and "principal typings" interchangeably when they apply to Poly$\bigstar$.

The general definition of principal typings [Wel02] becomes the following when specialized to Poly$\bigstar$.

Definition 10.1.1. *Call an $\mathcal{R}$-type Π of P* **principal** *(among $\mathcal{R}$-types) when for any Π'*

$$\mathcal{R} \Vdash_{\mathsf{type}} \Pi' \;\&\; \vdash P : \Pi' \qquad \textit{implies} \qquad \Pi \preccurlyeq \Pi'.$$

Let Π be a principal $\mathcal{R}$-type of P. The meaning of Π is included in the meaning of any other $\mathcal{R}$-type of P. In this sense Π provides the most specific information about P among all other $\mathcal{R}$-types of P. Furthermore, we can say that Π represents all other $\mathcal{R}$-types of P. In this sense Π is the most general among all other $\mathcal{R}$-types of P.

The existence of principal typings, which is called the *principal typing property*, is a desirable property of type systems for programming languages. It supports compositional automated type inference and it allows reusability of type inference results. A principal typing is a natural output of a type inference algorithm for a type system with the principal typing property. Furthermore, the existence of principal typings in Poly$\bigstar$ allows us to use Poly$\bigstar$ instead of another type system.

We demonstrate this is Chapter 14 where we prove that POLY✶ shape types can be used to precisely recognize processes typable in the π-calculus sort discipline [Mil99].

In POLY✶, the existence of principal $\mathcal{R}$-types depends on $\mathcal{R}$. We are not aware of any procedure to recognize rule descriptions $\mathcal{R}$ that instantiate POLY✶ to type systems with the principal typings property. In Section 10.3, we show that instantiations of POLY✶ with some infinite rule descriptions $\mathcal{R}$ do not have the principal typing property. Nevertheless, the finiteness of $\mathcal{R}$ does not ensure the principal typings property either. Let us consider to following rule description.

$$\mathcal{R} = \{\textbf{active}\{\,\mathring{\mathsf{P}}\ \textbf{in}\ \mathsf{a}.\mathring{\mathsf{P}}\,\}, \textbf{rewrite}\{\,\mathsf{a.middle.a}.\mathring{\mathsf{P}} \hookrightarrow \mathsf{a.a.middle.a.a}.\mathring{\mathsf{P}}\,\}\}$$

Section 10.4 proves there is no principal $\mathcal{R}$-type of the process "$\mathsf{a.middle.a}.0$".

Principal types in POLY✶ are not unique because for any shape predicate Π there is infinitely many of shape predicates with the same meaning. For example, renaming of nodes in Π preserves meaning but there are more complex graph operations which preserves the meaning as well, for example unification of all terminal nodes. Of course all the principal types of P have to have the same unique meaning.

We do not know how to compute principal $\mathcal{R}$-types for those rules $\mathcal{R}$ for which principal $\mathcal{R}$-types exist. This is because the set of all $\mathcal{R}$-types is too complex. However, in the next section we define a subset of *restricted* shape $\mathcal{R}$-types so that the existence of principal $\mathcal{R}$-types among restricted types can be proved. At the same time, restricted shape types maintain to be expressive enough for practical use as clearly demonstrated in Part III of this thesis.

10.2 Restricted Shape Types

As noted in the previous section, we do not know how to do a type inference which outputs a principal $\mathcal{R}$-type for an arbitrary $\mathcal{R}$. The set of all $\mathcal{R}$-types is too big and complex for this task to be easily achieved. Instead of trying to characterize descriptions $\mathcal{R}$ which instantiates POLY✶ to the type system with the principal typing property and instead of looking for a complete type inference algorithm for these $\mathcal{R}$ we apply the following, much simpler, approach. We define a subset of $\mathcal{R}$-types, called *restricted* $\mathcal{R}$-types, by restricting the structure of shape graphs. The existence of principal types among restricted $\mathcal{R}$-types can be proved for all finite $\mathcal{R}$ and for all infinite $\mathcal{R}$ which are of interest. Details about handling of infinite rule descriptions are found in Section 10.3.

At first we define the *similarity* relation "$\approx$" on form types as follows.

DEFINITION 10.2.1. *Form types φ_0 and φ_1 are called* **similar**, *written $\varphi_0 \approx \varphi_1$, iff* $[\![\varphi_0]\!] \cap [\![\varphi_1]\!] \neq \emptyset$. ∎

The $\approx$ relation is close to being the equality on form types. The only way for non-identical φ's to be related by $\approx$ is when one of them contains a starred message type $\Sigma*$. It is relatively safe to image $\approx$ to be equality ($=$), at least to the first approximation. It is necessary to take this relation instead of equality ($=$) in two definitions below in order to achieve the principal typing property.

In order to achieve the principal typings property, we restrict the number of nodes in shape graphs. The *width restriction* says that two edges outgoing from the same source which are labeled with similar (that is, related by "$\approx$") form types have the same destination node. This restriction makes rule TFRM of the type checking relation "$\vdash$" (and also rule CFRM of "$\models_s$") deterministic, because when $\vdash F.P_0 : \langle \Gamma, \chi \rangle$ then the node χ_0 (from TFRM) such that $\vdash P_0 : \langle \Gamma, \chi_0 \rangle$ holds is uniquely determined.

DEFINITION 10.2.2. *We say that a shape graph Γ is* **width-restricted** *or that Γ satisfies the* **width restriction** *iff whenever there are two edges $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ and $(\chi \xrightarrow{\varphi'} \chi_1) \in \Gamma$ with $\varphi \approx \varphi'$, then it holds that $\chi_0 = \chi_1$. A shape predicate $\langle \Gamma, \chi \rangle$ is width-restricted when Γ is.* ∎

Before we define the second restriction on shape graphs we define a path in shape predicate quite naturally as follows.

DEFINITION 10.2.3. *A* **path** *in $\langle \Gamma, \chi \rangle$ is a set of linearly connected edges*

$$\{\chi_0 \xrightarrow{\varphi_1} \chi_1, \chi_1 \xrightarrow{\varphi_2} \chi_2, \ldots, \chi_{k-1} \xrightarrow{\varphi_k} \chi_k\} \subseteq \Gamma$$

which we write as $\{\chi_0 \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi_k\}$. A path is **rooted** *when $\chi_0 = \chi$.* ∎

The *depth restriction* says that any two edges labeled with similar form types which lie on the same path have the same destination node. When an upper bound on form types that can appear in a graph is given, then the depth restriction bounds the total number of edges that a restricted shape graph can have. This fact will become the main argument for the termination of the type inference algorithm presented in Chapter 11.

DEFINITION 10.2.4. *We say that a shape graph Γ is* **depth-restricted** *or that Γ satisfies the* **depth restriction** *iff whenever Γ contains a path $\{\chi_0 \xrightarrow{\varphi_0} \chi_1 \xrightarrow{\varphi_1} \cdots \chi_k \xrightarrow{\varphi_k} \chi_{k+1}\} \subseteq \Gamma$ with $\varphi_0 \approx \varphi_k$, then it holds that $\chi_1 = \chi_{k+1}$. A shape predicate $\langle \Gamma, \chi \rangle$ is depth-restricted when Γ is.* ∎

The width and the depth restrictions do not depend on rule description $\mathcal{R}$. Restricted $\mathcal{R}$-types are now defined quite naturally as those $\mathcal{R}$-types which are also restricted.

DEFINITION 10.2.5. *A shape predicate* Π *is* **restricted** *when* Π *is both width- and depth-restricted. A shape predicate* Π *is a* **restricted $\mathcal{R}$-type**, *written* $\mathcal{R} \Vdash_{\mathsf{restr}} \Pi$, *when* Π *is restricted and* $\mathcal{R} \Vdash_{\mathsf{type}} \Pi$. ∎

Finally we defined principal restricted $\mathcal{R}$-types by restraining the general definition of principal typings only to restricted $\mathcal{R}$-types.

DEFINITION 10.2.6. *Call an* $\mathcal{R}$*-type* Π *of* P **principal among restricted types** *or a* **principal restricted $\mathcal{R}$-type** *iff for any* Π'

$$\mathcal{R} \Vdash_{\mathsf{restr}} \Pi' \;\&\; \vdash P : \Pi' \qquad \textit{implies} \qquad \Pi \preccurlyeq \Pi'.$$

In Chapter 11 we provide an effective type inference algorithm to compute a principal restricted $\mathcal{R}$-type for an arbitrary P. The proof of correctness (completeness) of this algorithm provides a constructive proof of the existence of principal types among restricted types.

10.3 Infinite Sets of Rewriting Rules

Infinite sets of rewriting rules are required to handle polyadic communication rules, that is, communication rules which can send tuples of an arbitrary arity. As an example we can take the polyadic π-calculus which is described in META✶ by the following infinite set of rewriting rules.

$$\mathcal{P}_{\mathsf{poly}} = \big\{\mathbf{rewrite}\{\,\mathring{c}\texttt{<}\mathring{a}_1,\ldots,\mathring{a}_n\texttt{>.}\mathring{P} \mid \mathring{c}\texttt{(}\mathring{x}_1,\ldots,\mathring{x}_n\texttt{).}\mathring{Q} \hookrightarrow \mathring{P} \mid \{\mathring{x}_1 := \mathring{a}_1,\ldots \mathring{x}_n := \mathring{a}_n\}\mathring{Q}\,\} : n \geqslant 0\big\}$$

It would be possible to extend the syntax of META✶ process templates so that the above can be described by a single rule. This single rule might look as follows.

$$\mathbf{rewrite}\{\,\mathring{c}\texttt{<}\mathring{a}\ldots\texttt{>.}\mathring{P} \mid \mathring{c}\texttt{(}\mathring{x}\ldots\texttt{).}\mathring{Q} \hookrightarrow \mathring{P} \mid \{\mathring{x}\ldots := \mathring{a}\ldots\}\mathring{Q}\,\}$$

The semantic of the above rule would need to decide what should happen for elements "$\texttt{<}\mathring{a}\ldots\texttt{>}$" and "$\texttt{(}\mathring{x}\ldots\texttt{)}$" with different lengths. Instead of extending the template syntax, we prefer to use infinite sets of rewriting rules and to keep the language of templates as simple as possible. Any actual implementation of POLY✶ would need to use some extended rule syntax as the above to handle polyadic communication rules.

We start with the observation that the principal typing property does not hold for some infinite rule descriptions. Let us consider the following infinite set of rewriting rules.

$$\mathcal{R}_0 = \{\mathbf{rewrite}\{\,\texttt{<}\mathring{x}\texttt{>.0} \hookrightarrow \texttt{<}\mathring{x},\mathring{x}\texttt{>.0}\,\}, \mathbf{rewrite}\{\,\texttt{<}\mathring{x},\mathring{x}\texttt{>.0} \hookrightarrow \texttt{<}\mathring{x},\mathring{x},\mathring{x}\texttt{>.0}\,\}, \ldots\}$$

It is easy to see that there are processes with no type at all not to say principal types. In the case of $\mathcal{R}_0$ it is, for example, "<a>.0". We can see that "<a>.0 $\overset{\mathcal{R}_0}{\hookrightarrow}$ <a, a>.0 $\overset{\mathcal{R}_0}{\hookrightarrow}$ <a, a, a>.0 $\overset{\mathcal{R}_0}{\hookrightarrow} \cdots$" and thus any $\mathcal{R}_0$-type of "<a>.0" has to contain the edge labeled with the form type "<a, ..., a>" for any possible arity of the output element type. It means that any $\mathcal{R}_0$-type of "<a>.0" would need to contain infinitely many number of edges which is not possible because shape graphs are finite.

In order to implement an effective type inference algorithm it is essential that for any $\mathcal{R}$ and P there is only finitely many rules in $\mathcal{R}$ that can ever be used when rewriting P (and its successors). We can see that one of the problems with $\mathcal{R}_0$ is that the right hand side of any rule from $\mathcal{R}_0$ generates an element which is longer than any element mentioned on the rule left-hand side. The first step to handle infinite rule descriptions is to eliminate "non-monotonic" rules like those from $\mathcal{R}_0$. In order to do that we formally define the length of a META✶ entity as follows.

DEFINITION 10.3.1. *The length of a* META✶ *entity is defined as follows.*

(1) a name "x" has the length 1
(2) a sequence "$x_0 \dots x_k$" has the length $k+1$
(3) an input element "(x1, ..., xk)" has the length k
(4) an output element "<M1, ..., Mk>" has the length k
(5) a form "$E_0 \dots E_k$" has the length $k+1$
(6) an input element template "($\mathring{x}_1, \dots, \mathring{x}_k$)" has the length k
(7) an output element template "<$\mathring{m}_1, \dots, \mathring{m}_k$>" has the length k
(8) a form template "$\mathring{E}_0 \dots \mathring{E}_k$" has the length $k+1$
(9) a substitution template "{$\mathring{x}_0$:= $\mathring{s}_0, \dots, \mathring{x}_k$:= $\mathring{s}_k$}" has the length $k+1$
(10) any other META✶ *entity has the length* 0

Let maxlen(P) *be the maximum of the lengths of all* META✶ *entities in P. Similarly, let* maxlen($\mathcal{R}$) *(*maxlen($\mathring{L}$), maxlen($\mathring{P}$) *respectively) be the maximum of the lengths of all* META✶ *entities in $\mathcal{R}$ ($\mathring{L}$, $\mathring{P}$ respectively).* ∎

Note that a composed message has the length 0 and thus its depth does not affect maxlen(P). For example the longest entity in the process

set <out a.in b.open c.in a>.get (x, y, z).0

is the element "(x, y, z)" with the length 3.

Now we can formally define *monotonic* rewriting rules.

DEFINITION 10.3.2. *Call a rule description $\mathcal{R}$* **monotonic** *iff for any rewriting rule* **rewrite**{ $\mathring{P} \hookrightarrow \mathring{Q}$ } *from $\mathcal{R}$ it holds that* maxlen($\mathring{Q}$) $\leqslant$ maxlen($\mathring{P}$). ∎

But the monotony of $\mathcal{R}$ is still not enough to ensure principal typings. When SpecialTag is infinite then we can construct the following well formed rule description.

$$\mathcal{R}_1 = \{\mathbf{rewrite}\{\, \mathsf{a}.0 \hookrightarrow \iota^0.0 \,\}\colon \iota \in \text{SpecialTag}\}$$

We can see that there is no $\mathcal{R}_1$-type of "$\mathsf{a}.0$". The requirement of the finiteness of SpecialTag is probably enough to ensure the existence of restricted principal types. However, it still does not ensure that there is only finitely many rewriting rules that can ever be used when rewriting a given process. Let us consider the following rule set which contains lot of redundant rules.

$$\mathcal{R}_2 = \{\mathbf{rewrite}\{\, \mathsf{a}\ \mathring{x}.0 \hookrightarrow \mathring{x}\ \mathsf{a}.0 \,\}\colon \mathring{x} \in \text{NameVar}\}$$

It is clear that $\mathcal{R}_2$ contains infinitely many of rules (because NameVar is infinite) that can apply to the process "$\mathsf{a}\ \mathsf{a}.0$". To select a finite subset of any $\mathcal{R}$ which can ever be used when rewriting a given P is required to effectively iterate over $\mathcal{R}$ in the type inference algorithm and in the algorithm to recognize $\mathcal{R}$-types.

Now we define a subset of rule descriptions which we call *standard*. We call them standard because, to our best knowledge, all the process calculi from the literature which can be described in META✶ syntax are covered in this subset. That is to say, that this property is "standard" for the rules in the literature.

DEFINITION 10.3.3. *A rule description $\mathcal{R}$ is* **standard** *iff $\mathcal{R}$ is monotonic and for every k natural it holds that $\{\mathring{L} \in \mathcal{R}\colon \text{maxlen}(\mathring{L}) \leqslant k\}$ is finite.* ∎

For all standard $\mathcal{R}$, the existence of restricted principal $\mathcal{R}$-types is proved in Chapter 11. The standard condition on $\mathcal{R}$ is sufficient but not necessary to ensure the existence of restricted principal $\mathcal{R}$-types. There are, for example, rule sets which are not monotonic and still instantiates POLY✶ to the type system with restricted principal types. We are not, however, aware of any calculus from the literature which uses non-standard rewriting rules. Thus the exact characterization of infinite rule descriptions which instantiate POLY✶ to the type system with the existence of principal restricted types is left for the future research.

10.4 Non-existence of Principal Types Among Unrestricted Types

In this section we construct a rewriting rule description $\mathcal{R}$ which is finite and well formed, and which instantiates POLY✶ to a type system without principal types among all $\mathcal{R}$-types. Principal typings still exist among restricted $\mathcal{R}$-types, however. From this we can conclude that we really need to work with restricted types to have principal typings for an arbitrary $\mathcal{R}$.

PROPOSITION 10.4.1. *Let us consider the following rewriting rule set* $\mathcal{R}$.

$$\mathcal{R} = \{\mathbf{active}\{\, \mathring{\mathsf{P}} \ \mathbf{in}\ \mathsf{a}.\mathring{\mathsf{P}}\,\}, \mathbf{rewrite}\{\, \mathsf{a.mid.a.}\mathring{\mathsf{P}} \hookrightarrow \mathsf{a.a.mid.a.a.}\mathring{\mathsf{P}}\,\}\}$$

There is no $\mathcal{R}$*-type of the process "*$\mathsf{a.mid.a.0}$*" which is principal (among all* $\mathcal{R}$*-types).*

PROOF. *Proof by contradiction. Let* Π *be a principal* $\mathcal{R}$*-type of "*$\mathsf{a.mid.a.0}$*". Let us define the infinite sequence of processes* P_1^{ok}, P_2^{ok}, ..., *as follows.*

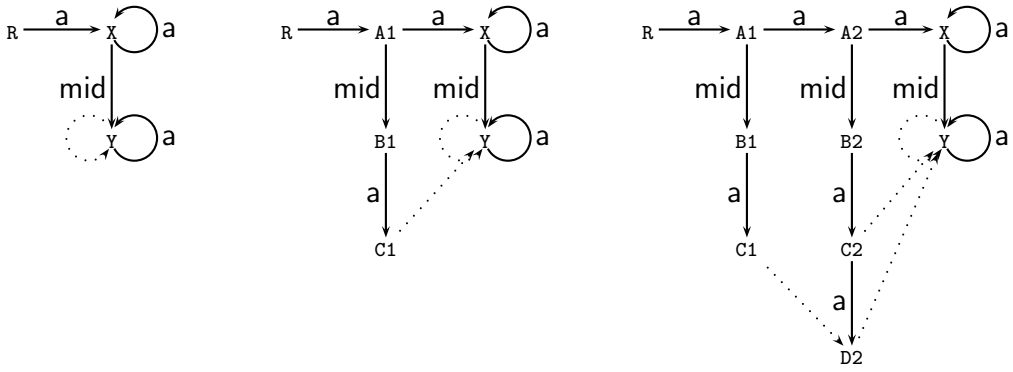
$$P_i^{\mathsf{ok}} = \underbrace{\mathsf{a.a.}\cdots\mathsf{.a}}_{i\ times}\mathsf{.mid.}\underbrace{\mathsf{a.a.}\cdots\mathsf{.a}}_{i\ times}\mathsf{.0}$$

We can see that "$P_1^{\mathsf{ok}} = \mathsf{a.mid.a.0}$*" and that the following rewritings can be proved.*

$$P_1^{\mathsf{ok}} \overset{\mathcal{R}}{\hookrightarrow} P_2^{\mathsf{ok}} \overset{\mathcal{R}}{\hookrightarrow} P_3^{\mathsf{ok}} \overset{\mathcal{R}}{\hookrightarrow} \cdots$$

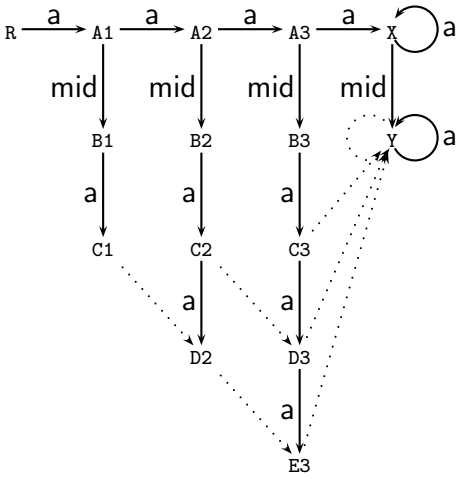
Hence $\vdash P_n^{\mathsf{ok}} : \Pi$ *for any* n *because* Π *is an* $\mathcal{R}$*-type of* P_1^{ok}.

Now let us define the infinite sequence of shape predicates Π_0, Π_1, ... *as depicted by the three first members.*



Π_0 Π_1 Π_2

For $n > 0$, *the shape predicate* Π_n *is constructed from* Π_{n-1} *by moving the column with nodes* `X` *and* `Y` *one place right, and by adding the additional column of nodes starting with node "*`An`*". The subtyping edges connect nodes in each column to the corresponding nodes in the next column for the shape predicate to be an* $\mathcal{R}$*-type. The subtyping edges from the last but one column all aim to node* `Y`. *For example, the next shape predicate* Π_3 *looks as follows.*



For any Π_n*, we can see that* $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi_n) = \{\mathtt{R}, \mathtt{A1}, \mathtt{A2}, \ldots, \mathtt{A}n, \mathtt{X}\}$*. Now it is easy to check that any* Π_n *is an* $\mathcal{R}$*-type. For example, let us consider* Π_3*. When we apply the only* **rewrite** *rule from* $\mathcal{R}$ *at the node* `A1` *we can see that the local* $\mathcal{R}$*-closure condition requires the presence of the subtyping edge* $\mathtt{C2} \dashrightarrow \mathtt{D3}$*. Thus* Π_3 *is locally closed at* `A1` *because this edge is present in the shape graph. Similarly, we can see that* Π_3 *is locally* $\mathcal{R}$*-closed at all the active nodes* $\{\mathtt{R}, \mathtt{A1}, \mathtt{A2}, \mathtt{A3}, \mathtt{X}\}$ *and hence* Π_3 *is an* $\mathcal{R}$*-type because it is clearly flow-closed. Finally it is trivial to check that* $\vdash P_1^{\mathsf{ok}} : \Pi_n$ *hold for any* n *and thus by the subject reduction we obtain that* $\vdash P_i^{\mathsf{ok}} : \Pi_n$ *for any* i *and* n (> 0)*.*

Now let us definite the infinite sequence of processes P_1^{ko}*,* P_2^{ko}*, ... as follows.*

$$P_i^{\mathsf{ko}} = \underbrace{\mathsf{a}.\mathsf{a}.\cdots.\mathsf{a}}_{i \text{ times}}.\mathsf{mid}.\underbrace{\mathsf{a}.\mathsf{a}.\cdots.\mathsf{a}}_{(i+1) \text{ times}}.0$$

Process P_i^{ko} *is process* P_i^{ok} *with one additional* a *at the end. We can see that the process* P_1^{ok} *can never rewrite to any of the processes* P_n^{ko}*. The key observation here is that*

$$\text{for any } n > 0 : \quad \not\vdash P_n^{\mathsf{ko}} : \Pi_n.$$

Now we can finally prove that Π *can not be a principal type of* P_1^{ok}*. Let* j *be the number of edges in* Π*. Now we shall prove* $\vdash P_j^{\mathsf{ko}} : \Pi$*. We know that* $\vdash P_j^{\mathsf{ok}} : \Pi$ *because* Π *is an* $\mathcal{R}$*-type of* P_1^{ok} *and* P_1^{ok} *can be rewritten to* P_j^{ok}*. Clearly at least one edge in* Π *has to be labeled with* mid *and thus the number of edges in* Π *which are labeled with* a *is strictly smaller than* j*. Thus some edge has to be used more than once when matching the second part of* P_j^{ok} *after* mid *against* Π *because the number of* a*'s* $(= j)$ *is bigger than the number of edges labeled with* a $(< j)$*. Hence these edges have to form a cycle which can be used to match an arbitrary number of* a*'s after the middle* mid*. More specifically we obtain* $\vdash P_j^{\mathsf{ko}} : \Pi$*.*

Hence Π *is not a principal* $\mathcal{R}$*-type of* P_1^{ok} *because* Π_j *is an* $\mathcal{R}$*-type of* P_1^{ok} *and we have that* $\vdash P_j^{\mathsf{ko}} : \Pi$ *but* $\not\vdash P_j^{\mathsf{ko}} : \Pi_j$*.* ∎

Note that the above proof can not be easily adapted to work with some description of an existing process calculus, for example $\mathcal{A}_{\mathsf{mon}}$. The reason is that flow and local closure conditions already prevent many $\mathcal{A}_{\mathsf{mon}}$-closed shape predicates from being $\mathcal{A}_{\mathsf{mon}}$-types. For example, the flow closure condition F1 applied to the graph $\{\mathtt{B0} \xrightarrow{\mathsf{x}} \mathtt{B1}, \mathtt{B0} \xrightarrow{\{\mathsf{x} \mapsto \{\mathsf{in\ a}\}*\}} \mathtt{R}\}$ insists on the presence of a loop labeled with "in a" at node R. It explains why the shape predicate from Example 7.5.2 is not an $\mathcal{A}_{\mathsf{mon}}$-type. Furthermore this precludes us from adapting the above proof to work with $\mathcal{A}_{\mathsf{mon}}$.

Chapter 11

Type Inference

In this chapter we present a formal description of the type inference algorithm. This implementation is supposed to provide a constructive proof of the existence of principal types. Thus at many places we prefer a less effective but simpler algorithm in order to make correctness proofs easier. On the other hand, this algorithm clearly depicts a basic idea of type inference and can be turned into an effective implementation by the use of more sophisticated data structures and common programming techniques. Our aim in this thesis is, however, to prove the existence of principal types.

11.1 Overview of the Type Inference Algorithm

The basic informal skeleton of the type inference algorithm is depicted in the following Algorithm 11.1.

Algorithm 11.1: Informal description of the type inference algorithm

input : a process P and a standard rule description $\mathcal{R}$
output: a principal $\mathcal{R}$-type of P

```
1 Π := the initial shape predicate directly corresponding to P;
2 while Π is not an R-type do
3 |   Π := make Π restricted (by unification of nodes);
4 |   Π := make Π locally R-closed (by adding edges);
5 |   Π := make Π flow-closed (by adding edges);
6 return Π;
```

The input of the type inference algorithm is a process P and a standard rule description $\mathcal{R}$. The algorithm starts by computing the initial shape predicate Π_P that directly corresponds to the syntax tree of P. Basically we forget name restrictions and replications and we translate parallel compositions into branching and sequential composition into sequencing in the graph. We shall prove later that the initial shape predicate Π_P is a minimal (w.r.t. $\preccurlyeq$) shape predicate such that $\vdash P : \Pi_P$.

The algorithm repeats the main cycle until an $\mathcal{R}$-type is found. The main cycle takes the currently computed shape predicate Π and makes it restricted by unifying nodes that need to be unified. Then it adds edges necessary to make the shape predicate Π locally $\mathcal{R}$-closed and flow-closed. Thus it is clear that the algorithm returns an $\mathcal{R}$-type iff it terminates.

In Chapter 12 we shall prove that the algorithm terminates for every standard $\mathcal{R}$. We know that P matches the initial shape predicate Π_P. Thus it is clear that the shape predicate computed by the algorithm is an $\mathcal{R}$-type of P because neither unification of nodes nor addition of edges can decrease the meaning of Π. We shall also prove that the resulting shape predicate is principal among restricted $\mathcal{R}$-types. This will be implied by the property that we do not unify nodes or add edges unless absolutely necessary.

The following sections provides a detailed description of the type inference algorithm. Proofs of termination and correctness are given separately in Chapter 12.

11.2 Initial Shape Predicate

We start by description of the algorithm which computes for every process P the initial shape predicate Π_P which directly corresponds to the syntax tree of P. The shape predicate Π_P is the smallest shape predicate w.r.t. $\preccurlyeq$ such that $\vdash P : \Pi_P$. In other words it holds that $\Pi_P \preccurlyeq \Pi$ whenever $\vdash P : \Pi$. This property is proved in Section 12.6.

In order to compute the smallest shape predicate which corresponds to the syntax tree of an input process we need to be able to compute the principal form type φ for every form F. This is done by algorithm `FormType` which uses the subroutines `SequenceTypeSet`, `MessageType`, and `ElementType`.

Algorithm 11.2: **Function** `SequenceTypeSet`(M)

input : a message M
output: the principal sequence type set of M

```
1 switch M do
2     case 0: return ∅;
3     case x_0 ... x_k:  return {x̄_0 ... x̄_k};
4     case M_0.M_1:
5         Σ_0 := SequenceTypeSet(M_0);
6         Σ_1 := SequenceTypeSet(M_1);
7         return Σ_0 ∪ Σ_1;
```

Algorithm `SequenceTypeSet`, which computes the principal sequence type set for every message M, proceeds simply by the structure of M. When M is some sequence $x_0 \dots x_k$ then the algorithms simply forgets the basic names in M and returns the

singleton sequence type set $\{\overline{x_0} \ldots \overline{x_k}\}$. The correctness of `SequenceTypeSet` is proved in Section 12.6.

Algorithm `MessageType` computes the principal message type μ for every message M. Its correctness is proved in Section 12.6. It calls `SequenceTypeSet` when M is not a single name.

Algorithm 11.3: **Function** `MessageType`(M)

input : a message M
output: the principal message type of M

```
1 switch M do
2 |  case x: return x̄;
3 |  otherwise  Σ := SequenceTypeSet(M); return Σ*;
```

Algorithm `ElementType` computes the principal element type ε for every element E. Its correctness is proved in Section 12.6. When E is an output-element type then `ElementType` calls `MessageType` for every message in E. For other E it simply forgets basic names in all names in E.

Algorithm 11.4: **Function** `ElementType`(E)

input : an element E
output: the principal element type of E

```
1 switch E do
2 |  case x: return x̄;
3 |  case (x_1, ..., x_k): return (x̄_1, ..., x̄_k);
4 |  case <M_1, ..., M_k>:
5 |  |  for i := 1 to k do  μ_i := MessageType(M_i);
6 |  |  return <μ_1 ... μ_k>;
```

Algorithm `FormType` computes the principal form type φ for every form F. Its correctness is proved in Section 12.6. The algorithm simply calls `ElementType` for every element in F.

Algorithm 11.5: **Function** `FormType`(F)

input : a form F
output: the principal form type of F

```
1 E_0 ... E_k := F;
2 for i := 0 to k do  ε_i := ElementType(E_i);
3 return ε_0 ... ε_k;
```

Algorithm `ProcessShape` computes for every process P the initial shape predicate Π_P which directly corresponds to the syntax tree of P.

Name restriction and replication are simply ignored. Sequential composition (".") is translated to edge sequencing as follows. The initial shape predicate of $F.P_0$

Algorithm 11.6: **Function** `ProcessShape`(P)

input : a process P
output: a shape predicate Π with $\vdash P : \Pi$ that directly corresponds to the syntax tree of P

```
1  switch P do
2      case 0:  return ⟨∅, R⟩;
3      case F.P_0:
4          ⟨Γ_0, χ_0⟩ := ProcessShape(P_0);
5          χ := a node fresh for Γ_0;
6          φ := FormType(F);
7          return ⟨{χ -φ-> χ_0} ∪ Γ_0, χ⟩;
8      case P_0 | P_1:
9          ⟨Γ_0, χ_0⟩ := ProcessShape(P_0);
10         ⟨Γ_1, χ_1⟩ := ProcessShape(P_1);
11         Γ'_1 := Γ_1 with all nodes except χ_1 replaced by nodes fresh for Γ_0;
12         Γ''_1 := Γ'_1 with all occurrences of χ_1 replaced by χ_0;
13         return ⟨Γ_0 ∪ Γ''_1, χ_0⟩;
14     case νx.P_0: return ProcessShape(P_0);
15     case !P_0: return ProcessShape(P_0);
```

is computed from Π_0 = `ProcessShape`(P_0) by creating a fresh root node χ and connecting χ to the original root of Π_0 by an edge labeled with φ = `FormType`(F). Parallel composition ("|") is translated to edge branching as follows. The initial shape predicate of P_0 | P_1 is computed from Π_0 = `ProcessShape`(P_0) and Π_1 = `ProcessShape`(P_1) by making the non-root nodes in Π_1 distinct from nodes in Π_0 and putting both graphs together. The correctness of `ProcessShape` is proved in Section 12.6.

11.3 Restriction Algorithm

Now we describe the algorithm `RestrictGraph` which unifies nodes in the input shape predicate so that it becomes restricted. The algorithm `RestrictGraph` uses two subprocedures `RestrictWidth` and `RestrictDepth` which unify edges in an input shape predicate to make it width- respectively depth-restricted. It can happen that a unification of nodes in `RestrictDepth` can violate the width restriction in a previously width-restricted graph. Thus the main algorithm `RestrictGraph` has to call the two subroutines consequently until the resulting graph is restricted.

Algorithm 11.7 `RestrictWidth` works as follows. It starts with the graph part of the input shape predicate Π and it repeats the main cycle until the graph contains two distinct edges $\chi \xrightarrow{\varphi_0} \chi_0$ and $\chi \xrightarrow{\varphi_1} \chi_1$ that violates the width restriction. If the above two edges are found then χ_0 and χ_1 are unified by renaming χ_0 to χ_1 or otherwise. Note that the algorithm preserves the root node, that is, the root

node is never renamed. The resulting shape predicate $\Pi' =$ `RestrictWidth(`Π`)` is the "smallest" shape predicate which is width-restricted and that $\Pi \preccurlyeq \Pi'$. This is to say that only those nodes that has to be unified are actually unified, and that the order in which the nodes are unified does not matter. The exact definition of what the "smallest" means and proofs of important properties of `RestrictWidth` can be found in Section 12.7.1. The main argument of the termination proof of `RestrictWidth` is that the number of nodes in Γ is decreased with every iteration of the **while** loop.

Algorithm 11.7: **Function** `RestrictWidth(`Π`)`

input : a shape predicate Π
output: a width-restricted Π' such that $\Pi \preccurlyeq \Pi'$

1 $\langle \Gamma, \chi_\mathsf{r} \rangle := \Pi$;
2 **while** $\exists \{\chi \xrightarrow{\varphi_0} \chi_0, \chi \xrightarrow{\varphi_1} \chi_1\} \subseteq \Gamma : \chi_0 \neq \chi_1 \;\&\; \varphi_0 \approx \varphi_1$ **do**
3 | **if** $\chi_1 = \chi_\mathsf{r}$ **then** $\chi' := \chi_1$; **else** $\chi' := \chi_0$; *// keep* χ_r
4 | $\Gamma := \Gamma$ with all occurences of χ_0 and χ_1 replaced by χ';
5 **return** $\langle \Gamma, \chi_\mathsf{r} \rangle$;

Algorithm 11.8 `RestrictDepth` works similarly as `RestrictWidth`. It unifies nodes until no edges violate the depth restriction. Again, it preserves the root node and the resulting shape predicate is the "smallest" possible. The termination follows again from the fact that the number of nodes is decreased with every iteration of the **while** loop. Main properties of `RestrictDepth` are proved Section 12.7.2.

Algorithm 11.8: **Function** `RestrictDepth(`Π`)`

input : a shape predicate Π
output: a depth-restricted Π' such that $\Pi \preccurlyeq \Pi'$

1 $\langle \Gamma, \chi_\mathsf{r} \rangle := \Pi$;
2 **while** $\exists \{\chi_0 \xrightarrow{\varphi_0} \chi_1 \xrightarrow{\varphi_1} \cdots \chi_k \xrightarrow{\varphi_k} \chi_{k+1}\} \subseteq \Gamma : \chi_1 \neq \chi_{k+1} \;\&\; \varphi_0 \approx \varphi_k$ **do**
3 | **if** $\chi_{k+1} = \chi_\mathsf{r}$ **then** $\chi' := \chi_{k+1}$; **else** $\chi' := \chi_1$; *// keep* χ_r
4 | $\Gamma := \Gamma$ with all occurences of χ_1 and χ_{k+1} replaced by χ';
5 **return** $\langle \Gamma, \chi_\mathsf{r} \rangle$;

Algorithm 11.9 `RestrictGraph` calls `RestrictWidth` and `RestrictDepth` consequently until the resulting shape graph is restricted. It holds that Π is width-restricted iff `RestrictWidth(`Π`)` $= \Pi$ and similarly for `RestrictDepth`. The algorithm inherits the main properties of its subroutines, that is, the resulting shape predicate is the "smallest" possible and the number of nodes in Π increases with every iteration of the **repeat** cycle (except the final iteration). Main properties are proved in Section 12.7.3.

Algorithm 11.9: **Function** `RestrictGraph(Π)`

input : a shape predicate Π
output: a restricted Π' such that $\Pi \preccurlyeq \Pi'$

```
1 repeat
2 |   Π0 := Π;                                  // save the initial value
3 |   Π := RestrictWidth(Π);
4 |   Π := RestrictDepth(Π);
5 until Π = Π0 ;
6 return Π;
```

11.4 Local Closure Algorithm

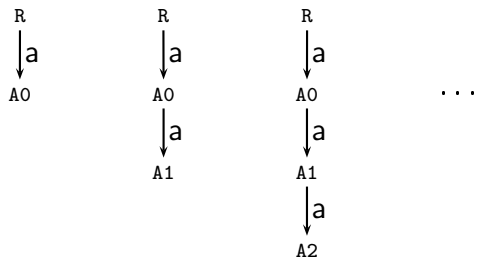
The local closure algorithm is the most complex part of the type inference algorithm. Basically we need, for a given $\mathcal{R}$ and Π, to apply every rewriting rule $\textbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ at every active node in Π and to add all edges required by the application of this rule.

The edges required by the application of $\textbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ at node χ are computed as follows. Firstly, the algorithm `LeftMatches` computes all possible $\Game$ such that $\Game \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. For each $\Game$, the algorithm `RightRequired` determines the minimal set Γ_0 of edges such that $\Game \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma \cup \Gamma_0, \chi \rangle$ holds. The edges from Γ_0 are added to Γ. The algorithm `LocalClosureStep` executes the above steps for every rewriting rule in $\mathcal{R}$ and every active node of Π.

The algorithm `LocalClosureStep` does not necessarily return a locally $\mathcal{R}$-closed graph. Basically, the above set Γ_0 contains the flow edges required by rules CFlow and CSub but it can also contain form edges required by CFrm. These form edges can create new opportunities to apply rewriting rules and thus the result of `LocalClosureStep` does not yet to be a locally $\mathcal{R}$-closed shape predicate. Moreover, just repeating `LocalClosureStep` would not give us a terminating algorithm as demonstrated by the following example. Consider the following $\mathcal{R}$.

$$\mathcal{R} = \{\textbf{active}\{\, \mathring{\mathtt{P}} \textbf{ in } \mathtt{a}.\mathring{\mathtt{P}} \,\}, \textbf{rewrite}\{\, \mathtt{a.0} \hookrightarrow \mathtt{a.a.0} \,\}\}$$

A consecutive application of `LocalClosureStep` to $\langle \{\mathtt{R} \xrightarrow{\mathtt{a}} \mathtt{A0}\}, \mathtt{R} \rangle$ would give us the following infinite sequence of non-locally $\mathcal{R}$-closed shape predicates.



We will interleave `LocalClosureStep` with the restriction algorithm `RestrictGraph` to ensure termination.

11.4.1 Matching Templates to Shape Graphs

We need an algorithm that for a given $\mathring{P}$ and $\langle \Gamma, \chi \rangle$ determines all possible $\mathbb{m}$ such that $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$. Here we are interested only in those $\mathbb{m}$ with $\mathrm{var}(\mathring{P}) = \mathrm{dom}(\mathbb{m})$ because values of $\mathbb{m}$ for variables not in $\mathrm{var}(\mathring{P})$ are irrelevant for $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$. It is to say that we are interested only in those type instantiations which are minimal w.r.t. set inclusion. For every Γ, χ, a $\mathring{P}$, there is only finitely many of minimal type instantiations $\mathbb{m}$ such that $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$. These instantiations are computed by Algorithm 11.12 `LeftMatches` which uses the following subroutines `MatchElement` and `MatchForm`.

Algorithm 11.10: **Function** `MatchElement`$(\mathbb{m}, \mathring{E}, \varepsilon)$

input : a type instantiation $\mathbb{m}$ to be extended to $\mathbb{m}'$ such that $\mathbb{m}'[\![\mathring{E}]\!] = \varepsilon$
output: $\{\mathbb{m}'\}$ such that $\mathbb{m} \subseteq \mathbb{m}'$ and $\mathbb{m}'[\![\mathring{E}]\!] = \varepsilon$, or $\emptyset$ iff there is no such $\mathbb{m}'$

```
1  switch Ė do
2      case x:  if x̄ = ε then return {m};
3      case ẋ:
4          if ε ∈ TypeTag & (ẋ ∈ dom(m) ⇒ m(ẋ) = ε) then
5              return {m[ẋ ↦ ε]};
6      case (ẋ_1, ..., ẋ_k):
7          if ∃ι_1, ..., ι_k : ε = (ι_1, ..., ι_k) then
8              if ∀j ∈ {1, ..., k} : ẋ_j ∈ dom(m) ⇒ m(ẋ_j) = ι_j then
9                  return {m[ẋ_1 ↦ ι_1, ..., ẋ_k ↦ ι_k]};
10     case <ṁ_1, ..., ṁ_k>:
11         if ∃μ_1, ..., μ_k : ε = <μ_1, ..., μ_k> then
12             return {m[ṁ_1 ↦ μ_1, ..., ṁ_k ↦ μ_k]};
13 return ∅;
```

For every $\mathring{E}$ and ε, Algorithm 11.10 `MatchElement` extends the input type instantiation $\mathbb{m}$ to the minimal (w.r.t. $\subseteq$) type instantiation $\mathbb{m}'$ such that $\mathbb{m}'[\![\mathring{E}]\!] = \varepsilon$. If there is such $\mathbb{m}'$ then it is unique and then the algorithm returns $\{\mathbb{m}'\}$. The algorithm returns $\emptyset$ if there is no $\mathbb{m}' \supseteq \mathbb{m}$ such that $\mathbb{m}'[\![\mathring{E}]\!] = \varepsilon$. The input argument $\mathbb{m}$ works as an accumulator which holds type instantiation computed so far during type inference. Note that the value of a message variable $\mathring{m}$ in $\mathbb{m}$ can be rewritten by `MatchElement` but it will not actually happen when $\mathring{E}$ comes from some well formed lhs-template $\mathring{P}$. It is because L3 says that there is at most one occurrence of $\mathring{m}$ in a well formed lhs-template $\mathring{P}$ and thus the accumulator will not contain $\mathring{m}$ when $\mathring{m} \in \mathrm{var}(\mathring{E})$. More details on the correctness and other properties of `MatchElement` are given in

Section 12.8.1.

Algorithm 11.11: **Function** `MatchForm`$(\mathbb{w}, \mathring{F}, \varphi)$

input : a type instantiation $\mathbb{w}$ to be extended to $\mathbb{w}'$ such that $\mathbb{w}'(\!|\mathring{F}|\!) = \varphi$
output: $\{\mathbb{w}'\}$ such that $\mathbb{w} \subseteq \mathbb{w}'$ and $\mathbb{w}'(\!|\mathring{F}|\!) = \varphi$, or $\emptyset$ iff there is no such $\mathbb{w}'$

```
1 Ė_0 ... Ė_k := F̊;
2 ε_0 ... ε_k' := φ;
3 if k ≠ k' then return ∅;
4 w_0 := w;
5 for i := 0 to k do
6 |   I := MatchElement(w_0, Ė_i, ε_i);
7 |   if ∃w_1 : I = {w_1} then w_0 := w_1; else return ∅;
8 return {w_0};
```

Algorithm 11.11 `MatchForm` is similar to `MatchElement` but it works with form templates and form types. That is, for every $\mathring{F}$ and φ, it extends the input $\mathbb{w}$ to the minimal (w.r.t. $\subseteq$) type instantiation $\mathbb{w}'$ such that $\mathbb{w}'(\!|\mathring{F}|\!) = \varphi$. If there is such $\mathbb{w}'$ then it is unique and then the algorithm returns $\{\mathbb{w}'\}$. The algorithm returns $\emptyset$ if there is no $\mathbb{w}' \supseteq \mathbb{w}$ such that $\mathbb{w}'(\!|\mathring{F}|\!) = \varphi$. More details on the correctness and other properties of `MatchForm` are given in Section 12.8.2.

Algorithm 11.12: **Function** `LeftMatches`$(\mathbb{w}, \mathring{P}, \Gamma, \chi)$

input : a type instantiation $\mathbb{w}$ to be extended to $\mathbb{w}'$ such that $\mathbb{w}' \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ where $\mathring{P}$ is a well formed lhs-template
output: the set of all minimal $\mathbb{w}' \supseteq \mathbb{w}$ such that $\mathbb{w}' \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$

```
1  I := ∅;
2  switch P̊ do
3  |   case 0: I := {w};
4  |   case p̊: I := {w[p̊ ↦ χ]};
5  |   case F̊.P̊_0:
6  |   |   foreach (χ -φ-> χ_0) ∈ Γ do
7  |   |   |   foreach w_0 ∈ MatchForm(w, F̊, φ) do
8  |   |   |   |   I := I ∪ LeftMatches(w_0, P̊_0, Γ, χ_0);
9  |   case P̊_0 | P̊_1:
10 |   |   foreach w_0 ∈ LeftMatches(w, P̊_0, Γ, χ) do
11 |   |   |   I := I ∪ LeftMatches(w_0, P̊_1, Γ, χ);
12 return I;
```

Finally, Algorithm 11.12 `LeftMatches` computes for every well formed lhs-template $\mathring{P}$ and $\langle \Gamma, \chi \rangle$ the set of all minimal extensions $\mathbb{w}'$ of $\mathbb{w}$ (that is, $\mathbb{w} \subseteq \mathbb{w}'$) such that $\mathbb{w}' \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. The argument $\mathbb{w}'$ serves again as an accumulator which contains the fixed values of variables computed so far by the previous run of the algorithm.

We suppose that $\mathring{P}$ is a well formed lhs-template and thus $\mathrm{dom}(\mathbb{\pi})$ will not contain $\mathring{p}$ when $\mathring{P} = \mathring{p}$, that is to say that no value will be replaced at line 4. In the case $\mathring{P} = \mathring{F}.\mathring{P}_0$ the algorithm iterates over all $\chi \xrightarrow{\varphi} \chi_0$ from Γ and it calls `FormType` to extend $\mathbb{\pi}$ to $\mathbb{\pi}_0$ such that $\mathbb{\pi}_0(\!|\mathring{F}|\!) = \varphi$. Then it calls recursively `LeftMatches` to compute all extensions of $\mathbb{\pi}_0$ that instantiates $\mathring{P}_0$ as required and it collect the results in the global variable $\mathbb{I}$. The case when $\mathring{P} = \mathring{P}_0 \mid \mathring{P}_1$ is even simpler. We extend $\mathbb{\pi}$ to $\mathbb{\pi}_0$ that matches $\mathring{P}_0$ at χ and then we further extend $\mathbb{\pi}_0$ to all minimal instantiations which match $\mathring{P}$ and collect the results in $\mathbb{I}$. More details on the correctness and other properties of `LeftMatches` are given in Section 12.8.3.

11.4.2 Edges Required by a Rewriting Rule

Consider a rewriting rule **rewrite**$\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\}$. When $\mathbb{\pi} \vDash_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ then we need to compute a minimal set $\Gamma_0 \supseteq \Gamma$ such that $\mathbb{\pi} \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma_0, \chi \rangle$ where Γ_0 reuses as much as edges from Γ as possible. Algorithm 11.13 `RightRequired` serves this purpose, that is, it computes the set of edges required by the application of the above rule to Γ at χ. The algorithm assumes $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{dom}(\mathbb{\pi})$ and that $\mathring{Q}$ is a well formed rhs-template w.r.t. some well formed lhs-template.

Algorithm 11.13: **Function** `RightRequired`$(\mathbb{\pi}, \mathring{Q}, \Gamma, \chi)$

input : a type instantiation $\mathbb{\pi}$ with $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{dom}(\mathbb{\pi})$ for a well formed rhs-template $\mathring{Q}$ (w.r.t. some $\mathring{P}$)
output: a minimal Γ_0 such that $\mathbb{\pi} \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma_0, \chi \rangle$

1 **switch** $\mathring{Q}$ **do**
2 **case** `0`*:* **return** $\emptyset$;
3 **case** $\mathring{p}$*:* **return** $\{\mathbb{\pi}(\mathring{p}) \xrightarrow{\emptyset} \chi\}$;
4 **case** $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$*:* **return** $\{\mathbb{\pi}(\mathring{p}) \xrightarrow{\{\ldots, \mathbb{\pi}(\mathring{x}_i) \mapsto \mathbb{\pi}(\mathring{s}_i), \ldots\}} \chi\}$;
5 **case** $\mathring{F}.\mathring{Q}_0$*:*
6 $\varphi := \mathbb{\pi}(\!|\mathring{F}|\!)$;
7 **if** $\exists \chi_0' \colon (\chi \xrightarrow{\varphi} \chi_0') \in \Gamma$ **then**
8 $\chi_0 := \chi_0'$;
9 **else**
10 $\chi_0 :=$ a node fresh for Γ;
11 $\eta := (\chi \xrightarrow{\varphi} \chi_0)$;
12 **return** $(\{\eta\} \cup$ `RightRequired`$(\mathbb{\pi}, \mathring{Q}_0, \Gamma \cup \{\eta\}, \chi_0))$;
13 **case** $\mathring{Q}_0 \mid \mathring{Q}_1$*:*
14 $\Gamma_0 :=$ `RightRequired`$(\mathbb{\pi}, \mathring{Q}_0, \Gamma, \chi)$;
15 **return** $\Gamma_0 \cup$ `RightRequired`$(\mathbb{\pi}, \mathring{Q}_1, \Gamma \cup \Gamma_0, \chi)$;

In the case of a process variable $\mathring{p}$ or a substitution application template $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}$, the algorithm simply returns the required flow edge. Note that the edges are correctly defined because we suppose that $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{dom}(\mathbb{\pi})$. When $\mathring{Q} =$

$\mathring{F}.\mathring{Q}_0$ then we at first compute $\varphi = \mathbb{m}(\!|\mathring{F}|\!)$. Rule CFRM requires an edge $\eta = \chi \xrightarrow{\varphi} \chi_0$ for some χ_0 to be present in the shape graph. When there is $\chi \xrightarrow{\varphi} \chi_0$ in Γ then the algorithm reuses this edge, otherwise a fresh node and a new edge are created. Note that η is added to the graph when the recursive call at line 12 is made. This ensures that any fresh node possibly created in the recursive call is distinct from χ_0.

The case when $\mathring{Q} = \mathring{Q}_0 \mid \mathring{Q}_1$ is a simple recursive call. Again, note that the shape graph in the second recursive call is extended with the result Γ_0 of the first recursive call. As above, it is to prevent node name clashes when creating new fresh nodes. More details on the correctness and other properties of `RightRequired` are given in Section 12.8.4.

11.4.3 Active Node Algorithm

We need to apply the rewriting rules to all active nodes of a given Π and thus we need an algorithm to compute the set $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. Algorithm 11.14 `ActiveNodes` serves this purpose.

Algorithm 11.14: **Function** `ActiveNodes`$(\Pi, \mathcal{R})$

input : a finite set of rewriting rules $\mathcal{R}$ and a shape predicate Π
output: the set $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ of active nodes of Π

```
1  ⟨Γ, χr⟩ := Π;
2  Ξ := ∅;
3  Ξnew := {χr};
4  while Ξnew ≠ ∅ do
5  |  χ0 := an arbitrary node from Ξnew;
6  |  foreach (active{ p̊ in P̊ } ∈ R) do
7  |  |  foreach m ∈ LeftMatches(∅, P̊, Γ, χ0) do
8  |  |  |  if m(p̊) ∉ Ξ then Ξnew := Ξnew ∪ {m(p̊)}
9  |  Ξnew := Ξnew\{χ0};
10 |  Ξ := Ξ ∪ {χ0};
11 return Ξ;
```

`ActiveNodes` performs a simple walk through the graph starting at the root node. Variable Ξ stores the active nodes whose active successors have already been visited. This set becomes the set $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ in the end. Variable Ξ_{new} stores the active nodes whose active successors are to be visited. Variable Ξ_{new} contains only the root node at the beginning and the algorithm ends when Ξ_{new} is empty. The algorithm uses `LeftMatches` for all **active** rules from $\mathcal{R}$ to compute active successors of a given node. The algorithm supposes that $\mathcal{R}$ is finite. More details on the correctness and other properties of `ActiveNodes` are given in Section 12.8.5.

11.4.4 Local Closure in Steps

Algorithm 11.15 `LocalClosureStep` puts the previous algorithms together. It takes a finite rewriting rule description $\mathcal{R}$ and a shape predicate $\langle \Gamma, \chi \rangle$ as input, and it returns $\langle \Gamma \cup \Gamma_0, \chi \rangle$ where Γ_0 are all the edges required by applications of rules $\mathcal{R}$ to all active nodes in $\langle \Gamma, \chi \rangle$. Algorithm `LocalClosureStep` computes only the immediately required edges, that is to say that the algorithm does not apply the rewriting rules to the newly generated edges Γ_0. Thus the result $\langle \Gamma \cup \Gamma_0, \chi \rangle$ does not need to be locally $\mathcal{R}$-closed.

Algorithm 11.15: **Function** `LocalClosureStep`$(\Pi, \mathcal{R})$

input : a finite set of rewriting rules $\mathcal{R}$ and a shape predicate Π
output: Π extended with edges required by one application of $\mathcal{R}$ to Π

```
1 ⟨Γ, χ_r⟩ := Π;
2 Γ_0 := ∅;
3 foreach χ ∈ ActiveNodes(Γ, χ_r, R, ∅) do
4     foreach (rewrite{ P̊ ↪ Q̊ } ∈ R) do
5         foreach 𝕞 ∈ LeftMatches(∅, P̊, Γ, χ) do
6             Γ_0 := Γ_0 ∪ RightRequired(𝕞, Q̊, Γ, χ);
7 return ⟨Γ ∪ Γ_0, χ_r⟩;
```

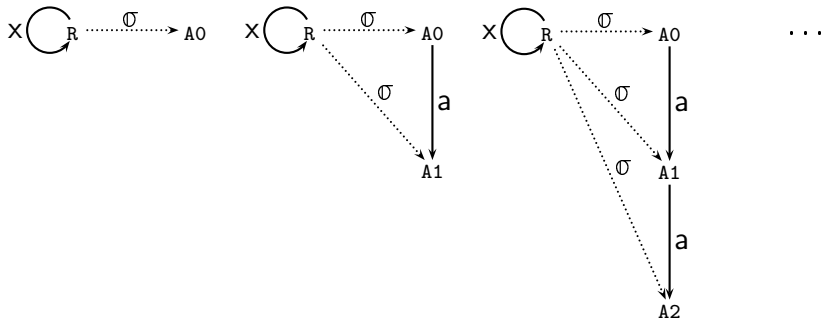
The algorithm simply iterates over all active nodes χ of $\Pi = \langle \Gamma, \chi_r \rangle$ and over all rewriting rules **rewrite**$\{ \mathring{P} \hookrightarrow \mathring{Q} \}$ from $\mathcal{R}$. Then it uses `LeftMatches` to compute all minimal (w.r.t. $\subseteq$) type instantiations $\mathbb{m}$ such that $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$. Then the algorithm collects edges computed by `RightRequired`. Note that the condition $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{dom}(\mathbb{m})$ required by `RightRequired` is implied by well-formedness conditions R2 and R3. More details on the correctness and other properties of `LocalClosureStep` are given in Section 12.8.6.

11.5 Flow Closure Algorithm

Algorithm 11.16 `FlowClosureStep` implements one step of a flow closure algorithm. For a given Π, it computes the edges Γ_0 immediately required by conditions F1 and F2 for (pairs of) edges from Π. It does not, however, compute the edges whose existence is consequently required by the newly added edges Γ_0. Thus the result of `FlowClosureStep` does not need to be a flow closed shape predicate, similarly as the result of `LocalClosureStep` does not need to be locally $\mathcal{R}$-closed.

Another similarity with `LocalClosureStep` is in that a mere repeating of flow closure steps `FlowClosureStep` would not give us a terminating algorithm to produce a flow closed shape predicate. To demonstrate this let us consider the follow-

ing sequence of shape graphs which do not satisfy the flow closure conditions. Let $\mathfrak{o} = \{\mathsf{x} \mapsto \mathsf{a}\}$.



In order to flow-close the first shape graph in the sequence we need to add the node `A1` and the edges $\texttt{A0} \xrightarrow{\mathsf{a}} \texttt{A1}$ and $\texttt{R} \overset{\mathfrak{o}}{\dashrightarrow} \texttt{A1}$. But the newly added $\texttt{R} \overset{\mathfrak{o}}{\dashrightarrow} \texttt{A1}$ together with $\texttt{R} \xrightarrow{\mathsf{x}} \texttt{R}$ insist on the existence of a new node `A2` with two new edges pointing to it. It is easy to see that we will never obtain a flow-closed graph in this way. The type inference will interleave flow closure steps with the restriction algorithm `RestrictGraph` in order to ensure termination.

Algorithm 11.16: **Function** `FlowClosureStep`(Π)

input : a shape predicate Π
output: Π extended with edges immediately required by flow closure conditions F1 and F2 for Π

1 $\langle \Gamma, \chi_\mathsf{r} \rangle = \Pi$;
2 $\Gamma_0 := \emptyset$;
3 **foreach** $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *and* $(\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$ **do**
4 **if** $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) \neq \emptyset$ **then continue**; *// skip this pair*
5 **if** $\varphi = \iota \in \mathrm{dom}(\mathfrak{o})$ & $\mathfrak{o}(\iota) = \Sigma*$ **then**
6 $\Gamma_0 := \Gamma_0 \cup \{\chi' \xrightarrow{\sigma} \chi' \colon \sigma \in \Sigma\} \cup \{\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi'\}$;
7 **else**
8 **if** $\exists \chi_0'' \colon (\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0'') \in \Gamma$ **then**
9 $\chi_0' := \chi_0''$;
10 **else**
11 χ_0' := a node fresh for $\Gamma \cup \Gamma_0$;
12 $\Gamma_0 := \Gamma_0 \cup \{\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0', \chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_0'\}$;
13 **return** $\langle \Gamma \cup \Gamma_0, \chi_\mathsf{r} \rangle$;

Algorithm `FlowClosureStep` iterates over all pairs of edges $\chi \xrightarrow{\varphi} \chi_0$ and $\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi'$ with the same source node χ such that $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$. For each edge pair, it determines whether the condition F1 or F2 applies and it collects the edges required by the appropriate condition in the variable Γ_0. The algorithm reuses form edges from the original graph when possible. More details on the correctness and other properties of `FlowClosureStep` are given in Section 12.9.

11.6 Type Inference Algorithm

Algorithm Algorithm 11.17 is used as a workaround to handle infinite set of rewriting rules. It takes, a possibly infinite, set of rules $\mathcal{R}$ and a process P and it returns the finite subset of $\mathcal{R}$ which contains all the rules from $\mathcal{R}$ that can ever apply to rewriting of P with $\mathcal{R}$. Of course, this is not always possible for an arbitrary $\mathcal{R}$ and P as discussed in Section 10.3 but it is possible when $\mathcal{R}$ is standard.

Algorithm 11.17: **Function** `SelectApplicableRules(`$\mathcal{R}$`, `P`)`

input : an arbitrary rewriting rule set $\mathcal{R}$ and a process P
output: a finite subset of rules in $\mathcal{R}$ that can ever be used when rewriting P; fails if $\mathcal{R}$ is not standard

```
1 if R is finite then return R;
2 if R is standard then
3 |  return {L̊ ∈ R : maxlen(L̊) ⩽ maxlen(P)};
4 fail "R is not standard";
```

The algorithm simply returns $\mathcal{R}$ when $\mathcal{R}$ is finite and it fails when $\mathcal{R}$ is not standard. When $\mathcal{R}$ is standard then the algorithm returns the set $\mathcal{R}^{\mathsf{fin}}$ of rules from $\mathcal{R}$ which do not contain any template entity longer then the longest META✶ entity in P. This set $\mathcal{R}^{\mathsf{fin}}$ is finite when $\mathcal{R}$ is standard (by Definition 10.3.3). The type inference algorithm `PrincipalType` would work correctly also for non-standard infinite rewriting rules $\mathcal{R}$ provided that an effective implementation of `SelectApplicableRules` specific for $\mathcal{R}$ is provided. If this specific implementation correctly computed a finite set $\mathcal{R}^{\mathsf{fin}}$ of all the rules that can ever be used when rewriting P with $\mathcal{R}$ then the type inference algorithm would correctly compute principal types.

Algorithm 11.18 `PrincipalType` implements type inference. For every standard $\mathcal{R}$ and a process P it computes a principal $\mathcal{R}$-type of P. It fails when $\mathcal{R}$ is not standard.

Algorithm 11.18: **Function** `PrincipalType(`P`, `$\mathcal{R}$`)`

input : a process P and a standard rewriting rule set $\mathcal{R}$
output: a principal $\mathcal{R}$-type of P

```
1 Π := ProcessShape(P);
2 R^fin := SelectApplicableRules(R, P);
3 repeat
4 |  Π0 := RestrictGraph(Π);
5 |  Π := LocalClosureStep(Π0, R^fin);
6 |  Π := FlowClosureStep(Π);
7 until Π = Π0 ;
8 return Π;
```

The algorithm at first calls `ProcessShape` to compute the initial shape predicate and it select a finite subset of applicable rules. Then it enters the main **repeat** loop

where the restriction algorithm `RestrictGraph` is called to make the initial shape predicate restricted. This restricted shape predicate is stored in the variable Π_0. Then one step of local $\mathcal{R}$-closure algorithm by `LocalClosureStep` and one step of the flow closure algorithm `FlowClosureStep` are executed. When the shape predicate is not changed during the execution of the two closure algorithms then Π_0 is both flow-closed and locally $\mathcal{R}$-closed (at all active nodes of Π_0). Thus Π_0 is already a restricted type and the algorithm returns Π_0 and terminates. Otherwise the **repeat** cycle is executed again until a restricted $\mathcal{R}$-type is found.

From the correctness of algorithms `RestrictGraph`, `LocalClosureStep`, and `FlowClosureStep` it is easy to conclude that `PrincipalType` returns a restricted $\mathcal{R}$-type of P when it terminates. In fact it returns an $\mathcal{R}^{\mathsf{fin}}$-type but it is easy to see that it is also an $\mathcal{R}$-type. The most difficult parts of the correctness proof are the following two properties. (1) `PrincipalType` terminates for every standard $\mathcal{R}$ and P. (2) The computed $\mathcal{R}$-type is principal among restricted $\mathcal{R}$-types.

In order to prove the termination (1) we count different edge paths in the shape predicate stored in variable Π. More specifically, we count different edge paths where only the last edge label in the path is allowed to repeat one of the previous labels. Then we prove that there is an upper bound on the count of these paths and that the number of these paths in Π is increased with every iteration of the **repeat** cycle. In order to prove (2), that the resulting type is principal, we at first observe that for the initial shape predicate Π_P it holds that $\Pi_P \preccurlyeq \Pi'$ whenever Π' is some restricted $\mathcal{R}$-type of P. Then we observe that this property is preserved by all the three algorithms executed inside the **repeat** loop. This is enough to prove that the resulting $\mathcal{R}$-type is principal among restricted $\mathcal{R}$-types of P. More details on the correctness and other properties of `PrincipalType` are given in Section 12.10.

Chapter 12

Technical Details on Type Inference

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

12.1 Overview of the Correctness Proof

For every algorithm from the previous chapter and mainly for `PrincipalType` we need to prove the following three properties.

Termination. The termination of an algorithm means that the algorithm terminates for all relevant inputs. Termination is discussed in Section 12.1.1.

Correctness. By the correctness of `PrincipalType` we mean the property that the algorithm does not fail for all relevant inputs and that the resulting value Π = `PrincipalType(`P, $\mathcal{R}$`)` is actually an $\mathcal{R}$-type of P. The correctness of any other algorithm from Chapter 11 is its property which is essential for the correctness of `PrincipalType`. Correctness is discussed in Section 12.1.2.

Completeness. By the completeness of `PrincipalType` we mean the property that the resulting type Π = `PrincipalType(`P, $\mathcal{R}$`)` is principal among restricted $\mathcal{R}$-types. The completeness of any other algorithm from Chapter 11 is its property which is essential for the completeness of `PrincipalType`. Completeness is discussed in Section 12.1.3.

12.1.1 Termination

In order to prove the termination of `PrincipalType` we need at first to prove that all the calls to functions `ProcessShape`, `SelectApplicableRules`, `RestrictGraph`, `LocalClosureStep`, and `FlowClosureStep` terminate. Secondly, we need to prove

that the **repeat** cycle is executed only finitely many times during the execution of `PrincipalType(`P, $\mathcal{R}$`)`, that is, that after finitely many steps the condition $\Pi = \Pi_0$ becomes satisfied. We are interested only in well formed P and $\mathcal{R}$. We additionally assume that $\mathcal{R}$ is standard because otherwise a terminating, correct and complete type inference algorithm does not need to exist.

Variable Π in `PrincipalType` contains the shape predicate computed so far. We shall find a numeric property related to Π which (1) is increased with (almost) every iteration of the **repeat** cycle and which (2) has an upper bound that can not be exceeded. This property is the count of *almost disjoint edge paths* in Π which is defined as follows.

DEFINITION 12.1.1. *Let* $\Pi = \langle \Gamma, \chi \rangle$. *An* **edge path in** Π *is a sequence of form types* $(\varphi_1, \ldots, \varphi_k)$ *such that there are nodes* $\chi_1, \ldots, \chi_k$ *and* $\{\chi \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi_k\}$ *is a rooted path in* Π. *The edge path is* disjoint *iff* $\varphi_i \neq \varphi_j$ *for all* $i, j \in \{1, \ldots, k\}$ *such that* $i \neq j$. *The edge path is* almost disjoint *iff* $\varphi_i \neq \varphi_j$ *for all* $i, j \in \{1, \ldots, k-1\}$ *such that* $i \neq j$.

Let paths(Π) *denote the count of different almost disjoint edge paths in* Π. ∎

There can be an infinite count of edge paths in a finite Π when Π contains loops. Thus we need to restrict ourselves to disjoint edge paths to keep the number paths(Π) finite. The count of disjoint paths is, however, not increased during some iterations of the **repeat** cycle and thus we count almost disjoint edge paths. The last form type on an almost disjoint edge path can repeat one of the preceding form types on the path. This is closely related to the depth-restriction.

The number paths(Π) never decreases during the execution of `PrincipalType`. The number paths(Π) increases with every iteration of the **repeat** cycle during which some form edge was added to Π. Some iterations, however, add only flow edges to Π and thus paths(Π) is not increased in these iterations. We shall prove that only finitely many flow edges can be added to a shape graph. Thus after finitely many iterations of the **repeat** cycle either the algorithm terminates or paths(Π) increases.

The remaining part of the termination argument is that there is an upper bound on paths(Π). Clearly only finitely many different form types of a fixed length can be constructed from a finite set of type tags. Moreover there is only finitely many tags in the input P and only finitely many rules from $\mathcal{R}$ which can be used compute the type of Π (because $\mathcal{R}$ is standard). Thus the number of different type tags in Π at any time of the execution of `PrincipalType(`P,$\mathcal{R}$`)` is limited (by the number of different tags in P and $\mathcal{R}$ plus one for "•"). Moreover, $\mathcal{R}$ is monotonic and thus maxlen(Π) stays constant during the execution of the algorithm. Thus there is only a finitely many form types which can appear in Π during the execution of `PrincipalType`. Only finitely many almost distinct edge paths can be constructed from finitely many form types. This gives as an upper bound on paths(Π) which is

more precisely evaluated in Section 12.3.

12.1.2 Correctness

In order to use the correctness of `PrincipalType` we at first prove that P matches the initial shape predicate computed by `ProcessShape`. Thus we obtain $\vdash P : \Pi$ holds after the execution of the first line. Then we observe that $\vdash P : \Pi$ is an invariant which is valid all the time during the execution of `PrincipalType`. This is because `RestrictGraph` can only unify nodes and thus can not reduce the meaning of Π. The following definition will become useful to prove this observation.

Definition 12.1.2. *A* **node map** δ *is a finite function from nodes to nodes. A node map* δ *is a* **node renaming of** Π *when* δ *is defined for all the nodes of* Π. *Application* $\delta(\Gamma)$ *of* δ *to the shape graph* Γ *is defined as follows.*

$$\delta(\Gamma) = \{\delta(\chi_0) \xrightarrow{\varphi} \delta(\chi_1) \colon (\chi_0 \xrightarrow{\varphi} \chi_1) \in \Gamma\}$$

For the shape predicate $\Pi = \langle \Gamma, \chi \rangle$ *we set* $\delta(\Pi) = \langle \delta(\Gamma), \delta(\chi) \rangle$. ∎

We shall prove that when Π_0 = `RestrictGraph(`Π`)` then there is some node renaming δ of Π such that $\delta(\Pi) = \Pi_0$. Moreover we shall prove that application of a node renaming to a shape predicate does not reduce its meaning. Furthermore we know that `LocalClosureStep` and `FlowClosureStep` only add edges to Π and thus they do not reduce its meaning either. Thus $\vdash P : \Pi$ clearly holds even for the result Π of `PrincipalType(`P`,`$\mathcal{R}$`)`.

In order to prove that the result Π = `PrincipalType(`P`,`$\mathcal{R}$`)` is actually an $\mathcal{R}$-type we shall prove a related correctness properties of algorithms `RestrictGraph`, `LocalClosureStep`, and `FlowClosureStep`. The correctness of `RestrictGraph` states that the resulting value is a restricted shape predicate. The correctness of `LocalClosureStep` says that when the return value is equal to the first argument then this argument is a locally $\mathcal{R}$-closed shape predicate (at any node active w.r.t. $\mathcal{R}$). That is, the correctness says that

`LocalClosureStep(`Π`,`$\mathcal{R}$`)` $= \Pi$ implies Π is locally $\mathcal{R}$-closed.

Similarly, the correctness of `FlowClosureStep` states that

`FlowClosureStep(`Π`)` $= \Pi$ implies Π is flow closed.

It is not hard to observe that both the algorithms actually returned the shape predicate unchanged in the last iteration of the **repeat** cycle in `PrincipalType`. Thus the result Π is a restricted $\mathcal{R}$-type of the input process P. Technically, using

the above argumentation we obtain only that the result an $\mathcal{R}^{\mathsf{fin}}$-type of P where $\mathcal{R}^{\mathsf{fin}}$ is the finite subset of $\mathcal{R}$ returned by `SelectApplicableRules`. However, we use the results proved in Section 12.2 to extend the validity of the claim to the original rule description $\mathcal{R}$ as long as $\mathcal{R}$ is standard.

12.1.3 Completeness

In order to prove the completeness of `PrincipalType`, that is, that its return value is a restricted principal type, we define the following notion of *nesting* of shape predicates.

Definition 12.1.3. *A node map δ is a* **nesting of** $\langle \Gamma, \chi_{\mathsf{r}} \rangle$ **in** $\langle \Gamma', \chi'_{\mathsf{r}} \rangle$*, which we write as* $\delta \mathrel{\therefore} \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$*, iff*

(1) $\delta(\chi_{\mathsf{r}}) = \chi'_{\mathsf{r}}$,
(2) for all $(\chi_0 \xrightarrow{\varphi} \chi_1) \in \Gamma$ *there is* $(\delta(\chi_0) \xrightarrow{\varphi'} \delta(\chi_1)) \in \Gamma'$ *with* $\varphi \preccurlyeq \varphi'$*, and*
(3) for all $(\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_1) \in \Gamma$ *there is* $(\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi_1)) \in \Gamma'$ *with* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$. ∎

It is easy to observe that the existence of a nesting of Π in Π' implies that $\Pi \preccurlyeq \Pi'$. The opposite implication does not necessarily hold. The nesting relation $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$ can be seen as an effective version of the subtyping relation and δ can be seen as the proof that $\Pi \preccurlyeq \Pi'$ actually holds.

Next we define an $\mathcal{R}$*-preprincipal* shape predicate for P to be a shape predicate which can be nested in any restricted $\mathcal{R}$-type of P. Clearly when a shape predicate Π is $\mathcal{R}$-preprincipal for P and also a restricted $\mathcal{R}$-type then Π is a restricted principal $\mathcal{R}$-type of P.

Definition 12.1.4. *A shape predicate* Π *is* $\mathcal{R}$**-preprincipal for** P *iff*

(1) $\vdash P : \Pi$ *and*
(2) for any Π' *such that* $\mathcal{R} \Vdash_{\mathsf{restr}} \Pi'$ *and* $\vdash P : \Pi'$ *there is* δ *such* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$. ∎

To prove the correctness of `PrincipalType(`P`,`$\mathcal{R}$`)` we at first prove that the algorithm `ProcessShape` returns an $\mathcal{R}$-preprincipal shape predicate for the input process P. In order to prove this we shall prove correctness of all the algorithms called from `ProcessShape`, that is, `SequenceTypeSet` and so on. Their correctness states simply that they return the principal type of their argument. The principal sequence type set of a message, the principal message type of a message, the principal element type of an element, and the principal form type of a form are defined as follows. In these four cases, the principal type of any Meta✶ basic entity is unique.

Definition 12.1.5. *A type entity* ζ *is a* principal type *of* Z *iff* $\vdash Z : \zeta$ *and for any* ζ' *such that* $\vdash Z : \zeta'$ *it holds that* $\zeta \preccurlyeq \zeta'$. ∎

Once we know that `ProcessShape` returns a preprincipal shape predicate we prove that the existence of a nesting of Π in any restricted $\mathcal{R}$-type of P is preserved during the execution of `PrincipalType`. This gives the completeness properties of the three algorithms called from the **repeat** cycle. For example, let Π_0 = `FlowClosureStep(`Π`)`. The completeness of `FlowClosureStep` says when there is a nesting $\delta \therefore \Pi \trianglelefteq \Pi'$ of Π in some restricted $\mathcal{R}$-type Π' of P then there is some nesting $\delta_0 \therefore \Pi_0 \trianglelefteq \Pi'$ of the result Π_0 in the very same Π'.

This gives us that the result Π = `PrincipalType(`P`,`$\mathcal{R}$`)` is $\mathcal{R}$-preprincipal for P. Technically, we again obtain that Π is $\mathcal{R}^{\mathsf{fin}}$-preprincipal but it clearly implies that Π is $\mathcal{R}$-preprincipal for P because every $\mathcal{R}$-type is automatically an $\mathcal{R}^{\mathsf{fin}}$-type. The correctness of `PrincipalType` states that Π is a restricted $\mathcal{R}$-type. The completeness property, that Π is a principal restricted $\mathcal{R}$-type, then follows directly from the definition of preprincipal shape predicates.

12.2 Infinite Rewriting Rules

The type inference algorithm can handle infinite rule descriptions provided they are standard as follows. When computing `PrincipalType(`P`,`$\mathcal{R}$`)` with some infinite but standard $\mathcal{R}$ we use `SelectApplicableRules` to compute a finite subset $\mathcal{R}^{\mathsf{fin}}$ of $\mathcal{R}$ that can ever be used during the type inference. When $\mathcal{R}$ is standard then the following $\mathcal{R}^{\mathsf{fin}} = \{\mathring{L} \in \mathcal{R} \colon \mathrm{maxlen}(\mathring{L}) \leqslant \mathit{len}\}$ is finite. The type inference algorithm then works solely with $\mathcal{R}^{\mathsf{fin}}$ and thus also the correctness and completeness results will be relative to $\mathcal{R}^{\mathsf{fin}}$. That is to say that we shall prove that resulting shape predicate is a restricted principal $\mathcal{R}^{\mathsf{fin}}$-type of P. The last step is to prove that the result is an $\mathcal{R}$-type as well and this section provides some definitions and techniques to do that.

Firstly we define lengths of POLY✶ type entities similarly to the lengths of META✶ entities.

DEFINITION 12.2.1. *The length of a* POLY✶ *entity is defined as follows.*

(1) a sequence type "$\iota_0 \ldots \iota_k$*" has the length* $k+1$

(2) an input element type "$(\iota_1, \ldots, \iota_k)$*" has the length* k

(3) an output element type "$<\mu_1, \ldots, \mu_k>$*" has the length* k

(4) a form type "$\varepsilon_0 \ldots \varepsilon_k$*" has the length* $k+1$

(5) any other POLY✶ *entity has the length* 0

Let $\mathrm{maxlen}(\varphi)$ *be the maximum of the lengths of all* POLY✶ *entities in* φ *(including* φ *itself). Let* $\mathrm{maxlen}(\Pi)$ *be the maximum of the lengths of all* POLY✶ *entities in* Π. ∎

The following lemma is used to prove Proposition 12.2.3 and it says that selection of a subset of rules does not affect the set of active nodes. Note that the lemma holds also for a non-standard and non-monotonic $\mathcal{R}$. An important condition is that $\mathrm{maxlen}(\Pi) \leqslant \mathit{len}$.

LEMMA 12.2.2. *Let $\mathcal{R}$, Π, and a natural number len be given. Let $\mathcal{R}_0 = \{\mathring{L} \in \mathcal{R} : \mathrm{maxlen}(\mathring{L}) \leqslant \mathit{len}\}$ and $\mathrm{maxlen}(\Pi) \leqslant \mathit{len}$. Then the following holds.*

$$\mathrm{ActiveNode}_{\mathcal{R}}(\Pi) = \mathrm{ActiveNode}_{\mathcal{R}_0}(\Pi)$$

PROOF. *Let $\Pi = \langle \Gamma, \chi_r \rangle$ and len and $\mathcal{R}_0$ be as above. It is enough to prove that for any χ it holds that $\mathrm{ActiveSucc}_{\mathcal{R}}(\chi, \Gamma) = \mathrm{ActiveSucc}_{\mathcal{R}_0}(\chi, \Gamma)$. At first let us prove the "$\subseteq$" inclusion. Let $\chi_0 \in \mathrm{ActiveSucc}_{\mathcal{R}}(\chi, \Gamma)$. Thus there are $\mathbb{m}$ and $\mathbf{active}\{\, \mathring{p} \,\mathbf{in}\, \mathring{P} \,\} \in \mathcal{R}$ such that $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ and $\mathbb{m}(\!|\mathring{p}|\!) = \chi_0$. Now $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ implies that $\mathrm{maxlen}(\mathring{P}) \leqslant \mathrm{maxlen}(\langle \Gamma, \chi \rangle) = \mathrm{maxlen}(\Pi)$ because otherwise $\mathring{P}$ would not be able to match at $\langle \Gamma, \chi \rangle$. Hence $\mathrm{maxlen}(\mathring{P}) \leqslant \mathit{len}$ and thus $\mathbf{active}\{\, \mathring{p} \,\mathbf{in}\, \mathring{P} \,\} \in \mathcal{R}_0$. Thus $\chi_0 \in \mathrm{ActiveSucc}_{\mathcal{R}_0}(\chi, \Gamma)$ because we have already proved above that $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ and $\mathbb{m}(\!|\mathring{p}|\!) = \chi_0$. The opposite inclusion "$\supseteq$" is trivial because $\mathcal{R}_0 \subseteq \mathcal{R}$.* ∎

The following proposition allows us to generalize the correctness of the type inference algorithm, that is, to prove that the result is an $\mathcal{R}$-type once we prove that it is an $\mathcal{R}^{\mathsf{fin}}$-type. Note that the proposition holds also for some non-standard rule sets. However even for standard $\mathcal{R}$, it does not hold for an arbitrary shape predicate Π but only when $\mathrm{maxlen}(\Pi) \leqslant \mathit{len}$. The problem is that an arbitrary Π can contain some extra edges on which some rewriting rule that is in $\mathcal{R}$ but not in $\mathcal{R}^{\mathsf{fin}}$ can apply. Nevertheless for a monotonic $\mathcal{R}$ and a principal $\mathcal{R}$-type of P it always hold that $\mathrm{maxlen}(\Pi) \leqslant \mathrm{maxlen}(P)$ and thus the type inference algorithm can never introduces the extra edges as above.

PROPOSITION 12.2.3. *Let $\mathcal{R}$ be monotonic and let a natural number len be given. Let $\mathcal{R}_0 = \{\mathring{L} \in \mathcal{R} : \mathrm{maxlen}(\mathring{L}) \leqslant \mathit{len}\}$ and $\mathrm{maxlen}(\Pi) \leqslant \mathit{len}$. Then*

$$\mathcal{R}_0 \Vdash_{\mathsf{type}} \Pi \quad \textit{implies} \quad \mathcal{R} \Vdash_{\mathsf{type}} \Pi.$$

PROOF. *Let $\mathcal{R}$, len, $\mathcal{R}_0$, and Π be as above. Let $\mathcal{R}_0 \Vdash_{\mathsf{type}} \Pi$. Let $\Pi = \langle \Gamma, \chi_r \rangle$. It is enough to prove that Γ is locally $\mathcal{R}$-closed at all active nodes $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. Let $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. By Lemma 12.2.2 we know that $\chi \in \mathrm{ActiveNode}_{\mathcal{R}_0}(\Pi)$ and thus $\mathcal{R}_0 \Vdash_{\mathsf{type}} \Pi$ implies that Γ is locally $\mathcal{R}_0$-closed at χ. Now let us prove that Γ is locally $\mathcal{R}$-closed at χ.*

Let $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ and let $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ for some $\mathbb{m}$. Now $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ implies that $\mathrm{maxlen}(\mathring{P}) \leqslant \mathrm{maxlen}(\langle \Gamma, \chi \rangle) = \mathrm{maxlen}(\Pi)$ because otherwise $\mathring{P}$ would not be able to match at $\langle \Gamma, \chi \rangle$. Hence $\mathrm{maxlen}(\mathring{P}) \leqslant \mathit{len}$ and because $\mathcal{R}$ is

monotonic we obtain that $\mathrm{maxlen}(\mathring{Q}) \leqslant len$ *as well. But it means that* $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}_0$ *and thus* $\Vdash_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$ *because* Γ *is locally* $\mathcal{R}_0$*-closed at* χ*. Hence* Γ *is locally* $\mathcal{R}$*-closed at* χ*.* ∎

12.3 Upper Bound on Almost Disjoint Paths

In this section we enumerate the upper bound on almost distinct edge paths in a shape predicate. We use this upper bound to construct an invariant valid during an execution of `PrincipalType` which will become part of the argument for the termination of the type inference algorithm.

Let two natural numbers *tags* and *len* be given. We shall count the number of different almost disjoint edge paths that can be constructed from *tags* type tags provided that no type entity has length more than *len*. Only finitely many sequence types σ with $\mathrm{maxlen}(\sigma) \leqslant len$ can be constructed from a finite set of type tags with *tags* elements. Firstly, there is only

$$seqs = \sum_{k=1}^{len} tags^k = \frac{tags^{len+1} - tags}{tags - 1}$$

of different form types which are not longer than *len* that can be constructed from *tags* type tags. From these form types only 2^{seqs} of different form type sets can be made. These give us

$$msgs = tags + 2^{seqs}$$

of different message types. Furthermore, *msgs* message types gives rise to

$$elems = tags + \sum_{k=0}^{len} k! + \sum_{k=0}^{len} msgs^k = tags + \sum_{k=0}^{len} k! + \frac{msgs^{len+1} - 1}{msgs - 1}$$

of different element types. Thus altogether there is only

$$forms = \sum_{k=1}^{len} elems^k = \frac{elems^{len+1} - elems}{elems - 1}$$

of different form types with no type entity longer than *len* constructible from *tags* of different type tags. Finally there is only

$$maxpaths = \sum_{k=0}^{forms} (k+1) \frac{forms!}{(forms - k)!}$$

of different almost disjoint edge paths constructible from *forms* form types. The fraction inside the sum denotes the number of different k-length sequences of different form types constructed from *forms* form types. It is multiplied by $(k+1)$ which

embodies the number of k possibilities to choose the last form type that repeats one of the preceding ones, plus the possibility that no form type repeats.

Let P and a finite $\mathcal{R}$ be given. It is clear that P and $\mathcal{R}$ contain only finitely many type tags. It is easy to check that the initial shape predicate $\Pi_0 = \texttt{ProcessShape}(P)$ is constructed only from the type tags in P. Moreover we can see that the type inference algorithm can not introduce a new type tag that is not in $\mathcal{R}$ (except "•") during the computation of $\texttt{PrincipalType}(P,\mathcal{R})$. Let *tags* denote the number of different type tags contained in P and $\mathcal{R}$ plus one (for "•"). Thus the shape predicate in variable Π does not contain more than *tags* different type tags at any time during the execution of `PrincipalType`.

Let $\mathcal{R}$ be additionally monotonic and let $len = \text{maxlen}(P)$. It is not hard to see that for the initial shape predicate Π_0 we have $\text{maxlen}(\Pi_0) \leqslant len$ (maxlen for shape types is defined Definition 12.2.1). Moreover it is easy to observe that no application of a monotonic rule can introduce a type entity longer than *len* to the shape predicate in variable Π during the execution of `PrincipalType`. Thus the shape predicate in variable Π never contain more than *maxpaths* of different almost distinct edge paths during the execution of the type inference algorithm. That is to say that $\text{paths}(\Pi) \leqslant maxpaths$ is an invariant valid at any time during the execution of $\texttt{PrincipalType}(P,\mathcal{R})$. Note that we have required $\mathcal{R}$ to be finite and monotonic.

12.4 Note on Time Complexity of Type Inference

We can see that runtime of the time inference algorithm is closely related to the number of edges in the resulting graph and thus it is reasonable to measure the runtime in the number of edges that were added to a shape graph. The above upper bound on the number of almost disjoint edge paths gives us over-approximation of time complexity which would look similarly as the following (*len* and *tags* are clearly smaller than the length n of inputs P and $\mathcal{R}$).

$$2^{(2^{(n^n)})}$$

Although this time complexity is not very optimistic it has to be noted that the actual complexity heavily depends on rule description $\mathcal{R}$. It is not hard to artificially construct $\mathcal{R}$ which will result in large principal typings which are very near to the above formula in size. Thus the height of the above approximation is not caused by ineffectiveness of our implementation but rather by the complexity of the problem. Finally, the time complexity of type inference for rule descriptions of process calculi from the literature which are of interest, like $\mathcal{P}_{\mathsf{sync}}$ or $\mathcal{A}_{\mathsf{mon}}$ from Section 5.3, is much lower. We believe that in the case of the π-calculus $\mathcal{P}_{\mathsf{sync}}$ it is polynomial although we have not formally proved it yet. In the case of calculi which communi-

cate non-single name messages, like Mobile Ambients $\mathcal{A}_{\mathsf{mon}}$, examples of processes with principal typings which are exponential in the size of the process are known[1]. These examples, however, are usually not meaningful Mobile Ambients processes. A proper investigation of the time complexity of type inference is left for the future research.

12.5 Properties of Renamings and Nestings

The following lemma says that application of a node renaming to a shape predicate Π can not reduce its meaning. It can, however, extend it when two different nodes are mapped to the same node.

LEMMA 12.5.1. *Let δ be a node renaming of Π. Then $\Pi \preccurlyeq \delta(\Pi)$.*

PROOF. *Let P be given. We prove by induction of the structure P that for any Π and any node renaming δ of Π, $\vdash P : \Pi$ implies $\vdash \delta(P) : \Pi$. Let Π and δ as the above be given and let $\vdash P : \Pi$. The only non-trivial case is when $P = F.P_0$. Let $\Pi = \langle \Gamma, \chi \rangle$. Then there are some φ and χ_0 such that $\vdash F : \varphi$ and $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ and $\vdash P_0 : \langle \Gamma, \chi_0 \rangle$. By the induction hypothesis we have that $\vdash P_0 : \delta(\langle \Gamma, \chi_0 \rangle)$. Using Definition 12.1.2 we obtain $\vdash P_0 : \langle \delta(\Gamma), \delta(\chi_0) \rangle$ and $(\delta(\chi) \xrightarrow{\varphi} \delta(\chi_0)) \in \delta(\Gamma)$. Thus $\vdash P : \langle \delta(\Gamma), \delta(\chi) \rangle$ which proves the claim.* ∎

The next lemma says that nesting of shape predicates implies subtyping. The opposite implication does not necessarily hold.

LEMMA 12.5.2. *When $\delta \therefore \Pi \trianglelefteq \Pi'$ then $\Pi \preccurlyeq \Pi'$.*

PROOF. *Let $\delta \therefore \Pi \trianglelefteq \Pi'$. Let $\vdash P : \Pi$. Let $\Pi = \langle \Gamma, \chi_{\mathsf{r}} \rangle$ and $\Pi = \langle \Gamma', \chi'_{\mathsf{r}} \rangle$. Let us prove by induction on the structure of P the property that for any χ it holds that $\vdash P : \langle \Gamma, \chi \rangle$ implies $\vdash P : \langle \Gamma', \delta(\chi) \rangle$. The only non-trivial case is when $P = F.P_0$. Then $\vdash P : \langle \Gamma, \chi \rangle$ implies that there are φ and χ_0 such that $\vdash F : \varphi$ and $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ and $\vdash P_0 : \langle \Gamma, \chi_0 \rangle$. The induction hypothesis gives us that $\vdash P_0 : \langle \Gamma', \delta(\chi_0) \rangle$. Moreover $\delta \therefore \Pi \trianglelefteq \Pi'$ gives us that there is $(\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)) \in \Gamma'$ with $\varphi \preccurlyeq \varphi'$. Clearly $\vdash F : \varphi$ and $\varphi \preccurlyeq \varphi'$ implies $\vdash F : \varphi'$. Thus $\vdash F.P_0 : \langle \Gamma', \delta(\chi) \rangle$ which was to be proved. Hence the claim of the lemma follows from the above property because $\delta(\chi_{\mathsf{r}}) = \chi'_{\mathsf{r}}$.* ∎

Let $\delta \therefore \Pi \trianglelefteq \Pi'$. The following lemma says that when a flow closure condition F1 or F2 applies for two edges from Π then the same condition applies for the corresponding edges of Π'. To demonstrate this let us suppose that $\chi \xrightarrow{\varphi} \chi_0$ and $\chi \overset{\varpi}{\dashrightarrow} \chi'$ are in Π. Now $\delta \therefore \Pi \trianglelefteq \Pi'$ implies that there are some $\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)$ $\delta(\chi) \overset{\varpi'}{\dashrightarrow} \delta(\chi')$ in Π' with $\varphi \preccurlyeq \varphi'$ and $\varpi \preccurlyeq \varpi'$. Now Lemma 8.1.6 implies that

[1] For example "(x).(y).(<x y>.0 | <x.y>.0) | <a>.0 | <b>.0".

$\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \mathrm{itags}(\varphi') \cap \mathrm{dom}(\mathfrak{o}')$. Finally Lemma 12.5.3 says that condition F1 is satisfied for the above two edges from Π iff condition F1 is satisfied for the above corresponding edges from Π'.

LEMMA 12.5.3. *Let $\varphi \preccurlyeq \varphi'$ and $\mathfrak{o} \preccurlyeq \mathfrak{o}'$. It holds that*

$$\varphi = \iota \in \mathrm{dom}(\mathfrak{o}) \;\&\; \mathfrak{o}(\iota) \notin \mathrm{TypeTag} \quad \text{iff} \quad \varphi' = \iota \in \mathrm{dom}(\mathfrak{o}') \;\&\; \mathfrak{o}'(\iota) \notin \mathrm{TypeTag}$$

PROOF. *Let $\varphi \preccurlyeq \varphi'$ and $\mathfrak{o} \preccurlyeq \mathfrak{o}'$. Firstly let us prove the "$\Rightarrow$" implication. Let $\varphi = \iota \in \mathrm{dom}(\mathfrak{o})$ and $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$. We see that $\iota = \varphi \in \mathrm{TypeTag}$ and thus $\varphi' = \iota$ by Lemma 8.1.5. Clearly $\iota \in \mathrm{dom}(\mathfrak{o}')$. From $\mathfrak{o}(\iota) \preccurlyeq \mathfrak{o}'(\iota)$ and $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$ we obtain $\mathfrak{o}'(\iota) \notin \mathrm{TypeTag}$ by Lemma 8.1.4. Hence the claim.*

Now let us prove the "$\Leftarrow$" implication. Let $\varphi' = \iota \in \mathrm{dom}(\mathfrak{o}')$ and $\mathfrak{o}'(\iota) \notin \mathrm{TypeTag}$. We see that $\iota = \varphi' \in \mathrm{TypeTag}$ and thus $\varphi = \iota$ by Lemma 8.1.5. Clearly $\iota \in \mathrm{dom}(\mathfrak{o})$. From $\mathfrak{o}(\iota) \preccurlyeq \mathfrak{o}'(\iota)$ and $\mathfrak{o}'(\iota) \notin \mathrm{TypeTag}$ we obtain $\mathfrak{o}(\iota) \notin \mathrm{TypeTag}$ by Lemma 8.1.4. Hence the claim. ∎

The following definition introduces nesting of type instantiations $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$. The main relation between nesting of shape predicates and nesting of type instantiations is as follows. When $\delta \therefore \Pi \trianglelefteq \Pi'$ and $\mathbb{m} \vDash_{\mathrm{L}} \mathring{P} : \Pi$ then there is some $\mathbb{m}'$ such that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$ and $\mathbb{m} \vDash_{\mathrm{L}} \mathring{P} : \Pi'$. This is is formally expressed by Lemma 12.5.8. Note that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$ does not depend on δ when $\mathrm{dom}(\mathbb{m})$ does not contain any process variables.

DEFINITION 12.5.4. *Write $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$ iff*

(1) $\mathrm{dom}(\mathbb{m}) = \mathrm{dom}(\mathbb{m}')$,

(2) $\mathbb{m}(\mathring{x}) = \mathbb{m}'(\mathring{x})$ *for all* $\mathring{x} \in \mathrm{dom}(\mathbb{m}) \cap \mathrm{NameVar}$.

(3) $\mathbb{m}(\mathring{m}) \preccurlyeq \mathbb{m}'(\mathring{m})$ *for all* $\mathring{m} \in \mathrm{dom}(\mathbb{m}) \cap \mathrm{MessageVar}$, *and*

(4) $\mathbb{m}(\mathring{p}) \in \mathrm{dom}(\delta)$ *and* $\delta(\mathbb{m}(\mathring{p})) = \mathbb{m}'(\mathring{p})$ *for all* $\mathring{p} \in \mathrm{dom}(\mathbb{m}) \cap \mathrm{ProcessVar}$. ∎

The following states that two nested type instantiations instantiates the same element (respectively form) template to two element (respectively form) types correspondingly related by the subtyping relation.

LEMMA 12.5.5. *Let $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$. Then $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \mathbb{m}'(\!|\mathring{E}|\!)$ and $\mathbb{m}(\!|\mathring{F}|\!) \preccurlyeq \mathbb{m}'(\!|\mathring{F}|\!)$.*

PROOF. *Let $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$. The first claim $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \mathbb{m}'(\!|\mathring{E}|\!)$ is easily proved by induction on the structure of $\mathring{E}$. The second claim follows directly from the first one.* ∎

The following two lemmas are used to prove the above relation between the nesting of shape predicates and the nesting of type instantiations. The first lemma describes a relation between a subtyping of element types and a nesting of type

instantiations as follows. Let $\mathrm{var}(\mathring{E}) = \mathrm{dom}(\mathbb{m})$. When $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \varepsilon$ then we can find type instantiation $\mathbb{m}'$ such that $\emptyset \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$ and such that $\mathbb{m}'$ instantiates $\mathring{E}$ to ε (that is, $\mathbb{m}'(\!|\mathring{E}|\!) = \varepsilon$). Note that $\mathring{E}$ can not contain any process variables and thus we simply use the empty node map $\emptyset$. The lemma also assumes that $\mathbb{m}$ can be defined for some variables not mentioned in $\mathring{E}$ and thus we restrict $\mathbb{m}$ in the lemma only to the variables $\mathrm{var}(\mathring{E})$ which are of interest.

LEMMA 12.5.6. *Let $\mathring{E}$ be a well lhs-formed element template and $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \varepsilon$. Then there is $\mathbb{m}'$ such that $\emptyset \therefore (\mathrm{var}(\mathring{E}) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'$ and $\mathbb{m}'(\!|\mathring{E}|\!) = \varepsilon$.*

PROOF. *Let us distinguish the following cases by the structure of $\mathring{E}$. Let*

$\mathring{E} = x$: Take $\mathbb{m}' = \emptyset$. We have that $\mathrm{var}(\mathring{E}) \lhd \mathbb{m} = \emptyset$ and clearly $\mathbb{m}'(\!|\mathring{E}|\!) = \overline{x} = \varepsilon$.

$\mathring{E} = \mathring{x}$: From $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \varepsilon$ it follows that there is ι such that $\iota = \varepsilon = \mathbb{m}(\!|\mathring{E}|\!)$. Take $\mathbb{m}' = \{\mathring{x} \mapsto \iota\}$. Clearly $\mathbb{m}' = \mathrm{var}(\mathring{E}) \lhd \mathbb{m}$. Hence the claim.

$\mathring{E} = (\mathring{x}_1, \ldots, \mathring{x}_k)$: From $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \varepsilon$ it follows that there are ι_1, ..., ι_k such that $(\iota_1, \ldots, \iota_k) = \varepsilon = \mathbb{m}(\!|\mathring{E}|\!)$. Take $\mathbb{m}' = \{\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k\}$. We see that $\mathbb{m}'$ is a function because $\mathring{E}$ is a well lhs-formed element template and thus $\mathring{x}_i \neq \mathring{x}_j$ when $i \neq j$. Clearly $\mathbb{m}' = \mathrm{var}(\mathring{E}) \lhd \mathbb{m}$. Hence the claim.

$\mathring{E} = \mathtt{<}\mathring{m}_1, \ldots, \mathring{m}_k\mathtt{>}$: From $\mathbb{m}(\!|\mathring{E}|\!) \preccurlyeq \varepsilon$ it follows that there are some μ_1, ..., μ_k such that $\varepsilon = \mathtt{<}\mu_1, \ldots, \mu_k\mathtt{>}$ and that $\mathbb{m}(\mathring{m}_i) \preccurlyeq \mu_i$ holds for all $i \in \{1, \ldots, k\}$. Take $\mathbb{m}' = \{\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k\}$. We see that $\mathbb{m}'$ is a function because $\mathring{E}$ is a well lhs-formed element template and thus $\mathring{m}_i \neq \mathring{m}_j$ when $i \neq j$. Clearly $\mathbb{m}'(\!|\mathring{E}|\!) = \varepsilon$. Moreover $\emptyset \therefore (\mathrm{var}(\mathring{E}) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'$ follows from $\mathbb{m}(\mathring{m}_i) \preccurlyeq \mu_i$ shown above. Hence the claim. ∎

The following is an equivalent of the previous lemma for form templates. Note that the requirement that $\mathring{F}$ is a well lhs-formed form template is essential (as is the equivalent requirement of the previous lemma). Basically it says that no message variable appears in $\mathring{F}$ more than once. Consider $\mathring{F} = \mathtt{<}\mathring{\mathtt{M}}, \mathring{\mathtt{M}}\mathtt{>}$ which is not well lhs-formed and let $\mathbb{m} = \{\mathring{\mathtt{M}} \mapsto \{\mathsf{a}\}*\}$. Clearly $\mathbb{m}(\!|\mathring{F}|\!) = \mathtt{<}\{\mathsf{a}\}*, \{\mathsf{a}\}*\mathtt{>} \preccurlyeq \mathtt{<}\{\mathsf{a}\}*, \{\mathsf{a}, \mathsf{b}\}*\mathtt{>}$ but there is no $\mathbb{m}'$ that would instantiate $\mathring{F}$ to $\mathtt{<}\{\mathsf{a}\}*, \{\mathsf{a}, \mathsf{b}\}*\mathtt{>}$.

LEMMA 12.5.7. *Let $\mathring{F}$ be a well lhs-formed form template and $\mathbb{m}(\!|\mathring{F}|\!) \preccurlyeq \varphi$. Then there is $\mathbb{m}'$ such that $\emptyset \therefore (\mathrm{var}(\mathring{F}) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'$ and $\mathbb{m}'(\!|\mathring{F}|\!) = \varphi$.*

PROOF. *We know that $\mathring{F} = \mathring{E}_0 \ldots \mathring{E}_k$ and $\varphi = \varepsilon_0 \ldots \varepsilon_k$. Now $\mathbb{m}(\!|\mathring{F}|\!) \preccurlyeq \varphi$ implies that $\mathbb{m}(\!|\mathring{E}_i|\!) \preccurlyeq \varepsilon_i$ hold for all $i \in \{0, \ldots, k\}$. Thus by Lemma 12.5.6 for every $i \in \{0, \ldots, k\}$ there is $\mathbb{m}'_i$ such that $\emptyset \therefore (\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'_i$ and $\mathbb{m}'_i(\!|\mathring{E}_i|\!) = \varepsilon_i$. Let us take $\mathbb{m}' = \mathbb{m}'_0 \cup \cdots \cup \mathbb{m}'_k$. Firstly, we need to prove that $\mathbb{m}'$ is a function, that is, that $\mathbb{m}'_i(\mathring{z}) = \mathbb{m}'_j(\mathring{z})$ whenever $\mathring{z} \in \mathrm{dom}(\mathbb{m}'_i) \cap \mathrm{dom}(\mathbb{m}'_j)$ for some i and j. We see that $\mathrm{dom}(\mathbb{m}'_i) = \mathrm{var}(\mathring{E}_i)$ holds for all $i \in \{0, \ldots, k\}$. Clearly $\mathring{F}$ does not contain any*

process variable. Moreover, we know that $\mathring{F}$ is a well lhs-formed form template and thus no message variable is both $\mathrm{var}(\mathring{E}_i)$ and $\mathrm{var}(\mathring{E}_j)$ when $i \neq j$. Thus the only case when $\mathring{z} \in \mathrm{dom}(\mathbb{m}'_i) \cap \mathrm{dom}(\mathbb{m}'_j)$ is when $\mathring{z}$ is a name variable, say $\mathring{x}$. But than $\mathbb{m}'_i(\mathring{x}) = \mathbb{m}(\mathring{x}) = \mathbb{m}'_j(\mathring{x})$ because $\emptyset \therefore (\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'_i$ and $\emptyset \therefore (\mathrm{var}(\mathring{E}_j) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'_j$. Thus we see that $\mathbb{m}'$ is a function. Moreover we see that $\mathbb{m}'(\mathring{z}) = \mathbb{m}'_i(\mathring{z})$ holds for any i and for all $\mathring{z} \in \mathrm{var}(\mathring{E}_i)$. Hence the claim. ∎

Finally, the following lemma states the relation between nesting of shape predicates and nesting of type instantiations.

Lemma 12.5.8. *Let $\mathring{P}$ be a well formed lhs-template and let $\delta \therefore \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$. When $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$ then there is $\mathbb{m}'$ such that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$ and $\mathbb{m}' \models_{\llcorner} \mathring{P} : \langle \Gamma', \delta(\chi) \rangle$.*

Proof. *Let $\mathring{P}$ be a well formed lhs-template and let $\delta \therefore \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$. Let $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$. Let us prove the claim by induction on the structure of $\mathring{P}$. Let*

$\mathring{P} = \mathsf{0}$**:** *Take $\mathbb{m}' = \mathbb{m}$. The claim is clear.*

$\mathring{P} = \mathring{p}$**:** *Here $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi \rangle$ implies that $\mathbb{m}(\mathring{p}) = \chi$. Take $\mathbb{m}' = \mathbb{m}[\mathring{p} \mapsto \delta(\chi)]$. Hence the claim because $\mathbb{m}'(\mathring{p}) = \delta(\chi) = \delta(\mathbb{m}(\mathring{p}))$*

$\mathring{P} = \mathring{F}.\mathring{P}_1$**:** *Let $\varphi = \mathbb{m}(\!|\mathring{F}|\!)$. We know that there is some χ_1 such that $\mathbb{m} \models_{\llcorner} \mathring{P}_1 : \langle \Gamma, \chi_1 \rangle$ and $(\chi \xrightarrow{\varphi} \chi_1) \in \Gamma$. Now $\delta \therefore \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$ implies that there is $(\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_1)) \in \Gamma'$ with $\varphi \preccurlyeq \varphi'$. Thus by Lemma 12.5.7 there is $\mathbb{m}'_0$ such that $\emptyset \therefore (\mathrm{var}(\mathring{F}) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'_0$ and $\mathbb{m}'_0(\!|\mathring{F}|\!) = \varphi'$. From $\mathbb{m} \models_{\llcorner} \mathring{P}_1 : \langle \Gamma, \chi_1 \rangle$ by the induction hypothesis we obtain that there is some $\mathbb{m}'_1$ such that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'_1$ and $\mathbb{m}'_1 \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi_1) \rangle$. Let us take*

$$\mathbb{m}'(\mathring{z}) = \begin{cases} \mathbb{m}'_0(\mathring{z}) & \text{if } \mathring{z} \in \mathrm{var}(\mathring{F}) \\ \mathbb{m}'_1(\mathring{z}) & \text{otherwise} \end{cases}$$

Firstly, we prove that $\mathbb{m}'(\mathring{z}) = \mathbb{m}'_1(\mathring{z})$ for all $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$. This clearly holds when $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$ and $\mathring{z} \notin \mathrm{var}(\mathring{F})$. When $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$ and $\mathring{z} \in \mathrm{var}(\mathring{F})$ then we can see that $\mathring{z}$ has to be a name variable, say $\mathring{x}$, because $\mathring{P}$ is a well formed lhs-template. But then $\emptyset \therefore (\mathrm{var}(\mathring{F}) \lhd \mathbb{m}) \trianglelefteq \mathbb{m}'_0$ and $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'_1$ implies that $\mathbb{m}'(\mathring{x}) = \mathbb{m}'_0(\mathring{x}) = \mathbb{m}(\mathring{x}) = \mathbb{m}'_1(\mathring{x})$. Hence $\mathrm{var}(\mathring{P}_1) \lhd \mathbb{m}'_1 \subseteq \mathbb{m}'$ and also it is clear that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$. Now from $\mathbb{m}'_1 \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi_1) \rangle$ and from the above we obtain by Lemma 8.6.2 and Lemma 8.6.1 that $\mathbb{m}' \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi_1) \rangle$. It is clear that $\mathbb{m}'(\!|\mathring{F}|\!) = \mathbb{m}'_0(\!|\mathring{F}|\!) = \varphi'$. Hence the claim because $(\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_1)) \in \Gamma'$ was shown above.

$\mathring{P} = \mathring{P}_0 \mid \mathring{P}_1$**:** *We know that $\mathbb{m} \models_{\llcorner} \mathring{P}_0 : \langle \Gamma, \chi \rangle$ and $\mathbb{m} \models_{\llcorner} \mathring{P}_1 : \langle \Gamma, \chi \rangle$. By the induction hypothesis we obtain that there are some $\mathbb{m}'_0$ and $\mathbb{m}'_1$ such that $\delta \therefore \mathbb{m} \trianglelefteq \mathbb{m}'_0$ and*

$\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'_1$ *and* $\mathbb{m}'_0 \models_{\llcorner} \mathring{P}_0 : \langle \Gamma', \delta(\chi) \rangle$ *and* $\mathbb{m}'_1 \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi) \rangle$*. Let us take*

$$\mathbb{m}'(\mathring{z}) = \begin{cases} \mathbb{m}'_0(\mathring{z}) & \text{if } \mathring{z} \in \mathrm{var}(\mathring{P}_0) \\ \mathbb{m}'_1(\mathring{z}) & \text{otherwise} \end{cases}$$

Firstly, we prove that $\mathbb{m}'(\mathring{z}) = \mathbb{m}'_1(\mathring{z})$ *for all* $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$*. This clearly holds when* $\mathring{z} \notin \mathrm{var}(\mathring{P}_0)$ *and* $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$*. When* $\mathring{z} \in \mathrm{var}(\mathring{P}_0)$ *and* $\mathring{z} \in \mathrm{var}(\mathring{P}_1)$ *then we can see that* $\mathring{z}$ *has to be a name variable, say* $\mathring{x}$*, because* $\mathring{P}$ *is a well formed lhs-template. But then* $\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'_1$ *and* $\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'_0$ *implies that* $\mathbb{m}'_1(\mathring{x}) = \mathbb{m}(\mathring{x}) = \mathbb{m}'_0(\mathring{x}) = \mathbb{m}'(\mathring{x})$*. Hence* $\mathrm{var}(\mathring{P}_1) \lhd \mathbb{m}'_1 \subseteq \mathbb{m}'$ *and also it is clear that* $\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'$*. Now from* $\mathbb{m}'_0 \models_{\llcorner} \mathring{P}_0 : \langle \Gamma', \delta(\chi) \rangle$ *and* $\mathbb{m}'_1 \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi) \rangle$ *and from the above we obtain by Lemma 8.6.2 and Lemma 8.6.1 that* $\mathbb{m}' \models_{\llcorner} \mathring{P}_0 : \langle \Gamma', \delta(\chi) \rangle$ *and* $\mathbb{m}' \models_{\llcorner} \mathring{P}_1 : \langle \Gamma', \delta(\chi) \rangle$*. Hence the claim.*

otherwise: $\mathring{P}$ *is a well formed lhs-template and thus condition* L6 *ensures that the above cases cover all possibilities.* ∎

The following lemma states that any nesting $\delta \mathrel{\dot{\cdot}} \Pi \trianglelefteq \Pi'$ maps an active node of Π to an active node of Π' (w.r.t. the same $\mathcal{R}$).

Lemma 12.5.9. *Let* $\delta \mathrel{\dot{\cdot}} \Pi \trianglelefteq \Pi'$*. Then*

$$\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi) \quad \textit{implies} \quad \delta(\chi) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi').$$

Proof. *Let* $\delta \mathrel{\dot{\cdot}} \Pi \trianglelefteq \Pi'$ *and let* $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$*. Let* $\Pi = \langle \Gamma, \chi_{\mathsf{r}} \rangle$*. We know that* $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ *is a finite set and thus it follows from Definition 7.6.7 that there is a finite sequence of nodes* $\chi_0, \ldots, \chi_k$ *such that* $\chi_0 = \chi_{\mathsf{r}}$*,* $\chi_k = \chi$*, and moreover that* $\chi_i \in \mathrm{ActiveSucc}_{\mathcal{R}}(\chi_{i-1}, \Gamma)$ *holds for all* $i \in \{1, \ldots, k\}$*. Let us prove by induction on* i *that* $\delta(\chi_i) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi')$*. For* $i = 0$ *we know that* $\delta(\chi_0) = \delta(\chi_{\mathsf{r}})$ *is the root node of* Π' *which is an active node. Now let* $\delta(\chi_i) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi')$ *for some* $i < k$*. We want to prove that* $\delta(\chi_{i+1}) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi')$*. We know that* $\chi_{i+1} \in \mathrm{ActiveSucc}_{\mathcal{R}}(\chi_i, \Gamma)$ *and thus there are* $\mathbf{active}\{\, \mathring{p} \,\mathbf{in}\, \mathring{P} \,\} \in \mathcal{R}$ *and* $\mathbb{m}$ *such that* $\mathbb{m} \models_{\llcorner} \mathring{P} : \langle \Gamma, \chi_i \rangle$ *and* $\mathbb{m}(\mathring{p}) = \chi_{i+1}$*. Now Lemma 12.5.8 gives us* $\mathbb{m}'$ *such that* $\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'$ *and* $\mathbb{m}' \models_{\llcorner} \mathring{P} : \langle \Gamma', \delta(\chi_i) \rangle$*. Thus we see that* $\mathbb{m}'(\mathring{p}) \in \mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma', \delta(\chi_i))$*. But it means that* $\mathbb{m}'(\mathring{p}) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi')$ *because* $\delta(\chi_i) \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi')$ *by the induction hypothesis. Finally* $\delta \mathrel{\dot{\cdot}} \mathbb{m} \trianglelefteq \mathbb{m}'$ *gives us that* $\mathbb{m}'(\mathring{p}) = \delta(\mathbb{m}(\mathring{p})) = \delta(\chi_{i+1})$*. Hence the claim.* ∎

12.6 Properties of the Initial Shape Predicate

In this section we prove termination, correctness, and completeness of `ProcessShape`. We start by proving corresponding properties of `SequenceTypeSet`, `MessageType`,

`ElementType`, and `FormType`. For each of these four algorithms we prove its termination, correctness, and completeness together. This combined property is similar all the four algorithms and it says that the algorithm terminates for every input entity and it returns the principal type of the input.

LEMMA 12.6.1 (`SequenceTypeSet` PROPERTIES). `SequenceTypeSet` *terminates for any* M *and its return value* $\Sigma =$ `SequenceTypeSet(`M`)` *is the principal sequence type set of* M.

PROOF. *By induction on the structure of* M. *We prove the following three claims, that (1)* `SequenceTypeSet(`M`)` *terminates, that (2) it returns a sequence type set of* M *which (3) is the principal sequence type set. To prove (3) let us take* Σ' *such that* $\vdash M : \Sigma'$. *We shall prove that* $\Sigma \preccurlyeq \Sigma'$. *Let*

$M =$ `0`: *Termination (1) is obvious. The return value is* $\Sigma = \emptyset$ *and thus (2) is clear. Now* $\{\mathsf{0}\} = [\![\emptyset]\!] \subseteq [\![\Sigma']\!]$ *and hence (3).*

$M = x_0 \ldots x_k$: *Let* $\sigma = \overline{x_0} \cdots \overline{x_k}$. *Termination (1) is clear and the algorithm returns* $\Sigma = \{\sigma\}$. *Thus (2) holds because* $\vdash M : \sigma$. *Let us prove (3). We have* $\vdash x_0 \cdots x_k : \Sigma'$ *which implies* $\sigma \in \Sigma'$ *because there is no sequence type* σ' *of* M *other than* σ. *Thus* $\vdash \sigma : \Sigma'$ *and hence the claim.*

$M = M_0.M_1$: *By the induction hypothesis and by the fact that* M *is finite we obtain that both recursive calls terminate and their results*

$$\Sigma_0 = \texttt{SequenceTypeSet(}M_0\texttt{)} \qquad \Sigma_1 = \texttt{SequenceTypeSet(}M_1\texttt{)}$$

are in turn principal sequence type sets of M_0 *and* M_1. *We have* $\Sigma = \Sigma_0 \cup \Sigma_1$ *and thus* $\vdash M_0 : \Sigma$ *and* $\vdash M_1 : \Sigma$. *Thus (2) holds. From the principality of* Σ_0 *it follows that only a sequence type of some sequence in* M_0 *can be contained in* Σ_0. *Similarly for* Σ_1 *and* M_0. *All these sequence types have to be present in* Σ' *because* $\vdash M : \Sigma'$. *Thus* $\Sigma \subseteq \Sigma'$ *which implies* $\Sigma \preccurlyeq \Sigma'$. *Hence (3).* ∎

The termination, correctness, and completeness of `MessageType` is as follows. Note that the relation between the principal sequence type set Σ of a message M and the principal message type μ of M is not $\mu = \Sigma*$ only when M is a name. For example, the principal sequence type set of $M =$ `a` is $\{\mathsf{a}\}$ but the principal message type of M is just `a`.

LEMMA 12.6.2 (`MessageType` PROPERTIES). `MessageType` *terminates for any* M *and its return value* $\mu =$ `MessageType(`M`)` *is the principal message type of* M.

PROOF. *When* $M = x$ *then* `MessageType` *terminates and returns* $\overline{x}$ *which is a valid message type of* x *because* $\vdash x : \overline{x}$. *Moreover* $\overline{x}$ *is the only possible message type of* x

because a single name message can not have a starred message type of the sequence $\Sigma*$*. Thus the claim holds when* $M = x$*.*

Now suppose $M \neq x$ *for any* x*. The algorithm terminates by Lemma 12.6.1 and it returns* $\Sigma*$ *where* Σ *is the principal sequence type set of* M*. Thus* $\vdash M : \Sigma*$*. Take some* μ' *such that* $\vdash M : \mu'$*. Obviously there is* Σ' *such that* $\mu = \Sigma'*$ *and thus also* $\vdash M : \Sigma'$*. It is easy to see that* $[\![\mu]\!] = [\![\Sigma]\!] \backslash \mathrm{Name}$ *and* $[\![\mu']\!] = [\![\Sigma']\!] \backslash \mathrm{Name}$*. Thus* $\mu \preccurlyeq \mu'$ *follows from* $\Sigma \preccurlyeq \Sigma'$*, that is, from the principality of* Σ*.* ∎

The termination, correctness, and completeness of `ElementType` is stated as follows.

LEMMA 12.6.3 (`ElementType` PROPERTIES). `ElementType` *terminates for any* E *and its return value* $\varepsilon =$ `ElementType(`E`)` *is the principal element type of* E*.*

PROOF. *Let*

$E = x$**:** *Then the algorithm terminates and it returns the only possible (and thus principal) element type* $\overline{x}$ *of* x*.*

$E =$ `(`$x_1, \ldots, x_k$`)`**:** *As in the previous case, the algorithm terminates and it returns the only possible (and thus principal) element type* $(\overline{x_1}, \ldots, \overline{x_k})$ *of* $(x_1, \ldots, x_k)$*.*

$E =$ `<`$M_1, \ldots, M_k$`>`**:** *Thus the algorithm terminates by Lemma 12.6.2 and it returns* $\varepsilon =$ `<`$\mu_1, \ldots, \mu_k$`>` *where* μ_i *is the principal message type of* M_i *for* $i \in \{1, \ldots, k\}$*. Thus* $\vdash E : \varepsilon$*. Take* ε' *such that* $\vdash E : \varepsilon'$*. Obviously* $\varepsilon' =$ `<`$\mu'_1, \ldots, \mu'_k$`>` *for some* μ'_1*, ...,* μ'_k *such that* $\vdash M_i : \mu'_i$ *holds for all* $i \in \{1, \ldots, k\}$*. Clearly, when* $\vdash E' : \varepsilon$ *for some* E' *then* $E' =$ `<`$M'_1, \ldots, M'_k$`>` *for some* M'_1*, ...,* M'_k *such that* $\vdash M'_i : \mu_i$*. Thus* $\vdash M'_i : \mu'_i$ *follows from the principality of message types* μ_i*. Hence* $\vdash E' : \varepsilon'$ *and thus* $\varepsilon \preccurlyeq \varepsilon'$*.* ∎

Finally the termination, correctness, and completeness of `FormType` is stated as follows.

LEMMA 12.6.4 (`FormType` PROPERTIES). `FormType` *terminates for any* F *and its return value* $\varphi =$ `FormType(`F`)` *is the principal form type of* F*.*

PROOF. *Let* $F = E_0 \ldots E_k$*. The algorithm terminates by Lemma 12.6.3 and it returns* $\varphi = \varepsilon_0 \ldots \varepsilon_k$ *where* ε_i *is the principal element type of* E_i *for all* $i \in \{0, \ldots, k\}$*. Thus* $\vdash F : \varphi$*. Let* $\vdash F : \varphi'$ *for some* φ'*. From the typing rules it follows that* $\varphi' = \varepsilon'_0 \ldots \varepsilon'_k$ *for some* ε'_0*, ...,* ε'_k *with* $\vdash E_i : \varepsilon'_i$*. It is easy to see that*

$$\begin{aligned} [\![\varphi]\!] &= \{E''_0 \ldots E''_k : E''_i \in [\![\varepsilon_i]\!] \;\&\; i \in \{0, \ldots, k\}\} \\ [\![\varphi']\!] &= \{E''_0 \ldots E''_k : E''_i \in [\![\varepsilon'_i]\!] \;\&\; i \in \{0, \ldots, k\}\} \end{aligned}$$

Thus $\varphi \preccurlyeq \varphi'$ *follows from* $\varepsilon_i \preccurlyeq \varepsilon'_i$*, that is, from the principality of element types* ε_i*.* ∎

The termination, correctness, and completeness of `ProcessShape` are stated separately by the following three propositions.

PROPOSITION 12.6.5 (`ProcessShape` TERMINATION). `ProcessShape(`P`)` *terminates for every* P.

PROOF. *By induction on the structure of* P *because* P *is a finite object and using Lemma 12.6.4.* ∎

The correctness of `ProcessShape(`P`)` simply says that it returns a shape predicate matching P. The resulting shape predicate does not necessarily (and in most cases it will not) be an $\mathcal{R}$-type.

PROPOSITION 12.6.6 (`ProcessShape` CORRECTNESS). *Let* `ProcessShape(`P`)` $= \Pi$. *Then* $\vdash P : \Pi$.

PROOF. *Let* `ProcessShape(`P`)` $= \Pi$. *Let us prove the claim by induction on the structure of* P. *Let*

$P =$ `0`: *Clearly* $\Pi = \langle \emptyset, \mathtt{R} \rangle$ *and* $\vdash P : \Pi$.

$P = F.P_0$: *We see that* $\Pi = \langle \{\chi \xrightarrow{\varphi} \chi_0\} \cup \Gamma_0, \chi \rangle$ *where* $\langle \Gamma_0, \chi_0 \rangle =$ `ProcessShape(`P_0`)` *and* $\varphi =$ `FormType(`F`)` *and* χ *is a node fresh for* Γ_0. *By the induction hypothesis we obtain that* $\vdash P_0 : \langle \Gamma_0, \chi_0 \rangle$. *By Lemma 12.6.4 we obtain that* $\vdash F : \varphi$. *Hence* $\vdash F.P_0 : \Pi$.

$P = P_0 \mid P_1$: *Let* Γ_0, Γ_0', Γ_0'', Γ_1, χ_0, *and* χ_1 *be values of variables of the corresponding names at the time of execution of line 13. We can see that* $\Pi = \langle \Gamma_0 \cup \Gamma_1'', \chi_0 \rangle$. *By the induction hypothesis we obtain that* $\vdash P_0 : \langle \Gamma_0, \chi_0 \rangle$ *and* $\vdash P_1 : \langle \Gamma_1, \chi_1 \rangle$. *Now it holds that* $\vdash P_1 : \langle \Gamma_1', \chi_1 \rangle$ *because* Γ_1 *and* Γ_1' *have the same structure, they differ only in names of nodes which do not participate in matching of processes against shape predicates. Clearly also* $\vdash P_1 : \langle \Gamma_1'', \chi_0 \rangle$ *because* Γ_1'' *is* Γ_1' *with node* χ_1 *renamed to* χ_0. *Hence the claim.*

$P = \nu x.P_0$: *We see that* $\Pi =$ `ProcessShape(`P_0`)`. *By the induction hypothesis we obtain that* $\vdash P_0 : \Pi$. *Hence* $\vdash \nu x.P_0 : \Pi$.

$P = !P_0$: *We see that* $\Pi =$ `ProcessShape(`P_0`)`. *By the induction hypothesis we obtain that* $\vdash P_0 : \Pi$. *Hence* $\vdash !P_0 : \Pi$. ∎

The completeness of `ProcessShape` says that the algorithm returns a shape predicate that is $\mathcal{R}$-preprincipal for the input P. Note this holds for an arbitrary $\mathcal{R}$. The completeness can be equivalently expressed as that the resulting shape predicate is $\emptyset$-preprincipal for P.

PROPOSITION 12.6.7 (`ProcessShape` COMPLETENESS). *Let* $\mathcal{R}$ *be arbitrary and let* $\Pi =$ `ProcessShape(`P`)`. *Then* Π *is* $\mathcal{R}$*-preprincipal for* P.

PROOF. *Let* $\Pi = \texttt{ProcessShape}(P)$. *Let* $\Pi' = \langle \Gamma', \chi' \rangle$ *be a restricted* $\mathcal{R}$*-type of* P. *We want to find* δ *such that* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$. *Let us prove the claim by induction on the structure of* P. *Let*

$P = \texttt{0}$: *Then* $\Pi = \langle \emptyset, \texttt{R} \rangle$. *Let* $\delta = \{\texttt{R} \mapsto \chi'\}$. *Clearly* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$.

$P = F.P_0$: *Then* $\Pi = \langle \Gamma_0 \cup \{\chi \xrightarrow{\varphi} \chi_0\}, \chi \rangle$ *where* $\langle \Gamma_0, \chi_0 \rangle = \texttt{ProcessShape}(P_0)$, *and* χ *is a node fresh for* Γ_0, *and* $\varphi = \texttt{FormType}(F)$. *We know* $\vdash P : \Pi'$ *and thus there are some* χ'_0, φ' *such that* $\vdash F : \varphi'$ *and* $(\chi' \xrightarrow{\varphi'} \chi'_0) \in \Gamma'$ *and* $\vdash P_0 : \langle \Gamma', \chi'_0 \rangle$. *By the induction hypothesis we have that there is some* δ_0 *such that* $\delta_0 \mathrel{\therefore} \langle \Gamma_0, \chi_0 \rangle \trianglelefteq \langle \Gamma', \chi'_0 \rangle$. *Thus* $\delta_0(\chi_0) = \chi'_0$. *By Lemma 12.6.4 we obtain that* $\vdash F : \varphi$ *and that* $\varphi \preccurlyeq \varphi'$. *Let us define* $\delta = \delta_0[\chi \mapsto \chi']$. *Clearly* δ *is defined for all nodes of* Π. *It is easy to see that* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$ *because for the edge* $\chi \xrightarrow{\varphi} \chi_0$ *from* Π *there is the edge* $\chi' = \delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0) = \chi'_0$ *in* Π' *with* $\varphi \preccurlyeq \varphi'$ *as required. The existence of the other required edges follows from* $\delta_0 \mathrel{\therefore} \langle \Gamma_0, \chi_0 \rangle \trianglelefteq \langle \Gamma', \chi'_0 \rangle$.

$P = P_0 \mid P_1$: *Then* $\Pi = \langle \Gamma_0 \cup \Gamma''_1, \chi_0 \rangle$ *where* $\langle \Gamma_0, \chi_0 \rangle = \texttt{ProcessShape}(P_0)$ *and* $\langle \Gamma_1, \chi_1 \rangle = \texttt{ProcessShape}(P_1)$ *and* Γ''_1 *is* Γ_1 *with the node* χ_1 *renamed to* χ_0 *and with the other nodes made distinct from the nodes in* Γ_0. *Thus there is some node renaming* δ'_1 *such that* $\Gamma''_1 = \delta'_1(\Gamma_1)$ *and also* $\delta'_1(\chi_1) = \chi_0$. *We know* $\vdash P : \Pi'$ *and thus* $\vdash P_0 : \Pi'$ *and* $\vdash P_1 : \Pi'$. *Thus by the induction hypothesis we obtain* δ_0 *and* δ_1 *such that* $\delta_0 \mathrel{\therefore} \langle \Gamma_0, \chi_0 \rangle \trianglelefteq \Pi'$ *and* $\delta_1 \mathrel{\therefore} \langle \Gamma_1, \chi_1 \rangle \trianglelefteq \Pi'$. *Clearly* $(\delta'_1)^{-1}$ *is a function a thus we can define* δ *on the nodes of* Π *as follows.*

$$\delta(\chi) = \begin{cases} \delta_0(\chi) & \text{if } \chi \text{ is in } \Gamma_0 \\ \delta_1(\delta_1'^{-1}(\chi)) & \text{otherwise} \end{cases}$$

We see that $\delta_0(\chi_0) = \delta_1(\delta_1'^{-1}(\chi_0))$ *where* χ_0 *is the only node both in* Γ_0 *and* Γ_1. *Let us prove that* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi'$. *When* $\chi_2 \xrightarrow{\varphi} \chi_3$ *is in* Π *then either* $(\chi_2 \xrightarrow{\varphi} \chi_3) \in \Gamma_0$ *or* $(\chi_2 \xrightarrow{\varphi} \chi_3) \in \Gamma''_1$. *In the first case clearly there is some* $\delta(\chi_2) \xrightarrow{\varphi'} \delta(\chi_3)$ *in* Π' *with* $\varphi \preccurlyeq \varphi'$ *because* $\delta(\chi_2) = \delta_0(\chi_2)$ *and* $\delta(\chi_3) = \delta_0(\chi_3)$. *In the second case, when* $(\chi_2 \xrightarrow{\varphi} \chi_3) \in \Gamma''_1$ *we have that* $(\delta_1'^{-1}(\chi_2) \xrightarrow{\varphi} \delta_1'^{-1}(\chi_3)) \in \Gamma_1$. *Because* $\delta_1 \mathrel{\therefore} \langle \Gamma_1, \chi_1 \rangle \trianglelefteq \Pi'$, *we have that there is some* $\delta(\chi_2) \xrightarrow{\varphi'} \delta(\chi_3)$ *in* Π' *with* $\varphi \preccurlyeq \varphi'$ *as required. The situation with flow edges is analogous. Hence the claim.*

$P = \nu x.P_0$: *The claim follows directly from the induction hypothesis.*

$P = !P_0$: *The claim follows directly from the induction hypothesis.* ∎

12.7 Properties of the Restriction Algorithm

In this section we prove termination, correctness, and completeness of the restriction algorithm, that is, of `RestrictGraph` and of its two subroutines `RestrictWidth` and

`RestrictDepth`.

12.7.1 Properties of `RestrictWidth`

Firstly we prove that `RestrictWidth` terminates for every input.

LEMMA 12.7.1. `RestrictWidth`(Π) *terminates for every* Π.

PROOF. *The algorithm increases the number of nodes in the graph assigned to variable* Γ *by one with each iteration of the* **while** *cycle. Thus the algorithm has to terminate after finitely steps because* Π *has only finitely many nodes.* ∎

The following property is used to prove termination of `PrincipalType`. It implies that paths(Π) never decreases during the execution of `PrincipalType`. Lemma 12.7.11 which proves that the restriction algorithm does not decrease number of almost distinct paths in a shape predicate uses this lemma. The proof contains an inductive definition of a node renaming δ which is used in other proofs.

LEMMA 12.7.2. *Let* $\Pi =$ `RestrictWidth`(Π_0). *Then there is a node renaming* δ *of* Π_0 *such that* $\delta(\Pi_0) = \Pi$.

PROOF. *Let* $\Pi =$ `RestrictWidth`(Π_0). *Let* $\Pi_0 = \langle \Gamma_0, \chi_\mathsf{r} \rangle$. *From Lemma 12.7.1 we know that* `RestrictWidth`(Π_0) *terminates and thus that* **while** *is executed only finitely many time during the execution of* `RestrictWidth`(Π_0). *Let it be executed* n *times.*

Let δ^0_ren *be the identity on the nodes of* Π_0. *We shall construct the sequence* δ^1_ren, ..., δ^n_ren *of node renaming inductively as follows. Let* χ^i_0 *be the value of variable* χ_0 *during the* i*-th iteration of the* **while** *cycle when computing* `RestrictWidth`(Π_0). *Similarly, let* χ^i_1 *and* χ'_i *be the values of* χ_1 *and* χ' *respectively. Let* $\delta^i_\mathsf{ren}(\chi)$ *be defined for any node* χ *from* Π_0 *as follows.*

$$\delta^i_\mathsf{ren}(\chi) = \begin{cases} \chi'_i & \text{if } \delta^{i-1}_\mathsf{ren}(\chi) = \chi^i_0 \text{ or } \delta^{i-1}_\mathsf{ren}(\chi) = \chi^i_1 \\ \delta^{i-1}_\mathsf{ren}(\chi) & \text{otherwise} \end{cases}$$

Let $\Pi^i = \langle \Gamma^i, \chi_\mathsf{r} \rangle$ *where* Γ^i *is the value variable* Γ *after the execution of the* i*-th iteration of the* **while** *cycle (*$\Gamma^0 = \Gamma_0$*). It is easy to prove by induction on* i *that* δ^i_ren *is a node renaming such that* $\delta^i_\mathsf{ren}(\Pi_0) = \Pi^i$ *for any* $i \in \{0, \ldots, n\}$. *We can also see that* $\Pi^n = \Pi$. *Finally let* $\delta = \delta^n_\mathsf{ren}$. *Hence the claim* $\delta(\Pi_0) = \Pi$. ∎

The following proves the correctness of `RestrictWidth` which is that when the algorithm returns its input unchanged then the input is width-restricted.

LEMMA 12.7.3 (`RestrictWidth` CORRECTNESS). *When* `RestrictWidth`$(\Pi) = \Pi$ *then* Π *is width-restricted.*

PROOF. *Let* `RestrictWidth`$(\Pi) = \Pi$. *Clearly the* **while** *cycle was not executed because otherwise the returned graph would have less nodes than the input graph and they could not be equal. Thus the condition of the* **while** *cycle is not satisfied which directly implies the claim.* ∎

The completeness of `RestrictWidth` says that the algorithm preserve existence of nesting of the input in any other restricted shape predicate. The same node map which defines the nesting of the input in some restricted shape predicate defines the nesting of the output in the same restricted shape predicated.

LEMMA 12.7.4 (`RestrictWidth` COMPLETENESS). *Let* Π_{restr} *be restricted and let* $\Pi =$ `RestrictWidth`(Π_0). *Then* $\delta \mathrel{\therefore} \Pi_0 \trianglelefteq \Pi_{\mathsf{restr}}$ *implies* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi_{\mathsf{restr}}$.

PROOF. *Let* Π_{restr} *be restricted and let* $\Pi =$ `RestrictWidth`(Π_0). *Let* $\Pi_{\mathsf{restr}} = \langle \Gamma_{\mathsf{restr}}, \chi_{\mathsf{restr}} \rangle$. *Let* $\Pi_0 = \langle \Gamma_0, \chi_{\mathsf{r}} \rangle$. *Let* $\delta \mathrel{\therefore} \Pi_0 \trianglelefteq \Pi_{\mathsf{restr}}$. *Let* n, χ_0^i, χ_1^i, χ_i', Γ^i, Π^i, *and* δ_{ren}^i *(for any* $i \in \{1, \ldots, n\}$*) be as in the proof of Lemma 12.7.2. Additionally let* χ^i *be the value of variable* χ *during the* i*-th iteration of the* **while** *cycle when computing* `RestrictWidth`(Π_0). *Similarly, let* φ_0^i *and* φ_1^i *be the values of* φ_0 *and* φ_1. *We have* $\Pi = \Pi^n = \delta_{\mathsf{ren}}^n(\Pi_0)$. *Let* $\Pi^0 = \Pi_0$.

We shall prove by induction on i *that* $\delta \mathrel{\therefore} \Pi^i \trianglelefteq \Pi_{\mathsf{restr}}$ *for all* $i \in \{0, \ldots, n\}$. *The case* $i = 0$ *follows directly from the assumptions because* $\Pi^0 = \Pi_0$. *Let* $i > 0$ *and* $\delta \mathrel{\therefore} \Pi^{i-1} \trianglelefteq \Pi_{\mathsf{restr}}$. *We know that* $\Pi^i = \delta_{\mathsf{ren}}^i(\Pi_0)$ *and* $\Pi^{i+1} = \delta_{\mathsf{ren}}^{i+1}(\Pi_0)$. *It means* Π^{i+1} *is obtained from* Π^i *by unification of* χ_0^i *and* χ_1^i *(that is, replacing both of them by* χ_i' *which is one of* χ_0^i *and* χ_1^i*). It is enough to prove that* $\delta(\chi_0^i) = \delta(\chi_1^i)$, *that is, that the nodes of* Π_0 *which are being unified in the* i*-th iteration of the* **while** *cycle are mapped by* δ *to the same node of* Π_{restr} *(they are already unified in* Π_{restr}*).*

Now let us prove that $\delta(\chi_0^i) = \delta(\chi_1^i)$. *We have* $(\chi^i \xrightarrow{\varphi_0^i} \chi_0^i) \in \Gamma^{i-1}$ *and* $(\chi^i \xrightarrow{\varphi_1^i} \chi_1^i) \in \Gamma^{i-1}$. *The induction hypothesis says that* $\delta \mathrel{\therefore} \Pi^{i-1} \trianglelefteq \Pi_{\mathsf{restr}}$ *and thus there are some* φ_0' *and* φ_1' *such that* $(\delta(\chi^i) \xrightarrow{\varphi_0'} \delta(\chi_0^i)) \in \Gamma_{\mathsf{restr}}$ *and* $(\delta(\chi^i) \xrightarrow{\varphi_1'} \delta(\chi_1^i)) \in \Gamma_{\mathsf{restr}}$ *and* $\varphi_0^i \preccurlyeq \varphi_0'$ *and* $\varphi_1^i \preccurlyeq \varphi_1'$. *We know that* $\varphi_0^i \approx \varphi_1^i$ *and thus the previous implies* $\varphi_0' \approx \varphi_1'$. *Thus* $\delta(\chi_0^i) = \delta(\chi_1^i)$ *because* Γ_{restr} *is width-restricted. Hence the claim.* ∎

12.7.2 Properties of `RestrictDepth`

The properties of `RestrictDepth` proved in this section are analogous to the properties of `RestrictWidth` from the previous section. The termination is stated is follows.

LEMMA 12.7.5 (`RestrictDepth` TERMINATION). `RestrictDepth`(Π) *terminates for every* Π.

PROOF. *The algorithm increases the number of nodes in the graph assigned to variable* Γ *by one with each iteration of the* **while** *cycle. Thus, because* Π *has only finitely many nodes, the algorithm has to terminate after finitely steps.* ∎

The following define the property of `RestrictDepth` used to prove that the restriction algorithm do not decrease the number of different almost disjoint edge paths (Lemma 12.7.11). Again, it also provides an induction definition of δ used in other proofs namely in the proof of the completeness of `RestrictDepth`.

LEMMA 12.7.6. *Let* $\Pi_0 =$ `RestrictDepth(`Π`)`*. Then there is a node renaming* δ *of* Π *such that* $\Pi_0 = \delta(\Pi)$*.*

PROOF. *Let* $\Pi_0 =$ `RestrictDepth(`Π`)`*. Let* $\Pi = \langle \Gamma, \chi_\mathsf{r} \rangle$*. From Lemma 12.7.5 we know that* `RestrictDepth(`Π`)` *terminates and thus that* **while** *is executed only finitely many time during the execution of* `RestrictDepth(`Π`)`*. Let it be executed* n *times.*

Let δ_0 *be the identity on the nodes of* Π*. We shall construct the sequence* δ^1_ren*, …,* δ^n_ren *of node maps inductively as follows. Let* χ^i_1 *be the value of variable* χ_1 *during the* i*-th iteration of the* **while** *cycle when computing* `RestrictDepth(`Π`)`*. Let* k_i *be the value of* k *in the* i*-th iteration. Similarly, let* $\chi^i_{k_i+1}$ *and* χ'_i *be the values of* χ_{k_i+1} *and* χ' *respectively. Let* $\delta^i_\mathsf{ren}(\chi)$ *be defined for any node* χ *from* Π *as follows.*

$$\delta^i_\mathsf{ren}(\chi) = \begin{cases} \chi'_i & \text{if } \delta^{i-1}_\mathsf{ren}(\chi) = \chi^i_1 \text{ or } \delta^{i-1}_\mathsf{ren}(\chi) = \chi^i_{k_i+1} \\ \delta^{i-1}_\mathsf{ren}(\chi) & \text{otherwise} \end{cases}$$

Let $\Pi^i = \langle \Gamma^i, \chi_\mathsf{r} \rangle$ *where* Γ^i *is the value variable* Γ *after the execution of the* i*-th iteration of the* **while** *cycle (*$\Gamma^0 = \Gamma$*). It is easy to prove by induction on* i *that* $\delta_i(\Pi) = \Pi^i$ *for any* $i \in \{0, \ldots, n\}$*. We can also see that* $\Pi^n = \Pi_0$*. Finally let* $\delta = \delta_n$*. Hence the claim* $\delta(\Pi) = \Pi_0$*.* ∎

The following defines and proves the correctness of `RestrictDepth`.

LEMMA 12.7.7 (`RestrictDepth` CORRECTNESS). *When* `RestrictDepth(`Π`)` $= \Pi$ *then* Π *satisfies the depth restriction.*

PROOF. *Let* `RestrictDepth(`Π`)` $= \Pi$*. Clearly the* **while** *cycle was not executed because otherwise the returned graph would have less nodes than the input graph and they could not be equal. Thus the condition of the* **while** *cycle is not satisfied which directly implies the claim.* ∎

The following proves the property of `RestrictDepth` which is essential for the completeness of `PrincipalType`.

LEMMA 12.7.8 (`RestrictDepth` COMPLETENESS). *Let* Π_{restr} *be restricted and let* $\Pi = \texttt{RestrictDepth}(\Pi_0)$. *Then* $\delta \mathrel{\therefore} \Pi_0 \trianglelefteq \Pi_{\mathsf{restr}}$ *implies* $\delta \mathrel{\therefore} \Pi \trianglelefteq \Pi_{\mathsf{restr}}$.

PROOF. *Let* Π_{restr} *be restricted and let* $\Pi = \texttt{RestrictDepth}(\Pi_0)$. *Let* $\Pi_{\mathsf{restr}} = \langle \Gamma_{\mathsf{restr}}, \chi_{\mathsf{restr}} \rangle$. *Let* $\Pi_0 = \langle \Gamma_0, \chi_{\mathsf{r}} \rangle$. *Let* $\delta \mathrel{\therefore} \Pi_0 \trianglelefteq \Pi_{\mathsf{restr}}$. *Let* n, Γ^i, Π^i, *and* δ^i_{ren} *(for any* $i \in \{1, \ldots, n\}$*) be as in the proof of Lemma 12.7.6. Additionally let* χ^i *be the value of variable* χ *during the* i*-th iteration of the* **while** *cycle when computing* $\texttt{RestrictDepth}(\Pi_0)$ *and let* k_i *be the value of* k *in the* i*-th iteration. Similarly, let* $\varphi^i_0, \ldots, \varphi^i_{k_i}$ *and* $\chi^i_1, \ldots, \chi^i_{k_i+1}$ *be the values of corresponding variables (that is,* $\varphi^i_{k_i}$ *is the value of* φ_k *[more precisely* φ_{k_i}*] in the* i*-th iteration and so on).*

We shall prove by induction on i *that* $\delta \mathrel{\therefore} \Pi^i \trianglelefteq \Pi_{\mathsf{restr}}$ *for all* $i \in \{0, \ldots, n\}$. *The case* $i = 0$ *follows directly from the assumptions because* $\Pi^0 = \Pi_0$. *Let* $i > 0$ *and* $\delta \mathrel{\therefore} \Pi^{i-1} \trianglelefteq \Pi_{\mathsf{restr}}$. *We know that* $\Pi^i = \delta^i_{\mathsf{ren}}(\Pi_0)$ *and* $\Pi^{i+1} = \delta^{i+1}_{\mathsf{ren}}(\Pi_0)$. *It means* Π^{i+1} *is obtained from* Π^i *by unification of* χ^i_1 *and* $\chi^i_{k_i+1}$ *(that is, replacing both of them by* χ'_i *which is one of* χ^i_1 *and* $\chi^i_{k_i+1}$*). It is enough to prove that* $\delta(\chi^i_1) = \delta(\chi^i_{k_i+1})$, *that is, that the nodes of* Π_0 *which are being unified in the* i*-th iteration of the* **while** *cycle are mapped by* δ *to the same node of* Π_{restr} *(they are already unified in* Π_{restr}*).*

Now let us prove that $\delta(\chi^i_1) = \delta(\chi^i_{k_i})$. *We know that there is the path* $\{\chi^i_0 \xrightarrow{\varphi^i_0} \chi^i_1 \xrightarrow{\varphi^i_1} \cdots \chi_{k_i} \xrightarrow{\varphi_{k_i}} \chi_{k_i+1}\} \subseteq \Gamma^i$. *The induction hypothesis says that* $\delta \mathrel{\therefore} \Pi^{i-1} \trianglelefteq \Pi_{\mathsf{restr}}$ *and thus there are some* $\varphi'_0, \ldots, \varphi'_{k_i}$ *with* $\varphi^i_j \preccurlyeq \varphi'_j$ *for all* $j \in \{0, \ldots, k_i\}$ *such that there is the path* $\{\delta(\chi^i_0) \xrightarrow{\varphi'_0} \delta(\chi^i_1) \xrightarrow{\varphi'_1} \cdots \delta(\chi_{k_i}) \xrightarrow{\varphi'_{k_i}} \delta(\chi_{k_i+1})\} \subseteq \Gamma_{\mathsf{restr}}$ *in* Π_{restr}. *We know that* $\varphi^i_0 \approx \varphi^i_{k_i}$ *and thus the previous implies* $\varphi'_0 \approx \varphi'_{k_i}$. *Thus* $\delta(\chi^i_1) = \delta(\chi^i_{k_i})$ *because* Γ_{restr} *is depth-restricted. Hence the claim.* ∎

12.7.3 Properties of `RestrictGraph`

The termination of `RestrictGraph` is closely related to the termination of its two subroutines. It is proved in the following lemma.

LEMMA 12.7.9 (`RestrictGraph` TERMINATION). $\texttt{RestrictGraph}(\Pi)$ *terminates for every* Π.

PROOF. *We can see that every call to* `RestrictWidth` *(resp.* `RestrictDepth`*) terminates by Lemma 12.7.1 (resp. Lemma 12.7.5). The number of nodes of the shape predicate assigned to variable* Π *is either decreased during the call to* `RestrictWidth` *(resp.* `RestrictDepth`*) or the whole shape predicate stays unchanged. Thus during every execution of the* **repeat** *cycle the number of nodes of* Π *is either decreased or* Π *stays unchanged. When it stays unchanged then the algorithm terminates. Thus the number of iterations of the* **repeat** *cycle is bound by the number of nodes in the input shape predicate which is a finite number.* ∎

The following lemma is used to prove that the restriction algorithm does not decrease the number of different almost disjoint edge paths (the next Lemma 12.7.11).

LEMMA 12.7.10. *Let* $\Pi_0 =$ `RestrictGraph(`Π`)`. *Then there is a node renaming* δ *of* Π *such that* $\Pi_0 = \delta(\Pi)$.

PROOF. *The node renaming* δ *is obtained by composition of the node renamings obtained from Lemma 12.7.2 and Lemma 12.7.6.* ∎

The following proves that the restriction algorithm does not decrease the number of different almost disjoint edge paths. It is used to prove the termination of `PrincipalType`.

LEMMA 12.7.11. *Let* $\Pi_0 =$ `RestrictGraph(`Π`)`. *Then* $\mathrm{paths}(\Pi) \leqslant \mathrm{paths}(\Pi_0)$.

PROOF. *Let* $\Pi = \langle \Gamma, \chi_\mathsf{r} \rangle$ *and* $\Pi_0 =$ `RestrictGraph(`Π`)`. *By Lemma 12.7.10 we have* δ *such that* $\delta(\Pi) = \Pi_0$. *Let* $\{\chi_\mathsf{r} \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi_k\}$ *be a rooted path in* Π *such that* $(\varphi_1, \ldots, \varphi_k)$ *is an almost disjoint edge path in* Π. *Then obviously*

$$\{\delta(\chi_\mathsf{r}) \xrightarrow{\varphi_1} \delta(\chi_1) \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \delta(\chi_k)\}$$

is a rooted path in $\delta(\Pi)$ *and that is why* $(\varphi_1, \ldots, \varphi_k)$ *is an almost disjoint edge path in* $\delta(\Pi)$. *Thus* $\mathrm{paths}(\Pi) \leqslant \mathrm{paths}(\delta(\Pi))$. ∎

The following lemma is used in the proof of the termination of `PrincipalType`. It helps to prove that when `LocalClosureStep` or `FlowClosureStep` adds a new form edge to Π then the number of different almost disjoint paths in Π is increased.

LEMMA 12.7.12. *Let* $\Pi_0 =$ `RestrictGraph(`Π`)`. *If there is a rooted path to every node in* Π *then there is a rooted path to every node in* Π_0 *such that the corresponding edge path in* Π_0 *is disjoint.*

PROOF. *Let* $\Pi_0 =$ `RestrictGraph(`Π`)`. *By Lemma 12.7.10 there is* δ *such that* $\Pi_0 = \delta(\Pi)$. *Let* χ' *be a node of* Π_0. *There is a node* χ *of* Π *such that* $\delta(\chi) = \chi'$. *By the assumption, there exists a rooted path* $\{\chi_\mathsf{r} \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi\}$ *to* χ *in* Π. *Then*

$$\{\delta(\chi_\mathsf{r}) \xrightarrow{\varphi_1} \delta(\chi_1) \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \delta(\chi)\}$$

is a rooted path to χ' *in* Π_0. *We know that* $\delta(\Pi)$ *satisfied the depth restriction. When* $\varphi_i = \varphi_j$ *for some* $i < j$ *then obviously* $\varphi_i \approx \varphi_j$. *Thus by the depth restriction it has to hold that* $\delta(\chi_i) = \delta(\chi_j)$. *Let us construct a path by removing the edges between* $\delta(\chi_i)$ *and* $\delta(\chi_j)$, *that is, shorten the above path*

$$\{\delta(\chi_\mathsf{r}) \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_i} \delta(\chi_i) \xrightarrow{\varphi_{i+1}} \cdots \xrightarrow{\varphi_j} \delta(\chi_j) \xrightarrow{\varphi_{j+1}} \cdots \xrightarrow{\varphi_k} \chi'\}$$

to

$$\{\delta(\chi_{\mathsf{r}}) \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{i-1}} \delta(\chi_{i-1}) \xrightarrow{\varphi_i} \delta(\chi_j) \xrightarrow{\varphi_{j+1}} \cdots \xrightarrow{\varphi_k} \chi'\}$$

Repeat this procedure until $\varphi_i \neq \varphi_j$ for all $i \neq j$ to obtain a rooted path to χ' in $\delta(\Pi)$. Clearly its corresponding edge path is disjoint. ∎

The following is used to prove the correctness of `RestrictGraph`.

LEMMA 12.7.13. *When* `RestrictDepth(RestrictWidth(`Π`))` $= \Pi$ *then* Π *is restricted.*

PROOF. *Let* `RestrictDepth(RestrictWidth(`Π`))` $= \Pi$. *We know that both the algorithms* `RestrictWidth` *and* `RestrictDepth` *preserves the root node and that none of these two algorithms can increase the number of nodes in the input shape predicate. When any of the two functions do not decrease the number of nodes than they return the input shape predicate unchanged. Thus it has to hold that* `RestrictWidth(`Π`)` $= \Pi$ *which gives us that* `RestrictDepth(`Π`)` $= \Pi$. *Thus* Π *satisfies the width restriction by Lemma 12.7.3 and the depth restriction by Lemma 12.7.7. Hence* Π *is restricted.* ∎

The following proves the correctness of `RestrictGraph`. This correctness property is slightly different from correctness properties of the subroutines `RestrictWidth` and `RestrictDepth`. The correctness of `RestrictGraph` its property which is required to prove the correctness of `PrincipalType`. On the other hand, correctness of `RestrictWidth` or `RestrictDepth` is its property required to prove the correctness of `RestrictGraph`.

LEMMA 12.7.14 (`RestrictGraph` CORRECTNESS). *Let* Π_0 = `RestrictGraph(`Π`)`. *Then* Π_0 *is restricted.*

PROOF. *Let* Π_0 = `RestrictGraph(`Π`)`. *The* **until** *condition at line 5 was satisfied in the last iteration of the* **repeat** *cycle and thus*

$$\texttt{RestrictDepth(RestrictWidth(}\Pi_0\texttt{))} = \Pi_0$$

because Π_0 *is the value of both variables* Π *and* Π_0 *at that point. Hence* Π_0 *is restricted by Lemma 12.7.13.* ∎

The completeness of `RestrictGraph` follows from the completeness of its subroutines `RestrictWidth` and `RestrictDepth`.

LEMMA 12.7.15 (`RestrictGraph` COMPLETENESS). *Let* Π_{restr} *be restricted and let* Π = `RestrictGraph(`Π_0`)`. *Then* $\delta \therefore \Pi_0 \trianglelefteq \Pi_{\mathsf{restr}}$ *implies* $\delta \therefore \Pi \trianglelefteq \Pi_{\mathsf{restr}}$.

PROOF. *Follows directly from Lemma 12.7.4 and Lemma 12.7.8.* ∎

12.8 Properties of the Local Closure Algorithm

The following notion of *compatible* type instantiations will be used in the proofs in this section. Two type instantiations are compatible when they agree on values of variables which are defined by both of them.

Definition 12.8.1. *Type instantiations* $\mathbb{T}$ *and* $\mathbb{T}'$ *are* **compatible** *iff for all variables* $\mathring{z} \in \mathrm{dom}(\mathbb{T}) \cap \mathrm{dom}(\mathbb{T}')$ *it holds that* $\mathbb{T}(\mathring{z}) = \mathbb{T}'(\mathring{z})$. ∎

12.8.1 Properties of `MatchElement`

We know that `MatchElement` returns either an empty set or an singleton set with one type instantiation. The correctness of `MatchElement` says that when $\mathbb{T}(\!|\mathring{E}|\!) = \varepsilon$ and $\mathrm{var}(\mathring{E}) = \mathrm{dom}(\mathbb{T})$ then `MatchElement(`$\emptyset$`,`$\mathring{E}$`,`ε`)` $= \{\mathbb{T}\}$. In other words, for a given $\mathring{E}$ and ε, `MatchElement` computes (representations of) all instantiations such that $\mathbb{T}(\!|\mathring{E}|\!) = \varepsilon$. The following lemma proves a slightly more general property which considers also type instantiations with $\mathrm{var}(\mathring{E}) \subset \mathrm{dom}(\mathbb{T})$ and nonempty accumulator $\mathbb{T}'$.

Lemma 12.8.2 (`MatchElement` Correctness). *Let* $\mathbb{T}$ *and* $\mathbb{T}'$ *be compatible. Then* $\mathbb{T}(\!|\mathring{E}|\!) = \varepsilon$ *implies*

$$\texttt{MatchElement(}\mathbb{T}', \mathring{E}, \varepsilon\texttt{)} = \{\mathbb{T}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{T})\}.$$

Proof. *Let* $\mathbb{T}$ *and* $\mathbb{T}'$ *be compatible and let* $\mathbb{T}(\!|\mathring{E}|\!) = \varepsilon$. *Let*

- $\mathring{E} = x$**:** *Then* $\varepsilon = \overline{x}$ *and obviously* `MatchElement(`$\mathbb{T}', \mathring{E}, \varepsilon$`)` $= \{\mathbb{T}'\} = \{\mathbb{T}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{T})\}$.
- $\mathring{E} = \mathring{x}$**:** *Thus* $\varepsilon = \mathbb{T}(\!|\mathring{x}|\!) \in$ TypeTag *and* `MatchElement(`$\mathbb{T}', \mathring{E}, \varepsilon$`)` $= \{\mathbb{T}'[\mathring{x} \mapsto \varepsilon]\}$ *because* $\mathbb{T}$ *and* $\mathbb{T}'$ *are compatible. Now also because* $\mathbb{T}$ *and* $\mathbb{T}'$ *are compatible and because* $\mathrm{var}(\mathring{E}) = \{\mathring{x}\}$ *we have that* $\mathbb{T}'[\mathring{x} \mapsto \varepsilon] = \mathbb{T}' \cup \{\mathring{x} \mapsto \varepsilon\} = \mathbb{T}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{T})$.
- $\mathring{E} = (\mathring{x}_1, \ldots, \mathring{x}_k)$**:** *Let* $\iota_j = \mathbb{T}(\!|\mathring{x}_j|\!)$ *for* $j \in \{1, \ldots, k\}$. *Thus* $\varepsilon = (\iota_1, \ldots, \iota_k)$ *and* `MatchElement(`$\mathbb{T}', \mathring{E}, \varepsilon$`)` $= \{\mathbb{T}'[\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k]\}$ *because* $\mathbb{T}$ *and* $\mathbb{T}'$ *are compatible and thus* $\mathring{x}_j \in \mathrm{dom}(\mathbb{T}')$ *implies* $\mathbb{T}'(\!|\mathring{x}_j|\!) = \iota_j$. *From the same reason and because* $\mathrm{var}(\mathring{E}) = \{\mathring{x}_1, \ldots, \mathring{x}_k\}$ *we obtain that* $\mathbb{T}'[\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k] = \mathbb{T}' \cup \{\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k\} = \mathbb{T}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{T})$.
- $\mathring{E} = \texttt{<}\mathring{m}_1, \ldots, \mathring{m}_k\texttt{>}$**:** *Let* $\mu_j = \mathbb{T}(\!|\mathring{m}_j|\!)$ *for* $j \in \{1, \ldots, k\}$. *Thus* $\varepsilon = \texttt{<}\mu_1, \ldots, \mu_k\texttt{>}$ *and* `MatchElement(`$\mathbb{T}', \mathring{E}, \varepsilon$`)` $= \{\mathbb{T}'[\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k]\}$. *Because* $\mathbb{T}$ *and* $\mathbb{T}'$ *are compatible, and because* $\mathrm{var}(\mathring{E}) = \{\mathring{m}_1, \ldots, \mathring{m}_k\}$ *we obtain that* $\mathbb{T}'[\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k] = \mathbb{T}' \cup \{\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k\} = \mathbb{T}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{T})$. ∎

The completeness of `MatchElement` says that the set returned by the algorithm `MatchElement(`$\emptyset$`,`$\mathring{E}$`,`ε`)` contains only type instantiations $\mathbb{m}$ such that $\mathbb{m}(\!|\mathring{E}|\!) = \varepsilon$. We know that ε and $\mathring{E}$ uniquely determines $\mathbb{m}$ such that $\mathbb{m}(\!|\mathring{E}|\!) = \varepsilon$ (with $\mathrm{var}(\mathring{E}) = \mathrm{dom}(\mathbb{m})$) and thus the set returned by the algorithm never has more than one member. The formulation in the following lemma is again slightly more general as it considers nonempty accumulators $\mathbb{m}'$. Recall that well lhs-formed $\mathring{E}$ is an element template that appears in some well formed lhs-template and thus the same message variable can not appear more than $\mathring{E}$ once in $\mathring{E}$.

LEMMA 12.8.3 (`MatchElement` COMPLETENESS). *Let $\mathring{E}$ be a well lhs-formed element template. Let* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{E}) \subseteq \mathrm{NameVar}$. *When* $\mathbb{m} \in$ `MatchElement(`$\mathbb{m}'$`,`$\mathring{E}$`,`ε`)` *then* $\mathbb{m}(\!|\mathring{E}|\!) = \varepsilon$ *and* $\mathbb{m} = \mathbb{m}' \cup (\mathrm{var}(\mathring{E}) \lhd \mathbb{m})$.

PROOF. *Let $\mathring{E}$ be well lhs-formed and let* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{E}) \subseteq \mathrm{NameVar}$. *Let* $\mathbb{m} \in$ `MatchElement(`$\mathbb{m}'$`,`$\mathring{E}$`,`ε`)`. *Clearly* `MatchElement(`$\mathbb{m}'$`,`$\mathring{E}$`,`ε`)` $= \{\mathbb{m}\}$. *Let*

$\mathring{E} = x$**:** *The* **if** *condition at line 2 was satisfied during the execution and thus* $\mathbb{m} = \mathbb{m}'$. *Now* $\mathrm{var}(\mathring{E}) = \emptyset$ *implies the claim.*

$\mathring{E} = \mathring{x}$**:** *The* **if** *condition at line 4 was satisfied during the execution and thus* $\mathbb{m} = \mathbb{m}'[\mathring{x} \mapsto \varepsilon] = \mathbb{m}' \cup \{\mathring{x} \mapsto \varepsilon\}$. *Now claim holds because* $\mathrm{var}(\mathring{E}) = \{\mathring{x}\}$.

$\mathring{E} =$ `(`$\mathring{x}_1, \ldots, \mathring{x}_k$`)`**:** *Both relevant* **if** *conditions were satisfied and thus* $\mathbb{m} = \mathbb{m}'[\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k]$ *for some ι_1, ..., ι_k. We know that $\mathring{x}_i \neq \mathring{x}_j$ whenever $i \neq j$ because $\mathring{E}$ is well lhs-formed. The* **if** *condition at line 7 was satisfied and thus* $\varepsilon =$ `(`$\iota_1, \ldots, \iota_k$`)`. *This gives us* $\mathbb{m}(\!|\mathring{E}|\!) = \varepsilon$. *Moreover the condition at line 8 was satisfied and thus* $\mathbb{m} = \mathbb{m}' \cup \{\mathring{x}_1 \mapsto \iota_1, \ldots, \mathring{x}_k \mapsto \iota_k\}$. *Now* $\mathrm{var}(\mathring{E}) = \{\mathring{x}_1, \ldots, \mathring{x}_k\}$ *implies the claim.*

$\mathring{E} =$ `<`$\mathring{m}_1, \ldots, \mathring{m}_k$`>`**:** *The* **if** *condition at line 11 was satisfied and thus* $\mathbb{m} = \mathbb{m}'[\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k]$ *for some μ_1, ..., μ_k. We know that $\mathring{m}_i \neq \mathring{m}_j$ whenever $i \neq j$ because $\mathring{E}$ is well formed. The* **if** *condition at line 11 was satisfied and thus* $\varepsilon =$ `<`$\mu_1, \ldots, \mu_k$`>`. *This gives us* $\mathbb{m}(\!|\mathring{E}|\!) = \varepsilon$. *Moreover the assumption that* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{E}) \subseteq \mathrm{NameVar}$ *implies that* $\mathring{m}_i \notin \mathrm{dom}(\mathbb{m}')$ *for all* $i \in \{1, \ldots, k\}$ *and thus* $\mathbb{m} = \mathbb{m}' \cup \{\mathring{m}_1 \mapsto \mu_1, \ldots, \mathring{m}_k \mapsto \mu_k\}$. *Now* $\mathrm{var}(\mathring{E}) = \{\mathring{m}_1, \ldots, \mathring{m}_k\}$ *implies the claim.* ∎

12.8.2 Properties of `MatchForm`

The correctness of `MatchForm` is similar to the correctness of `MatchElement`. It says that `MatchForm(`$\emptyset$`,`$\mathring{F}$`,`φ`)` returns the set containing all instantiations $\mathbb{m}$ such that $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$ (and $\mathrm{var}(\mathring{E}) = \mathrm{dom}(\mathbb{m})$). The following slightly more general formulation considers nonempty accumulators $\mathbb{m}'$.

LEMMA 12.8.4 (`MatchForm` CORRECTNESS). *Let* $\mathbb{m}$ *and* $\mathbb{m}'$ *be compatible. Then* $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$ *implies*

$$\texttt{MatchForm}(\mathbb{m}', \mathring{F}, \varphi) = \{\mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m})\}.$$

PROOF. *Let* $\mathbb{m}$ *and* $\mathbb{m}'$ *be compatible and* $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$. *It is clear that* $\mathring{F} = \mathring{E}_0 \ldots \mathring{E}_k$ *and* $\varphi = \varepsilon_0 \ldots \varepsilon_k$ *for some* k. *We have that* $\mathbb{m}(\!|\mathring{E}_i|\!) = \varepsilon_i$ *for* $i \in \{0, \ldots, k\}$. *Now we can construct two finite sequences* $\mathbb{m}_0$, ..., $\mathbb{m}_k$ *and* $\mathbb{m}'_0$, ..., $\mathbb{m}'_k$ *of type instantiations such that* $\mathbb{m}'_0 = \mathbb{m}'$, *and that* $\mathbb{m}_i$ *and* $\mathbb{m}'_i$ *are compatible, and it holds that*

$$\{\mathbb{m}_i\} = \texttt{MatchElement}(\mathbb{m}'_i, \mathring{E}_i, \varepsilon_i) \qquad \mathbb{m}_i = \mathbb{m}'_i \cup (\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}_i) \qquad \mathbb{m}'_i = \mathbb{m}_{i-1}$$

The existence of the above sequences is easily proved by induction on i *using the above Lemma 12.8.2.*

Now let us consider the execution of the algorithm `MatchForm`$(\mathbb{m}', \mathring{F}, \varphi)$. *We can see that* $\mathbb{m}'_i$ *is the value of variable* $\mathbb{m}_0$ *at the time of evaluation of line 6 in the* i*-th iteration of the* **for** *cycle, that is, when the value of variable* i *is* i. *Moreover we can see the* **if** *condition at line 7 is satisfied in all iterations of the* **for** *cycle and that* $\mathbb{m}_i$ *is the value assigned to the existentially quantified variable* $\mathbb{m}_1$ *in the condition in the* i*-th iteration of the cycle. Hence the algorithm returns the singleton set* $\{\mathbb{m}_k\}$. *It is easy to prove that* $\mathbb{m}_i$ *and* $\mathbb{m}$ *are compatible for all* $i \in \{0, \ldots, k\}$. *Thus we can see that* $\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}_i = \mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}$. *Using the above equations we obtain*

$$\begin{aligned}\mathbb{m}_k = \mathbb{m}' \cup (\mathrm{var}(\mathring{E}_0) \lhd \mathbb{m}_0) \cup \cdots \cup (\mathrm{var}(\mathring{E}_k) \lhd \mathbb{m}_k) = \\ \mathbb{m}' \cup (\mathrm{var}(\mathring{E}_k) \lhd \mathbb{m}) \cup \cdots \cup (\mathrm{var}(\mathring{E}_0) \lhd \mathbb{m}) = \mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m})\end{aligned}$$

The last equation holds because $\mathrm{var}(\mathring{F}) = \mathrm{var}(\mathring{E}_0) \cup \cdots \cup \mathrm{var}(\mathring{E}_k)$ *and because* $\mathbb{m}$ *and* $\mathbb{m}'$ *are compatible. Hence the claim.* ∎

The completeness of `MatchForm` is similar to the completeness of `MatchElement`. It says that `MatchForm`$(\emptyset, \mathring{F}, \varphi)$ returns the set containing only instantiations $\mathbb{m}$ such that $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$ and no other instantiations. The following slightly more general formulation considers nonempty accumulators $\mathbb{m}'$.

LEMMA 12.8.5 (`MatchForm` COMPLETENESS). *Let* $\mathring{F}$ *be a well lhs-formed form template. Let* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{F}) \subseteq \mathrm{NameVar}$. *When* $\mathbb{m} \in$ `MatchForm`$(\mathbb{m}', \mathring{F}, \varphi)$ *then* $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$ *and* $\mathbb{m} = \mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m})$.

PROOF. *Let* $\mathring{F}$ *be well formed and let* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{F}) \subseteq \mathrm{NameVar}$. *Let* $\mathbb{m} \in$ `MatchForm`$(\mathbb{m}', \mathring{F}, \varphi)$. *Obviously* `MatchForm`$(\mathbb{m}', \mathring{F}, \varphi) = \{\mathbb{m}\}$ *and it implies that* $\mathring{F} = \mathring{E}_0 \ldots \mathring{E}_k$ *and* $\varphi = \varepsilon_0 \ldots \varepsilon_k$ *for some* k *(which is equal to the value of variable* k*). Moreover is implies that the* **if** *condition at line 7 was satisfied in all iterations of the* **for** *cycle in* `MatchForm`. *Thus we can construct the sequence of type instantiations*

$\mathbb{m}'_0, \ldots, \mathbb{m}'_k$ *such that* $\mathbb{m}_i$ *is the value of variable* $\mathbb{m}_0$ *at the time of evaluation of line 6 in the i-th iteration of the* **for** *cycle, that is, in the iteration when the value of variable i is i. Similarly we construct the sequence* $\mathbb{m}_0, \ldots, \mathbb{m}_k$ *such that* $\mathbb{m}_i$ *is the value of variable* $\mathbb{m}_1$ *evaluated at line 7 in the i-th iteration of the* **for** *cycle. We see that* $\mathbb{m} = \mathbb{m}_k$ *and also that* $\mathbb{m}'_0 = \mathbb{m}'$ *and* $\mathbb{m}'_i = \mathbb{m}_{i-1}$ *for* $i \in \{1, \ldots, k\}$*. Moreover we see that* $\{\mathbb{m}_i\} = \texttt{MatchElement}(\mathbb{m}'_i, \mathring{E}_i, \varepsilon_i)$ *for* $i \in \{0, \ldots, k\}$*. Let us verify the assumptions of Lemma 12.8.3. The first assumption of Lemma 12.8.3 that* $\mathring{E}_i$ *is well formed is satisfied because* $\mathring{F}$ *is well formed. The second assumption is* $\mathrm{dom}(\mathbb{m}_i) \cap \mathrm{var}(\mathring{E}'_i) \subseteq \mathrm{NameVar}$*. We know that* $\mathbb{m}_0 = \mathbb{m}'$ *and thus the assumption is satisfied for* $i = 0$ *and the lemma can be used for* $i = 0$*. The lemma proves that* $\mathbb{m}' \subseteq \mathbb{m}_0 = \mathbb{m}'_1$*. By induction on i we can prove that* $\mathbb{m}' \subseteq \mathbb{m}_i$*. Moreover the assumption that* $\mathring{F}$ *is well formed implies that no message variable from* $\mathring{E}_i$ *is contained in the previous* $\mathring{E}_0$*, ...,* $\mathring{E}_{i-1}$*. Thus there is no message variable in both* $\mathrm{dom}(\mathbb{m}_i)$ *and* $\mathrm{var}(\mathring{E}_i)$ *because* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{E}_i) \subseteq \mathrm{NameVar}$*. So by Lemma 12.8.3 we obtain that* $\mathbb{m}_i(\!|\mathring{E}_i|\!) = \varepsilon_i$ *and* $\mathbb{m}_i = \mathbb{m}'_i \cup (\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}_i)$ *for all* $i \in \{0, \ldots, k\}$*.*

This implies that $\mathbb{m}_{i-1} \subseteq \mathbb{m}_i$ *for* $i \in \{1, \ldots, k\}$ *and also that* $\mathbb{m}_i \subseteq \mathbb{m}_k = \mathbb{m}$ *for* $i \in \{0, \ldots, k\}$ *and also more specifically that* $\mathbb{m}' \subseteq \mathbb{m}$*. Now* $\mathbb{m}_i \subseteq \mathbb{m}$ *and* $\mathbb{m}_i(\!|\mathring{E}_i|\!) = \varepsilon_i$ *implies* $\mathbb{m}(\!|\mathring{E}_i|\!) = \varepsilon_i$ *which proves the first part of the claim that* $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$*. Using the above equation that* $\mathbb{m}_i = \mathbb{m}'_i \cup (\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}_i)$ *we can prove that*

$$\mathbb{m} = \mathbb{m}_k = \mathbb{m}' \cup (\mathrm{var}(\mathring{E}_0) \lhd \mathbb{m}_0) \cup \cdots \cup (\mathrm{var}(\mathring{E}_k) \lhd \mathbb{m}_k)$$

We have already proved that $\mathbb{m}_i(\!|\mathring{E}_i|\!)$ *is defined for all* $i \in \{0, \ldots, k\}$ *and thus we know that* $\mathrm{var}(\mathring{E}_i) \subseteq \mathrm{dom}(\mathbb{m}_i)$*. Thus for all* $i \in \{0, \ldots, k\}$ *we can see that* $\mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}_i = \mathrm{var}(\mathring{E}_i) \lhd \mathbb{m}$ *because* $\mathbb{m}_i \subseteq \mathbb{m}$*. Thus* $\mathbb{m} = \mathbb{m}' \cup \mathrm{var}(\mathring{F}) \lhd \mathbb{m}$ *because* $\mathrm{var}(\mathring{F}) = \mathrm{var}(\mathring{E}_0) \cup \cdots \cup \mathrm{var}(\mathring{E}_k)$ *and* $\mathbb{m}' \subseteq \mathbb{m}$*. Hence the claim.* ∎

12.8.3 Properties of `LeftMatches`

The correctness of `LeftMatches` is similar to the correctness of its above two subroutines. It says that `LeftMatches`$(\emptyset,\mathring{P},\Gamma,\chi)$ computes the set which contains all instantiations $\mathbb{m}$ such that $\mathbb{m} \models_{\mathrm{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ (and $\mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})$). Contrary to the previous algorithms, the resulting set can have more than one member because $\mathring{P}$, Γ, and χ do not uniquely determine $\mathbb{m}$. The following slightly more general formulation considers a nonempty accumulator $\mathbb{m}'$.

Lemma 12.8.6 (`LeftMatches` Correctness). *Let* $\mathbb{m}$ *and* $\mathbb{m}'$ *be compatible. Then* $\mathbb{m} \models_{\mathrm{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *implies*

$$(\mathrm{var}(\mathring{P}) \lhd \mathbb{m}) \cup \mathbb{m}' \in \texttt{LeftMatches}(\mathbb{m}', \mathring{P}, \Gamma, \chi)\,.$$

PROOF. *By induction on the structure of* $\mathring{P}$. *Let* $\mathbb{T}$ *and* $\mathbb{T}'$ *be compatible and* $\mathbb{T} \models_{\text{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. *Let us take the set of type instantiations* $\mathbb{I} = \texttt{LeftMatches}(\mathbb{T}', \mathring{P}, \Gamma, \chi)$. *We need to prove that* $(\text{var}(\mathring{P}) \lhd \mathbb{T}) \cup \mathbb{T}' \in \mathbb{I}$. *Let*

$\mathring{P} = \texttt{0}$**:** *It is cleat that* $\mathbb{I} = \{\mathbb{T}'\}$. *Now* $(\text{var}(\mathring{P}) \lhd \mathbb{T}) \cup \mathbb{T}' = \mathbb{T}'$ *because* $\text{var}(\mathring{P}) = \emptyset$.

$\mathring{P} = \mathring{p}$**:** *We see that* $\mathbb{I} = \{\mathbb{T}'[\mathring{p} \mapsto \chi]\}$. *Now* $\mathbb{T} \models_{\text{L}} \mathring{p} : \langle \Gamma, \chi \rangle$ *implies that* $\mathbb{T}(\mathring{p}) = \chi$ *and thus* $(\{\mathring{p}\} \lhd \mathbb{T}) = \{\mathring{p} \mapsto \chi\}$. *Also* $\text{var}(\mathring{P}) = \{\mathring{p}\}$ *and thus* $(\text{var}(\mathring{P}) \lhd \mathbb{T}) \cup \mathbb{T}' = \mathbb{T}'[\mathring{p} \mapsto \chi] \in \mathbb{I}$.

$\mathring{P} = \mathring{F}.\mathring{P}_0$**:** *In this case we know that there is some* χ_0 *such that* $\mathbb{T} \models_{\text{L}} \mathring{P}_0 : \langle \Gamma, \chi_0 \rangle$ *and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *where* $\varphi = \mathbb{T}(\!|\mathring{F}|\!)$. *Moreover we can see that*

$$\{\mathbb{T}_1 : \mathbb{T}_1 \in \texttt{LeftMatches}(\mathbb{T}_0, \mathring{P}_1, \Gamma, \chi_0) \;\&\; \mathbb{T}_0 \in \texttt{MatchForm}(\mathbb{T}', \mathring{F}, \varphi)\} \subseteq \mathbb{I}$$

By Lemma 12.8.4 we obtain that

$$(\text{var}(\mathring{F}) \lhd \mathbb{T}) \cup \mathbb{T}' \in \texttt{MatchForm}(\mathbb{T}', \mathring{F}, \varphi).$$

Let $\mathbb{T}'_0 = (\text{var}(\mathring{F}) \lhd \mathbb{T}) \cup \mathbb{T}'$. *It is easy to see that* $\mathbb{T}$ *and* $\mathbb{T}'_0$ *are compatible. Thus by the induction hypothesis for* $\mathbb{T}$, $\mathbb{T}'_0$, *and* $\mathring{P}_0$ *we obtain that*

$$(\text{var}(\mathring{F}) \lhd \mathbb{T}) \cup \mathbb{T}'_0 \in \texttt{LeftMatches}(\mathbb{T}'_0, \mathring{P}_0, \Gamma, \chi_0) \subseteq \mathbb{I}$$

Finally, we know that $\text{var}(\mathring{P}) = \text{var}(\mathring{F}) \cup \text{var}(\mathring{P}_0)$ *and thus*

$$(\text{var}(\mathring{F}) \lhd \mathbb{T}) \cup \mathbb{T}'_0 = (\text{var}(\mathring{F}) \lhd \mathbb{T}) \cup (\text{var}(\mathring{P}_0) \lhd \mathbb{T}) \cup \mathbb{T}' = (\text{var}(\mathring{P}) \lhd \mathbb{T}) \cup \mathbb{T}'$$

$\mathring{P} = \mathring{P}_0 \mid \mathring{P}_1$**:** *We have that* $\mathbb{T} \models_{\text{L}} \mathring{P}_0 : \langle \Gamma, \chi \rangle$ *and* $\mathbb{T} \models_{\text{L}} \mathring{P}_1 : \langle \Gamma, \chi \rangle$. *We can see that*

$$\mathbb{I} = \{\mathbb{T}_0 : \mathbb{T}_0 \in \texttt{LeftMatches}(\mathbb{T}_1, \mathring{P}_1, \Gamma, \chi) \;\&\; \mathbb{T}_1 \in \texttt{LeftMatches}(\mathbb{T}', \mathring{P}_0, \Gamma, \chi)\}$$

Let $\mathbb{T}'_1 = (\text{var}(\mathring{P}_0) \lhd \mathbb{T}) \cup \mathbb{T}'$. *By the induction hypothesis for* $\mathbb{T}$, $\mathbb{T}'$, *and* $\mathring{P}_0$ *we obtain that* $\mathbb{T}'_1 \in \texttt{LeftMatches}(\mathbb{T}', \mathring{P}_0, \Gamma, \chi)$. *It is easy to see that* $\mathbb{T}$ *and* $\mathbb{T}_1$ *are compatible. Thus by the induction hypothesis for* $\mathbb{T}$, $\mathbb{T}'_1$, *and* $\mathring{P}_1$ *we obtain that*

$$(\text{var}(\mathring{P}_1) \lhd \mathbb{T}) \cup \mathbb{T}'_1 \in \texttt{LeftMatches}(\mathbb{T}'_1, \mathring{P}_1, \Gamma, \chi) \subseteq \mathbb{I}$$

Finally, we know that $\text{var}(\mathring{P}) = \text{var}(\mathring{P}_0) \cup \text{var}(\mathring{P}_1)$ *and thus*

$$(\text{var}(\mathring{P}_1) \lhd \mathbb{T}) \cup \mathbb{T}'_1 = (\text{var}(\mathring{P}_1) \lhd \mathbb{T}) \cup (\text{var}(\mathring{P}_0) \lhd \mathbb{T}) \cup \mathbb{T}' = (\text{var}(\mathring{P}) \lhd \mathbb{T}) \cup \mathbb{T}'$$

otherwise: $\mathbb{T} \models_{\text{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *implies that* $\mathring{P}$ *can not contain a substitution application template and thus the above cases cover all possibilities.* ∎

The completeness of `LeftMatches` is similar to the completeness of its above two subroutines. It says that `LeftMatches(`$\emptyset$,$\mathring{P}$,Γ,χ`)` computes the set which contains only instantiations $\mathbb{m}$ such that $\mathbb{m} \models_{\text{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ (and $\mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})$). The following slightly more general formulation considers a nonempty accumulator $\mathbb{m}'$.

LEMMA 12.8.7 (`LeftMatches` COMPLETENESS). *Let $\mathring{P}$ be a well formed lhs-template and* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{P}) \subseteq \mathrm{NameVar}$. *When* $\mathbb{m} \in$ `LeftMatches(`$\mathbb{m}', \mathring{P}, \Gamma, \chi$`)` *then* $\mathbb{m} \models_{\text{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *and* $\mathbb{m} = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}) \lhd \mathbb{m})$.

PROOF. *Let $\mathring{P}$ be a well formed lhs-template and* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{P}) \subseteq \mathrm{NameVar}$. *Let* $\mathbb{I} =$ `LeftMatches(`$\mathbb{m}', \mathring{P}, \Gamma, \chi$`)` *and* $\mathbb{m} \in \mathbb{I}$. *Let* $\Pi = \langle \Gamma, \chi \rangle$. *Let*

$\mathring{P} =$ `0`**:** *Then* $\mathbb{I} = \{\mathbb{m}'\}$ *and thus* $\mathbb{m} = \mathbb{m}'$. *Also* $\mathrm{var}(\mathring{P}) = \emptyset$. *Hence the claim.*

$\mathring{P} = \mathring{x}$**:** *Here* $\mathbb{m} = \mathbb{m}'[\mathring{x} \mapsto \chi]$ *and* $\mathbb{I} = \{\mathbb{m}\}$. *Obviously* $\mathbb{m}(\!|\mathring{x}|\!) = \chi$ *and thus* $\mathrm{var}(\mathring{P}) \lhd \mathbb{m} = \{\mathring{x} \mapsto \chi\}$. *Also* $\mathring{p} \notin \mathrm{dom}(\mathbb{m}')$ *because* $\mathring{p} \in \mathrm{var}(\mathring{P})$ *and* $\mathring{p} \notin \mathrm{NameVar}$. *Thus* $\mathbb{m}'[\mathring{p} \mapsto \chi] = \mathbb{m}' \cup \{\mathring{p} \mapsto \chi\} = \mathbb{m}' \cup \mathrm{var}(\mathring{P}) \lhd \mathbb{m}$. *Hence the claim.*

$\mathring{P} = \mathring{F}.\mathring{P}_0$**:** *We can see that*

$$\mathbb{I} = \{\mathbb{m}_1 : \mathbb{m}_1 \in \texttt{LeftMatches}(\mathbb{m}_0, \mathring{P}_0, \Gamma, \chi_0)\ \&\ \mathbb{m}_0 \in \texttt{MatchForm}(\mathbb{m}', \mathring{F}, \varphi)\ \&\ (\chi \xrightarrow{\varphi} \chi_0) \in \Gamma\}$$

Now $\mathbb{m} \in \mathbb{I}$ *and thus there are some* $\mathbb{m}_0$, χ_0, *and* φ *such that it holds that* $\mathbb{m} \in$ `LeftMatches(`$\mathbb{m}_0, \mathring{P}_0, \Gamma, \chi_0$`)` *and* $\mathbb{m}_0 \in$ `MatchForm(`$\mathbb{m}', \mathring{F}, \varphi$`)` *and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$. *Clearly $\mathring{F}$ is well formed because $\mathring{P}$ is a well formed lhs-template and also* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{F}) \subseteq \mathrm{NameVar}$. *Thus by Lemma 12.8.5 we obtain that* $\mathbb{m}_0(\!|\mathring{F}|\!) = \varphi$ *and* $\mathbb{m}_0 = \mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m}_0)$. *The first statement implies that* $\mathrm{var}(\mathring{F}) \subseteq \mathrm{dom}(\mathbb{m}_0)$.

For any $\mathring{m} \in \mathrm{var}(\mathring{P}_0)$ *we have that* $\mathring{m} \notin \mathrm{dom}(\mathbb{m}_0)$ *because* $\mathring{m} \notin \mathrm{var}(\mathring{F})$ *and* $\mathring{m} \in \mathrm{var}(\mathring{P})$ *and* $\mathring{m} \notin \mathrm{dom}(\mathbb{m}')$ *and* $\mathring{m} \notin \mathrm{NameVar}$. *It is even more clear for any process variable because an action template can not contain a process variable. Thus* $\mathrm{dom}(\mathbb{m}_0) \cap \mathrm{var}(\mathring{P}_0) \subseteq \mathrm{NameVar}$. *Clearly $\mathring{P}_0$ is a well formed lhs-template and thus by the induction hypothesis for $\mathbb{m}_0$ and $\mathring{P}_0$ we obtain that* $\mathbb{m} \models_{\text{L}} \mathring{P}_0 : \langle \Gamma, \chi_0 \rangle$ *and* $\mathbb{m} = \mathbb{m}_0 \cup (\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m})$. *The second statement implies that* $\mathbb{m}_0 \subseteq \mathbb{m}$.

Clearly $\mathbb{m}_0 \subseteq \mathbb{m}$ *and* $\mathbb{m}_0(\!|\mathring{F}|\!) = \varphi$ *implies that* $\mathbb{m}(\!|\mathring{F}|\!) = \varphi$. *This proves* $\mathbb{m} \models_{\text{L}} \mathring{P} : \Pi$. *Moreover* $\mathbb{m}_0 \subseteq \mathbb{m}$ *and* $\mathrm{var}(\mathring{F}) \subseteq \mathrm{dom}(\mathbb{m}_0)$ *implies that* $\mathrm{var}(\mathring{F}) \lhd \mathbb{m}_0 = \mathrm{var}(\mathring{F}) \lhd \mathbb{m}$ *and thus* $\mathbb{m}_0 = \mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m})$. *It gives us that* $\mathbb{m} = \mathbb{m}' \cup (\mathrm{var}(\mathring{F}) \lhd \mathbb{m}) \cup (\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m}) = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}) \lhd \mathbb{m})$ *because* $\mathrm{var}(\mathring{P}) = \mathrm{var}(\mathring{F}) \cup \mathrm{var}(\mathring{P}_0)$. *Hence the claim.*

$\mathring{P} = \mathring{P}_0$ `|` $\mathring{P}_1$**:** *We can see that*

$$\mathbb{I} = \{\mathbb{m}_1 : \mathbb{m}_1 \in \texttt{LeftMatches}(\mathbb{m}_0, \mathring{P}_1, \Gamma, \chi)\ \&\ \mathbb{m}_0 \in \texttt{LeftMatches}(\mathbb{m}', \mathring{P}_0, \Gamma, \chi)\}$$

Now $\mathbb{m} \in \mathbb{I}$ *and thus there is some* $\mathbb{m}_0 \in$ `LeftMatches`$(\mathbb{m}', \mathring{P}_0, \Gamma, \chi)$ *such that* $\mathbb{m} \in$ `LeftMatches`$(\mathbb{m}_0, \mathring{P}_1, \Gamma, \chi)$. *Clearly* $\mathring{P}_0$ *is a well formed lhs-template and* $\mathrm{dom}(\mathbb{m}') \cap \mathrm{var}(\mathring{P}_0) \subseteq \mathrm{NameVar}$ *and thus we obtain by the induction hypothesis for* $\mathbb{m}'$ *and* $\mathring{P}_0$ *that* $\mathbb{m}_0 \models_{\mathsf{L}} \mathring{P}_0 : \Pi$ *and* $\mathbb{m}_0 = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m}_0)$. *The first statement implies that* $\mathrm{var}(\mathring{P}_0) \subseteq \mathrm{dom}(\mathbb{m}_0)$.

For any $\mathring{p} \in \mathrm{var}(\mathring{P}_1)$ *we have that* $\mathring{p} \notin \mathrm{dom}(\mathbb{m}_0)$ *because* $\mathring{p} \in \mathrm{var}(\mathring{P})$ *and* $\mathring{p} \notin \mathrm{dom}(\mathbb{m}')$ *and* $\mathring{p} \notin \mathrm{var}(\mathring{P}_0)$ *and* $\mathring{p} \notin \mathrm{NameVar}$. *The same holds for any message variable* $\mathring{m} \in \mathrm{var}(\mathring{P}_1)$ *and thus* $\mathrm{dom}(\mathbb{m}_0) \cap \mathrm{var}(\mathring{P}_1) \subseteq \mathrm{NameVar}$. *Obviously* $\mathring{P}_1$ *is a well formed lhs-template and thus by the induction hypothesis for* $\mathbb{m}_0$ *and* $\mathring{P}_1$ *we obtain that* $\mathbb{m} \models_{\mathsf{L}} \mathring{P}_1 : \Pi$ *and* $\mathbb{m} = \mathbb{m}_0 \cup (\mathrm{var}(\mathring{P}_1) \lhd \mathbb{m})$. *The second statement implies that* $\mathbb{m}_0 \subseteq \mathbb{m}$.

Now $\mathbb{m}_0 \subseteq \mathbb{m}$ *and* $\mathbb{m}_0 \models_{\mathsf{L}} \mathring{P}_0 : \Pi$ *implies* $\mathbb{m} \models_{\mathsf{L}} \mathring{P}_0 : \Pi$ *by Lemma 8.6.1. This proves* $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \Pi$. *Moreover* $\mathbb{m}_0 \subseteq \mathbb{m}$ *and* $\mathrm{var}(\mathring{P}_0) \subseteq \mathrm{dom}(\mathbb{m}_0)$ *implies that* $\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m}_0 = \mathrm{var}(\mathring{P}_0) \lhd \mathbb{m}$ *and thus* $\mathbb{m}_0 = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m})$. *It gives us that* $\mathbb{m} = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}_0) \lhd \mathbb{m}) \cup (\mathrm{var}(\mathring{P}_1) \lhd \mathbb{m}) = \mathbb{m}' \cup (\mathrm{var}(\mathring{P}) \lhd \mathbb{m})$ *because* $\mathrm{var}(\mathring{P}) = \mathrm{var}(\mathring{P}_0) \cup \mathrm{var}(\mathring{P}_1)$. *Hence the claim.*

otherwise: $\mathring{P}$ *is a well formed lhs-template and thus can not contain a substitution application template and thus the above cases cover all possibilities.* ∎

`LeftMatches` is called from `LocalClosureStep` and from `ActiveNodes` always with the empty accumulator which saves the result computed so far (when a recursive call to `LeftMatches` is made). Thus the following proposition combine the above correctness and completeness properties to the property which is required to prove correctness and completeness of `LocalClosureStep` and `ActiveNodes`. It says that `LeftMatches`$(\emptyset,\mathring{P},\Gamma,\chi)$ computes the set which contains exactly the instantiations $\mathbb{m}$ such that $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ (and $\mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})$).

Proposition 12.8.8 (`LeftMatches` Correctness). *Let* $\mathring{P}$ *be a well formed lhs-template. It holds that*

$$\texttt{LeftMatches}(\emptyset, \mathring{P}, \Gamma, \chi) = \{\mathbb{m} \colon \mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle \ \&\ \mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})\}.$$

Proof. *Let* $\mathring{P}$ *be a well formed lhs-template. Let* $\mathbb{I} =$ `LeftMatches`$(\emptyset, \mathring{P}, \Gamma, \chi)$. *Let* $\mathbb{m} \in \mathbb{I}$. *Clearly* $\emptyset \cap \mathrm{var}(\mathring{P}) = \emptyset \subseteq \mathrm{NameVar}$ *and thus by Lemma 12.8.7 we obtain that* $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *and* $\mathbb{m} = \emptyset \cup (\mathrm{var}(\mathring{P}) \lhd \mathbb{m})$. *The second statement implies that* $\mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})$. *This proves the "*$\subseteq$*" direction of the equality in question.*

Now let us prove the "$\supseteq$*" direction. Let* $\mathbb{m}$ *be such that* $\mathbb{m} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *and* $\mathrm{dom}(\mathbb{m}) = \mathrm{var}(\mathring{P})$. *Let* $\mathbb{m}' = \emptyset$. *Clearly* $\mathbb{m}$ *and* $\mathbb{m}'$ *are compatible. We see that* $(\mathrm{var}(\mathring{P}) \lhd \mathbb{m}) \cup \mathbb{m}' = \mathbb{m}$ *and thus* $\mathbb{m} \in \mathbb{I}$ *by Lemma 12.8.6.* ∎

12.8.4 Properties of `RightRequired`

The correctness of $\texttt{RightRequired}(\mathbb{m}, \mathring{Q}, \Gamma, \chi)$ says that the algorithm returns the graph Γ' which contains all the edges required for $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \chi \rangle$ to hold. The returned graph Γ' can contain some edges which present in Γ.

Lemma 12.8.9 (`RightRequired` Correctness). *Let*

(1) $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{dom}(\mathbb{m})$ *and*

(2) $\mathbb{m}(\mathring{x}_i) \neq \mathbb{m}(\mathring{x}_j)$ *whenever* $\mathring{Q}$ *contains* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$ *and* $i \neq j$.

When $\Gamma' = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}, \Gamma, \chi)$ *then it* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \chi \rangle$.

Proof. *By induction on the structure of* $\mathring{Q}$.

$\mathring{Q} = \texttt{0}$**:** *Clear.*

$\mathring{Q} = \mathring{p}$**:** *Clear because* $\Gamma' = \{\mathbb{m}(\mathring{p}) \overset{\emptyset}{\dashrightarrow} \chi\}$ *and because* $\mathring{p} \in \mathrm{dom}(\mathbb{m})$.

$\mathring{Q} = \{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\,\mathring{p}$**:** *Here* $\Gamma' = \{\mathbb{m}(\mathring{p}) \overset{\mathbb{m}([\![\mathring{x}_0]\!]) \mapsto \mathbb{m}([\![\mathring{s}_0]\!]), \ldots, \mathbb{m}([\![\mathring{x}_k]\!]) \mapsto \mathbb{m}([\![\mathring{s}_k]\!])}{\dashrightarrow} \chi\}$ *contains a properly defined type substitution because* $\mathbb{m}(\mathring{x}_i) \neq \mathbb{m}(\mathring{x}_j)$ *for* $i \neq j$ *and because* $\{\mathring{x}_0, \ldots, \mathring{x}_k, \mathring{s}_0, \ldots, \mathring{s}_k, \mathring{p}\} \subseteq \mathrm{dom}(\mathbb{m})$. *Thus the claim.*

$\mathring{Q} = \mathring{F}.\mathring{Q}_0$**:** *Let* $\varphi = \mathbb{m}([\![\mathring{F}]\!])$ *(which is defined because* $\mathrm{var}(\mathring{F}) \subseteq \mathrm{dom}(\mathbb{m})$*). The value of variable* χ_0 *gives us node* χ_0 *such that such that* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma'$. *Let* $\Gamma'_0 = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}_0, \Gamma \cup \{\chi \xrightarrow{\varphi} \chi_0\}, \chi_0)$. *We see that* $\Gamma' = \Gamma'_0 \cup \{\chi \xrightarrow{\varphi} \chi_0\}$. *It is clear that* $\mathrm{var}(\mathring{Q}_0) \subseteq \mathrm{dom}(\mathbb{m})$ *and thus by the induction hypothesis we obtain that* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma'_0, \chi_0 \rangle$. *By Lemma 8.6.3 we have that* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma', \chi_0 \rangle$. *Hence* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{F}.\mathring{Q}_0 : \langle \Gamma', \chi \rangle$.

$\mathring{Q} = \mathring{Q}_0 \mid \mathring{Q}_1$**:** *Let the recursive calls result in* $\Gamma'_0 = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}_0, \Gamma, \chi)$ *and* $\Gamma'_1 = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}_1, \Gamma \cup \Gamma'_0, \chi)$. *It is clear that* $\Gamma' = \Gamma'_0 \cup \Gamma'_1$. *Obviously* $\mathrm{var}(\mathring{Q}_0) \subseteq \mathrm{dom}(\mathbb{m})$ *and* $\mathrm{var}(\mathring{Q}_1) \subseteq \mathrm{dom}(\mathbb{m})$ *and thus by the induction hypothesis we obtain that* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma'_0, \chi \rangle$ *and* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_1 : \langle \Gamma'_1, \chi \rangle$. *Thus by Lemma 8.6.3 we have that* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_0 : \langle \Gamma', \chi \rangle$ *and* $\mathbb{m} \vDash_{\mathsf{R}} \mathring{Q}_1 : \langle \Gamma', \chi \rangle$. *Hence the claim.* ∎

The completeness of `RightRequired` says that it preserves existence of nesting of the input in any restricted $\mathcal{R}$-type (where $\mathcal{R}$ is the argument of the call of `LocalClosureStep` from which `RightRequired` is called). That is, when $\Gamma_0 = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}, \Gamma, \chi)$ and there is a nesting of $\langle \Gamma, \chi \rangle$ in some restricted Π' then we can construct a nesting of $\langle \Gamma \cup \Gamma_0, \chi \rangle$ in Π'. In order words it means that Γ_0 does not contain unnecessary edges, that is, that the local closure algorithm adds only those edges which have to be added. Assumptions *(4)* & *(5)* will be satisfied for all locally $\mathcal{R}$-closed $\Pi' = \langle \Gamma', \chi'_{\mathsf{r}} \rangle$.

Lemma 12.8.10 (`RightRequired` Completeness). *Let*

(1) $\Gamma_0 = \texttt{RightRequired}(\mathbb{T}, \mathring{Q}, \Gamma, \chi)$*, and*

(2) $\delta \mathrel{\therefore} \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$*, and*

(3) let $\langle \Gamma', \chi'_{\mathsf{r}} \rangle$ *be restricted, and*

(4) $\delta \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$*, and*

(5) $\mathbb{T}' \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \delta(\chi) \rangle$*.*

Then there is δ_0 *such that* $\delta_0 \mathrel{\therefore} \langle \Gamma \cup \Gamma_0, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$ *and* $\delta_0 \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$*.*

Proof. *Let the assumptions hold. Let us prove the claim by induction on the structure of* $\mathring{Q}$*. Let*

$\mathring{Q} = \texttt{0}$**:** *Then* $\Gamma_0 = \emptyset$ *and thus we can simply take* $\delta_0 = \delta$*.*

$\mathring{Q} = \mathring{p}$**:** *We have* $\Gamma_0 = \{\mathbb{T}(\mathring{p}) \overset{\emptyset}{\dashrightarrow} \chi\}$*. An important observation here is that both the nodes* $\mathbb{T}(\mathring{p})$ *and* χ *are in* $\mathrm{dom}(\delta)$*. The node* $\mathbb{T}(\mathring{p})$ *because (4) & (5) and* χ *because* $\delta(\chi)$ *is mentioned and thus defined in (5). Now let us prove that* $\delta \mathrel{\therefore} \langle \Gamma \cup \Gamma_0, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$*. We know that* $\delta \mathrel{\therefore} \langle \Gamma, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$ *and thus we only need to prove that* Γ' *contains an edge which corresponds to the only edge in* Γ_0*. From* $\mathbb{T}' \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \delta(\chi) \rangle$ *we know that* $(\mathbb{T}'(\mathring{p}) \overset{\emptyset}{\dashrightarrow} \delta(\chi)) \in \Gamma'$*. Now* $\delta \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$ *implies that* $\mathbb{T}'(\mathring{p}) = \delta(\mathbb{T}(\mathring{p}))$ *and thus we have that* $(\delta(\mathbb{T}(\mathring{p})) \overset{\emptyset}{\dashrightarrow} \delta(\chi)) \in \Gamma'$*. Hence the claim because clearly* $\emptyset \preccurlyeq \emptyset$*.*

$\mathring{Q} = \{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\, \mathring{p}$**:** *Let* $\mathfrak{o} = \{\mathbb{T}(\mathring{x}_0) \mapsto \mathbb{T}(\mathring{s}_0), \ldots, \mathbb{T}(\mathring{x}_k) \mapsto \mathbb{T}(\mathring{s}_k)\}$*. Let* $\mathfrak{o}' = \{\mathbb{T}'(\mathring{x}_0) \mapsto \mathbb{T}'(\mathring{s}_0), \ldots, \mathbb{T}'(\mathring{x}_k) \mapsto \mathbb{T}'(\mathring{s}_k)\}$*. Firstly assumptions (4) & (5) implies that* $\mathfrak{o}$ *is a correctly defined function, that is, that* $\mathbb{T}(\mathring{x}_i) \neq \mathbb{T}(\mathring{x}_j)$ *for* $i \neq j$*. Assumption (5) alone implies that* $\mathfrak{o}$ *is a function. Assumption (4) implies that* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$*. We have* $\Gamma_0 = \{\mathbb{T}(\mathring{p}) \overset{\mathfrak{o}}{\dashrightarrow} \chi\}$*. As in the previous case, both the nodes* $\mathbb{T}(\mathring{p})$ *and* χ *are in* $\mathrm{dom}(\delta)$*. Thus again we only need to prove the existence of an edge postulated by Definition 12.1.3 for the only new edge in* Γ_0*. From* $\mathbb{T}' \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \delta(\chi) \rangle$ *we know that* $(\mathbb{T}'(\mathring{p}) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi)) \in \Gamma'$*. Clearly Now* $\delta \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$ *implies that* $\mathbb{T}'(\mathring{p}) = \delta(\mathbb{T}(\mathring{p}))$*. Hence the claim because* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$*.*

$\mathring{Q} = \mathring{F}.\mathring{Q}_1$**:** *Let* $\varphi = \mathbb{T}(\!|\mathring{F}|\!)$ *and* $\varphi' = \mathbb{T}'(\!|\mathring{F}|\!)$*. By Lemma 12.5.5 we have that* $\varphi \preccurlyeq \varphi'$*. Let* η *be the value of variable* η*. Thus we see that* $\eta = (\chi \xrightarrow{\varphi} \chi_0)$ *for some* χ_0*. Let* $\Gamma_1 = \texttt{RightRequired}(\mathbb{T}, \mathring{Q}_1, \Gamma \cup \{\eta\}, \chi_0)$*. Now we can see that* $\Gamma_0 = \{\eta\} \cup \Gamma_1$*. In order to use the induction hypothesis we need at first find some* δ_1 *such that* $\delta_1 \mathrel{\therefore} \langle \Gamma \cup \{\eta\}, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$ *and* $\delta_1 \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$*. From* $\mathbb{T}' \vDash_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \delta(\chi) \rangle$ *we obtain that there is some* χ''_0 *such that* $\mathbb{T}' \vDash_{\mathsf{R}} \mathring{Q}_1 : \langle \Gamma', \chi''_0 \rangle$ *and* $(\delta(\chi) \xrightarrow{\varphi'} \chi''_0) \in \Gamma'_0$*. We shall prove that we can take* $\delta_1 = \delta[\chi_0 \mapsto \chi''_0]$*.*

We have that $\delta_1(\chi_0) = \chi''_0$*. When* χ_0 *was chosen fresh for* Γ *then* $\chi_0 \notin \mathrm{dom}(\delta)$ *and thus it is easy to see that* $\delta_1 \mathrel{\therefore} \langle \Gamma \cup \{\eta\}, \chi_{\mathsf{r}} \rangle \trianglelefteq \langle \Gamma', \chi'_{\mathsf{r}} \rangle$ *as well as* $\delta_1 \mathrel{\therefore} \mathbb{T} \trianglelefteq \mathbb{T}'$*. Now, let us consider the case where* χ_0 *is a node already present in* Γ*. We know thus that* $\eta \in \Gamma$*. Thus there is* $(\delta(\chi) \xrightarrow{\varphi''} \delta(\chi_0)) \in \Gamma'$ *such that* $\varphi \preccurlyeq \varphi''$*.*

We also know from the above that $(\delta(\chi) \xrightarrow{\varphi'} \chi_0'') \in \Gamma_0'$ *with* $\varphi \preccurlyeq \varphi'$. *Now* $\varphi \preccurlyeq \varphi'$ *and* $\varphi \preccurlyeq \varphi''$ *implies that* $\varphi' \approx \varphi''$ *because* $[\![\varphi]\!] \neq \emptyset$. *We know that* Γ' *is restricted and thus the width restrictions requires that* $\delta(\chi_0) = \chi_0''$. *Thus we can see that in this case* $\delta_1 = \delta$ *and thus clearly* $\delta_1 \therefore \langle \Gamma \cup \{\eta\}, \chi_\mathsf{r}\rangle \trianglelefteq \langle \Gamma', \chi_\mathsf{r}'\rangle$ *because* $\Gamma \cup \{\eta\} = \Gamma$.

Thus we obtain $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q}_1 : \langle \Gamma', \delta_1(\chi_0)\rangle$ *and we can use the induction hypothesis for* Γ_1 *(as* Γ_0*),* $\mathring{Q}_1$, χ_0, $\Gamma \cup \{\eta\}$ *(as* Γ*), and* δ_1. *The induction hypothesis gives us* δ_0 *such that* $\delta_0 \therefore \langle \Gamma \cup \{\eta\} \cup \Gamma_1, \chi_\mathsf{r}\rangle \trianglelefteq \langle \Gamma', \chi_\mathsf{r}'\rangle$ *and* $\delta_0 \therefore \mathbb{m} \trianglelefteq \mathbb{m}'$. *Hence the claim because* $\Gamma_0 = \{\eta\} \cup \Gamma_1$.

$\mathring{Q} = \mathring{Q}_1 \mid \mathring{Q}_2$**:** *Let* $\Gamma_0' = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}_1, \Gamma, \chi)$ *and moreover let us take* $\Gamma_1' = \texttt{RightRequired}(\mathbb{m}, \mathring{Q}_2, \Gamma \cup \Gamma_0', \chi)$. *It is easy to see that* $\Gamma_0 = \Gamma_0' \cup \Gamma_1'$. *Now* $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q} : \langle \Gamma', \delta(\chi)\rangle$ *implies that* $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q}_1 : \langle \Gamma', \delta(\chi)\rangle$ *and* $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q}_2 : \langle \Gamma', \delta(\chi)\rangle$. *Using the induction hypothesis for* Γ_0' *and* $\mathring{Q}_1$ *we obtain* δ_1 *such that* $\delta_1 \therefore \langle \Gamma \cup \Gamma_0', \chi_\mathsf{r}\rangle \trianglelefteq \langle \Gamma', \chi_\mathsf{r}'\rangle$. *Furthermore using the induction hypothesis for* Γ_1', $\mathring{Q}_2$, $\Gamma \cup \Gamma_0'$, *and* δ_1 *we obtain that there is* $\delta_0 \therefore \langle \Gamma \cup \Gamma_0' \cup \Gamma_1', \chi_\mathsf{r}\rangle \trianglelefteq \langle \Gamma', \chi_\mathsf{r}'\rangle$. *Hence the claim because* $\Gamma_0 = \Gamma_0' \cup \Gamma_1'$. ∎

12.8.5 Properties of `ActiveNodes`

The following proposition proves the termination, correctness, and completeness of `ActiveNodes`. It says that the algorithm correctly computes the set of active nodes. The correctness of `ActiveNodes` is the "$\supseteq$" direction of the equality below while the "$\subseteq$" direction is its completeness.

Proposition 12.8.11. *Let* $\mathcal{R}$ *be finite. Then* `ActiveNodes` *terminates for every input and* $\texttt{ActiveNodes}(\Pi, \mathcal{R}) = \mathrm{ActiveNode}_\mathcal{R}(\Pi)$.

Proof. *Firstly let us observe that the following invariants are valid all the time during the execution of the algorithm. (1)* Ξ *and* Ξ_new *are set of nodes of* Γ *and thus are finite and there is an upper bound on the number of their members. (2)* Ξ *and* Ξ_new *are disjoint. (3) The size of* Ξ *is increased by one with every iteration of the* **while** *loop. Invariant (2) is useful to observe (3), and (1&3) implies that the algorithm terminates.*

From Lemma 12.8.7 it follows that the **foreach** *cycle on line 7 iterates through all type instantiations that could provide different node values* $\mathbb{m}(\mathring{p})$. *Thus the* **foreach** *cycles on lines 6 and 7 iterates through* $\mathrm{ActiveSucc}_\mathcal{R}(\Gamma, \chi_0)$. *It means that line 8 is executed for every* $\mathbb{m}(\mathring{p}) \in \mathrm{ActiveSucc}_\mathcal{R}(\Gamma, \chi_0)$. *Thus lines 6-8 can be equivalently expressed by a one line statement*

$$\Xi_\mathsf{new} := \Xi_\mathsf{new} \cup (\mathrm{ActiveSucc}_\mathcal{R}(\Gamma, \chi_0) \setminus \Xi);$$

Now we can observe that another invariant (4) is satisfied:

$$\forall \chi' \in \Xi\colon \mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi') \subseteq (\Xi \cup \Xi_{\mathsf{new}})$$

Let $\Xi' =$ `ActiveNodes`$(\Pi, \mathcal{R})$. *The algorithm terminates when* $\Xi_{\mathsf{new}} = \emptyset$ *and then returns* Ξ. *Thus from (4) we have that* $\forall \chi' \in \Xi' : \mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi') \subseteq \Xi'$. *When* $\Pi = \langle \Gamma, \chi_{\mathsf{r}} \rangle$ *we can conclude that the following recursive property is satisfied (because* χ_{r} *is added to* Ξ *in the first* **while** *loop iteration):*

$$\Xi' = \{\chi_{\mathsf{r}}\} \cup \{\chi_1 \in \mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi_0)\colon \chi_0 \in \Xi'\}$$

Now $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ *is the smallest set with the above property by Definition 7.6.7 and thus we obtain that* $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi) \subseteq \Xi'$.

Opposite wise, when $\chi' \in \Xi'$ *we can find a natural number* k *and nodes* $\chi'_0, \ldots, \chi'_k$ *such that* $\chi'_0 = \chi_{\mathsf{r}}$, *and* $\chi'_k = \chi$, *and* $\chi'_i \in \mathrm{ActiveSucc}_{\mathcal{R}}(\Gamma, \chi'_{i-1})$ *for* $0 < i \leqslant k$. *The node* χ'_i *is the value of the program variable* χ_0 *evaluated right after the execution of line 5 in the* $(i+1)$*-th iteration of the* **while** *loop in* `ActiveNodes` *(and* k *is simply the number of the iteration which added* χ' *to* Ξ'*). In other words, this node sequence provides a justification that the node* χ' *has to be a member of the smallest set* $\mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$ *of active nodes. This gives us* $\Xi' \subseteq \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. *Hence the claim.* ∎

12.8.6 Properties of `LocalClosureStep`

At first we prove the termination of `LocalClosureStep` together with the termination of all its subroutines whose termination has not been proved yet.

Lemma 12.8.12 (`LocalClosureStep` Termination). *Every call of the algorithm* `LocalClosureStep`$(\Pi,\mathcal{R})$ *terminates for every* Π *and every finite* $\mathcal{R}$.

Proof. *Let* $\mathcal{R}$ *be finite.* `ActiveNodes` *terminates by Proposition 12.8.11 and the same proposition says the it returns a finite set of nodes (because the set of active nodes is subset of all nodes which is finite). The termination of* `LeftMatches` *and its subroutines* `MatchElement` *and* `MatchForm` *is proved easily by structural induction on its template argument. Thus also* `LeftMatches` *has to return a finite set. The termination of* `RightRequired` *is proved by induction on its template argument as well. Hence every call* `LocalClosureStep`$(\Pi,\mathcal{R})$ *terminates because* $\mathcal{R}$ *is finite.* ∎

A shape predicate is well formed when none of its form types contain more than one occurrence of an input-bound type tag. For example, "(ι, ι)" can not appear in a well formed shape predicate.

DEFINITION 12.8.13. *A shape predicate* Π *is* **well formed** *when any* φ *in* Π *contains exactly one occurrence of every* $\iota \in \mathrm{itags}(\varphi)$. ∎

In the case of `LocalClosureStep`, the correctness is similar the correctness of `RestrictWidth`. It says that when the algorithm returns its input shape predicate uncharged then the input is locally $\mathcal{R}$-closed. The assumption of well formedness of the input shape predicate is used to prove assumption (2) of Lemma 12.8.9 (`RightRequired` Correctness).

LEMMA 12.8.14 (`LocalClosureStep` CORRECTNESS). *Let* $\mathcal{R}$ *be finite and let* Π *be well formed. It holds that when* $\texttt{LocalClosureStep}(\Pi, \mathcal{R}) = \Pi$ *then* Π *is locally* $\mathcal{R}$*-closed at any active node* $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$.

PROOF. *Let* $\mathcal{R}$ *be finite and let* Π *and* $\mathcal{R}$ *be well formed. Let* $\Pi = \langle \Gamma, \chi_{\mathsf{r}} \rangle$ *and* $\chi \in \mathrm{ActiveNode}_{\mathcal{R}}(\Pi)$. *Let* $\texttt{LocalClosureStep}(\Pi, \mathcal{R}) = \Pi$. *Moreover let* $\textbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *and* $\mathbb{T} \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$. *To prove the claim we need to show that* $\mathbb{T} \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$.

Let Γ_0 *be the value of variable* Γ_0 *at the time of evaluation of line 7 in the execution of* `LocalClosureStep`. *Also we see that* χ_{r} *is the value of variable* χ_{r} *during the whole execution. Thus* $\texttt{LocalClosureStep}(\Pi, \mathcal{R}) = \Pi$ *implies* $\Gamma_0 \subseteq \Gamma$. *By Proposition 12.8.11 we obtain that* $\chi \in \texttt{ActiveNodes}(\Gamma, \chi_{\mathsf{r}}, \mathcal{R}, \emptyset)$ *and thus the rewriting rule* $\textbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} \in \mathcal{R}$ *is processed by the* **for** *cycle at line 4 at the point when the value of variable* χ *is* χ.

Let $\mathbb{T}_0 = \mathrm{var}(\mathring{P}) \lhd \mathbb{T}$. *Wee see that* $\mathrm{var}(\mathring{P}) \subseteq \mathrm{dom}(\mathbb{T})$ *and thus* $\mathrm{dom}(\mathbb{T}_0) = \mathrm{var}(\mathring{P})$. *Thus* $\mathbb{T}_0 \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *by Lemma 8.6.2. Now by Proposition 12.8.8 we obtain that* $\mathbb{T}_0 \in \texttt{LeftMatches}(\emptyset, \mathring{P}, \Gamma, \chi)$. *It means that* $\mathbb{T}_0$ *is processed at some point by the* **for** *cycle at line 5. At this point the value of variable* $\mathbb{T}$ *becomes* $\mathbb{T}_0$ *and line 6 is executed. The values of variables* $\mathbb{T}$, $\mathring{Q}$, Γ, *and* χ *are in turn* $\mathbb{T}_0$, $\mathring{Q}$, Γ, *and* χ. *Let* $\Gamma' = \texttt{RightRequired}(\mathbb{T}_0, \mathring{Q}, \Gamma, \chi)$.

Let us verify the assumptions of Lemma 12.8.9 for $\mathbb{T}_0$, $\mathring{Q}$, Γ, *and* χ. *The rewriting rule is well formed and thus* $\mathrm{var}(\mathring{Q}) \subseteq \mathrm{var}(\mathring{P})$ *by rule* R2 *&* R3 *for* $\mathring{Q}$. *Thus assumption (1) follows from* $\mathrm{var}(\mathring{P}) = \mathrm{dom}(\mathbb{T}_0)$. *Let* $\mathring{Q}$ *contain* $\{\mathring{x}_0 := \mathring{s}_0, \ldots, \mathring{x}_k := \mathring{s}_k\}\mathring{p}$ *and let* $i \neq j$. *Rule* R4 *for* $\mathring{Q}$ *implies that* $\mathring{x}_i \neq \mathring{x}_j$. *By Lemma 6.3.1 we have that there is some* $\mathring{F}$ *in* $\mathring{P}$ *such that* $\{\mathring{x}_0, \ldots, \mathring{x}_k\} \subseteq \mathrm{bv}(\mathring{F})$. *Now* $\mathbb{T}_0 \models_{\mathsf{L}} \mathring{P} : \langle \Gamma, \chi \rangle$ *implies that* $\mathbb{T}_0[\![\mathring{F}]\!] \in \Gamma$. *It implies that* $\mathbb{T}_0[\![\mathring{x}_i]\!] \neq \mathbb{T}_0[\![\mathring{x}_j]\!]$ *because* Γ *is well formed. Thus assumption (2) is satisfied and by Lemma 12.8.9 we obtain that* $\mathbb{T}_0 \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma', \chi \rangle$. *Clearly* $\Gamma' \subseteq \Gamma_0 \subseteq \Gamma$ *and thus by Lemma 8.6.3 it holds that* $\mathbb{T}_0 \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$. *Finally by Lemma 8.6.1 we prove the claim* $\mathbb{T} \models_{\mathsf{R}} \mathring{Q} : \langle \Gamma, \chi \rangle$. ∎

The completeness of `LocalClosureStep` says that it preserves existence of a nesting of the input shape predicate in any restricted $\mathcal{R}$-type.

LEMMA 12.8.15 (`LocalClosureStep` COMPLETENESS). *Let $\mathcal{R}$ be well formed and finite and let Π' be locally $\mathcal{R}$-closed and restricted. Let $\delta \mathrel{:}\!\!\cdot \Pi \trianglelefteq \Pi'$ and let $\Pi_1 =$ `LocalClosureStep`$(\Pi, \mathcal{R})$. Then there is δ_1 such that $\delta_1 \mathrel{:}\!\!\cdot \Pi_1 \trianglelefteq \Pi'$.*

PROOF. *Let $\mathcal{R}$ be well-formed and finite, and let Π' be locally $\mathcal{R}$-closed and restricted. Let $\delta \mathrel{:}\!\!\cdot \Pi \trianglelefteq \Pi'$ and let $\Pi_1 =$ `LocalClosureStep`$(\Pi, \mathcal{R})$. Let $\Pi = \langle \Gamma, \chi_\mathsf{r} \rangle$ and $\Pi' = \langle \Gamma', \chi'_\mathsf{r} \rangle$. We know that* `LocalClosureStep` *terminates by Lemma 12.8.12 because $\mathcal{R}$ is finite. Thus line 6 is evaluated only finitely many times during the execution of the* `LocalClosureStep`$(\Pi, \mathcal{R})$. *Let us consider that the line was evaluated k-times. Let Γ_i be the value of variable Γ_0 after the i-th evaluation of line 6. This gives a finite sequence of graphs $\Gamma_1, \ldots, \Gamma_k$. Let us take $\Gamma_0 = \emptyset$. We can see that $\Pi_1 = \langle \Gamma \cup \Gamma_k, \chi_\mathsf{r} \rangle$.*

Let us prove by induction on i that there is δ'_i such that $\delta'_i \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_i, \chi_\mathsf{r} \rangle \trianglelefteq \Pi'$. Clearly, for $i = 0$ we can take $\delta'_0 = \delta$ because $\Gamma_0 = \emptyset$. Now let $\delta'_i \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_i, \chi_\mathsf{r} \rangle \trianglelefteq \Pi'$ for $i < k$. We want to prove that there is δ'_{i+1} such that $\delta'_{i+1} \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_{i+1}, \chi_\mathsf{r} \rangle \trianglelefteq \Pi'$. Let χ, $\mathring{P}$, $\mathring{Q}$, and $\mathbb{m}$ be the values of the correspondingly named variables at the time of the $(i+1)$-th evaluation of line 6 during the execution of the algorithm. We see that $\chi \in$ `ActiveNodes`$(\Gamma, \chi_\mathsf{r}, \mathcal{R}, \emptyset)$ and **rewrite**`{` $\mathring{P} \hookrightarrow \mathring{Q}$ `}` $\in \mathcal{R}$ *and that $\mathbb{m} \in$ `LeftMatches`$(\emptyset, \mathring{P}, \Gamma, \chi)$. From the first and the third claim we obtain in turn by Proposition 12.8.11 and Proposition 12.8.8 that $\chi \in \mathrm{ActiveNode}_\mathcal{R}(\Pi)$ and $\mathbb{m} \vDash_\mathsf{L} \mathring{P} : \langle \Gamma, \chi \rangle$. Now the induction hypothesis says that $\delta'_i \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_i, \chi_\mathsf{r} \rangle \trianglelefteq \langle \Gamma', \chi'_\mathsf{r} \rangle$. Thus from $\mathbb{m} \vDash_\mathsf{L} \mathring{P} : \langle \Gamma, \chi \rangle$ by Lemma 12.5.8 we obtain that there is some $\mathbb{m}'$ such that $\delta'_i \mathrel{:}\!\!\cdot \mathbb{m} \trianglelefteq \mathbb{m}'$ and $\mathbb{m}' \vDash_\mathsf{L} \mathring{P} : \langle \Gamma', \delta'_i(\chi) \rangle$. Furthermore, χ is a node in Γ and thus $\chi \in \mathrm{dom}(\delta'_i)$. Thus by Lemma 12.5.9 we obtain that $\delta'_i(\chi) \in \mathrm{ActiveNode}_\mathcal{R}(\Pi')$. We know that Π' is locally $\mathcal{R}$-closed and thus $\mathbb{m}' \vDash_\mathsf{L} \mathring{P} : \langle \Gamma', \delta'_i(\chi) \rangle$ implies $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q} : \langle \Gamma', \delta'_i(\chi) \rangle$.*

Let $\Gamma'_{i+1} =$ `RightRequired`$(\mathbb{m}, \mathring{Q}, \Gamma, \chi)$. We see that $\Gamma_{i+1} = \Gamma_i \cup \Gamma'_{i+1}$. We have already showed above that $\delta'_i \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_i, \chi_\mathsf{r} \rangle \trianglelefteq \Pi'$ and $\delta'_i \mathrel{:}\!\!\cdot \mathbb{m} \trianglelefteq \mathbb{m}'$ and $\mathbb{m}' \vDash_\mathsf{R} \mathring{Q} : \langle \Gamma', \delta'_i(\chi) \rangle$. Thus by Lemma 12.8.10 we obtain that there is some δ_{i+1} such that $\delta_{i+1} \mathrel{:}\!\!\cdot \langle \Gamma \cup \Gamma_i \cup \Gamma'_{i+1}, \chi_\mathsf{r} \rangle \trianglelefteq \Pi'$. Hence the claim because $\Gamma_i \cup \Gamma'_{i+1} = \Gamma_{i+1}$. ∎

12.9 Properties of the Flow Closure Algorithm

The following lemma helps to prove the completeness of `PrincipalType`. It says then when the condition "$\Pi = \Pi_0$" of the **repeat** cycle in `PrincipalType` is valid (and thus the algorithm terminates) then both the last calls to the subroutines `LocalClosureStep` and `FlowClosureStep` returned its argument unchanged.

LEMMA 12.9.1. *When* `FlowClosureStep(LocalClosureStep`$(\Pi, \mathcal{R})$`)` $= \Pi$ *then it holds that* `LocalClosureStep`$(\Pi, \mathcal{R}) = \Pi$.

PROOF. *The property in question follows from the fact that both* `FlowClosureStep` *and* `LocalClosureStep` *preserves the root node and both algorithms can only add edges to an input shape predicate.* ∎

The following proves the correctness of `FlowClosureStep`.

LEMMA 12.9.2 (`LocalClosureStep` CORRECTNESS). `FlowClosureStep`$(\Pi) = \Pi$ *implies that* Π *is flow-closed.*

PROOF. *Let* `FlowClosureStep`$(\Pi) = \Pi$. *Let* $\Pi = \langle \Gamma, \chi_\mathsf{r} \rangle$. *Let* Γ_0 *be the value of variable* Γ_0 *at the point when the last line 13 of* `FlowClosureStep` *is being evaluated. From* `FlowClosureStep`$(\Pi) = \Pi$ *is follows that* $\Gamma_0 \subseteq \Gamma$.

Let χ, χ_0, χ', φ, *and* $\mathfrak{o}$ *such that* $\{\chi \xrightarrow{\varphi} \chi_0, \chi \overset{\mathfrak{o}}{\cdots\!\!\rightarrow} \chi'\} \subseteq \Gamma$ *and* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$ *be given. It is clear that during the computation of* `FlowClosureStep`(Π) *there is an iteration of the* **foreach** *cycle when these values are assign to the corresponding variables. Moreover, the* **if** *condition on line 4 do not apply and thus the remaining body of the* **foreach** *cycle is processed. It is easy to see that when* (F1) $\varphi = \iota$ *and* $\bar{\mathfrak{o}}\iota = \{\sigma_1, \ldots, \sigma_k\}*$ *for some* ι *and* σ_1, ..., σ_k, *then* $\{\chi' \xrightarrow{\sigma_i} \chi' : 0 < i \leqslant k\} \cup \{\chi_0 \overset{\mathfrak{o}}{\cdots\!\!\rightarrow} \chi'\} \subseteq \Gamma_0$. *Otherwise, when* (F2) $\varphi \notin \mathrm{TypeTag}$ *or* $\bar{\mathfrak{o}}\varphi$ *is not a starred message type, then we see that there is some* χ_0' *such that* $\{\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0', \chi_0 \overset{\mathfrak{o}}{\cdots\!\!\rightarrow} \chi_0'\} \subseteq \Gamma_0$. *We conclude that* Π *is flow-closed because* $\Gamma_0 \subseteq \Gamma$. ∎

The following proves the completeness of `LocalClosureStep`, that is, that it preserves the existence of a nesting of its input in any restricted $\mathcal{R}$-type. In fact it is enough to assume that the shape predicate is flow-closed instead of being an $\mathcal{R}$-type.

LEMMA 12.9.3 (`FlowClosureStep` COMPLETENESS). *Let* Π' *be flow-closed and restricted. Let* $\Pi_1 =$ `FlowClosureStep`(Π). *When* $\delta \therefore \Pi \trianglelefteq \Pi'$ *then there is* δ_1 *such that* $\delta_1 \therefore \Pi_1 \trianglelefteq \Pi'$.

PROOF. *Let* $\Pi = \langle \Gamma, \chi_\mathsf{r} \rangle$ *and let* $\Pi_1 =$ `FlowClosureStep`(Π). *We see that* $\Pi_1 = \langle \Gamma \cup \Gamma_0, \chi_\mathsf{r} \rangle$ *for some* Γ_0 *and thus* Π_1 *contains all the edges of* Π *but* Π_1 *can contain some additional edges and nodes. Let* $\delta \therefore \Pi \trianglelefteq \Pi'$. *Firstly we shall construct* δ_1 *by extending* δ *with values for the additional nodes from* Π_1. *Secondly we shall prove that indeed* $\delta_1 \therefore \Pi_1 \trianglelefteq \Pi'$.

Let χ_0' *be a node in* Γ_0 *which is not in* Γ. *We need to define the value* $\delta_1(\chi_0')$. *We see that* χ_0' *was added to* Γ_0 *at line 12 as a fresh node in the iteration of the* **foreach** *cycle when the value of variable* χ_0' *was* χ_0'. *Let* χ, χ_0, χ', φ, *and* $\mathfrak{o}$ *be the values of the variables with corresponding names in this* **foreach** *iteration. We know that* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *and* $(\chi \overset{\mathfrak{o}}{\cdots\!\!\rightarrow} \chi') \in \Gamma$ *as well as that* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$. *It is easy to see that the flow-closure condition* (F2) *is satisfied for the two above edges. Now* $\delta \therefore \Pi \trianglelefteq \Pi'$ *gives us two edges* $\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)$ *and* $\delta(\chi) \overset{\mathfrak{o}'}{\cdots\!\!\rightarrow} \delta(\chi')$ *in* Π'

such that $\varphi \preccurlyeq \varphi'$ *and* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$. *Now Lemma 12.5.3 and Lemma 8.1.6 imply that the flow-closure condition* F2 *applies for two edges of* Π' *as noted in the paragraph before Lemma 12.5.3. Because* Π' *is flow closed, we thus obtain that there is some* χ_0'' *such that* $\delta(\chi') \xrightarrow{\bar{\mathfrak{o}}'\varphi'} \chi_0''$ *and* $\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \chi_0''$ *are in* Π'. *We set* $\delta_1(\chi_0') = \chi_0''$. *By Lemma 8.3.2 we obtain that* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$. *Thus we directly see that* δ_1 *acts as a correct simulation for the two edges* $\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0'$ *and* $\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_0'$ *newly added to* Γ_0 *in the iteration of the* **foreach** *cycle under consideration.*

Now we need to prove that $\delta_1 \therefore \Pi_1 \trianglelefteq \Pi'$. *We see that the root node of* Π_1 *is the root node of* Π *and thus it is enough to prove that for every edge in* Π_1 *there exists the edge in* Π' *proposed by Definition 12.1.3. This is clearly true for every edge already present on* Π *and thus it is enough to check only the edges from* Γ_0. *Edges were added to* Γ_0 *during the execution of the algorithm at lines 6 or 12 in some iteration of the* **foreach** *cycle. Let* χ, χ_0, χ', φ, *and* $\mathfrak{o}$ *be the values of the variables with corresponding names in an iteration of the* **foreach** *cycle when some new edges were added to* Γ_0. *We know that* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$ *and* $(\chi \overset{\mathfrak{o}}{\dashrightarrow} \chi') \in \Gamma$ *as well as that* $\mathrm{itags}(\varphi) \cap \mathrm{dom}(\mathfrak{o}) = \emptyset$. *Now* $\delta \therefore \Pi \trianglelefteq \Pi'$ *gives us two edges* $\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)$ *and* $\delta(\chi) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi')$ *in* Π' *such that* $\varphi \preccurlyeq \varphi'$ *and* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$.

Let new edges were added to Γ_0 *at line 6. Then we know that* $\varphi = \iota \in \mathrm{dom}(\mathfrak{o})$ *and* $\mathfrak{o}(\iota) = \Sigma*$ *for some* k *and* $\sigma_1, \ldots, \sigma_k$. *Moreover the edges newly added to* Γ_0 *are* $\{\chi' \xrightarrow{\sigma} \chi' : \sigma \in \Sigma\} \cup \{\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi'\}$. *Now Lemma 12.5.3 implies that* $\varphi' = \iota$ *and using also Lemma 8.1.6 we obtain that the flow-closure condition* (F1) *is satisfied for the edges* $\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)$ *and* $\delta(\chi) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi')$ *in* Π' *because* Π' *is flow closed. We know that* $\mathfrak{o}'(\iota) = (\Sigma')*$ *for some* Σ'. *Because* Π' *is flow closed we know that the edges* $\{\delta(\chi') \xrightarrow{\sigma} \delta(\chi') : \sigma \in \Sigma'\} \cup \{\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi')\}$ *are present in* Π'. *Now using Lemma 8.1.3 we obtain that* $\Sigma \subseteq \Sigma'$. *Thus for the edge* $\chi' \xrightarrow{\sigma} \chi'$ *with* $\sigma \in \Sigma$ *added to* Γ_0 *there is the edge* $\delta(\chi') \xrightarrow{\sigma} \delta(\chi')$ *in* Π' *as required (clearly* $\sigma \preccurlyeq \sigma$*). Moreover for the edge* $\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi'$ *added to* Γ_0 *there is* $\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi')$ *in* Π' *with* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$ *as required.*

Let some new edges were added to Γ_0 *at line 12. In this case the edges added to* Γ_0 *are* $\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0'$ *and* $\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_0'$. *During the construction of* δ_1 *we have already handled the case when the destination node* χ_0' *is a freshly created node. Thus the remaining case is when* $\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0'$ *is already present in* Γ. *Using* δ *we obtain the edge* $\delta(\chi') \xrightarrow{\varphi''} \delta(\chi_0')$ *in* Π' *with* $\bar{\mathfrak{o}}\varphi \preccurlyeq \varphi''$. *Now Lemma 12.5.3 and Lemma 8.1.6 give us that the flow-closure condition* (F2) *is satisfied for the edges* $\delta(\chi) \xrightarrow{\varphi'} \delta(\chi_0)$ *and* $\delta(\chi) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi')$ *from* Π'. *Because* Π' *is flow closed, we obtain that there is some node* χ_0'' *such that the edges* $\delta(\chi') \xrightarrow{\bar{\mathfrak{o}}'\varphi'} \chi_0''$ *and* $\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \chi_0''$ *are present in* Π'. *To prove the claim it is enough to prove that* $\delta(\chi_0') = \chi_0''$. *By Lemma 8.3.2 we obtain that* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$. *Now* $\bar{\mathfrak{o}}\varphi \preccurlyeq \varphi''$ *and* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$ *implies* $\varphi'' \approx \bar{\mathfrak{o}}'\varphi'$ *because* $[\![\bar{\mathfrak{o}}\varphi]\!] \neq \emptyset$. *We have that both* $\delta(\chi') \xrightarrow{\varphi''} \delta(\chi_0')$ *and* $\delta(\chi') \xrightarrow{\bar{\mathfrak{o}}'\varphi'} \chi_0''$ *are present in* Π' *and thus, because* Π' *is restricted, the width restriction implies that* $\delta(\chi_0) = \chi_0''$. *Thus for the edge* $\chi' \xrightarrow{\bar{\mathfrak{o}}\varphi} \chi_0'$ *in* Γ_0 *there is* $\delta(\chi') \xrightarrow{\bar{\mathfrak{o}}'\varphi'} \delta(\chi_0')$ *in* Π' *with* $\bar{\mathfrak{o}}\varphi \preccurlyeq \bar{\mathfrak{o}}'\varphi'$, *and for the edge*

$\chi_0 \overset{\mathfrak{o}}{\dashrightarrow} \chi_0'$ *there is the edge* $\delta(\chi_0) \overset{\mathfrak{o}'}{\dashrightarrow} \delta(\chi_0')$ *in* Π' *with* $\mathfrak{o} \preccurlyeq \mathfrak{o}'$ *as required.* ∎

12.10 Properties of the Type Inference Algorithm

Now we prove the termination of `PrincipalType` which is based on the argument outlined in Section 12.1.1 and it uses the upper bound of almost disjoint edge paths computed in Section 12.3.

PROPOSITION 12.10.1 (`PrincipalType` TERMINATION). `PrincipalType`$(P, \mathcal{R})$ *terminates for every* P *and* $\mathcal{R}$.

PROOF. `ProcessShape` *terminates by Proposition 12.6.5.* `SelectApplicableRules` *by design returns a finite number of rules when* $\mathcal{R}$ *is standard. When* $\mathcal{R}$ *is infinite and non-standard then the algorithm terminates with failure.* `RestrictGraph` *terminates by Lemma 12.7.14.* `LocalClosureStep` *terminates by Lemma 12.8.12 and it is easy to check that* `FlowClosureStep` *terminates for all inputs because it iterates over a finite objects. An iterative execution of the* **repeat** *loop in* `PrincipalType` *gives us the following sequence of shape predicates.*

$$\Pi_0 = \texttt{RestrictGraph(ProcessShape}(P)\texttt{)}$$
$$\Pi_{i+1} = \texttt{RestrictGraph(FlowClosureStep(LocalClosureStep(}\Pi_i, \mathcal{R}^{\mathsf{fin}}\texttt{)))}$$

The shape predicate Π_i *is the value of variable* Π *after the execution of line 4 in the* $(i+1)$*-th iteration of the* **repeat** *loop in the call of* `PrincipalType`$(P, \mathcal{R})$. *To prove the termination of* `PrincipalType` *it is enough to prove that there is* k *such that* $\Pi_k = \Pi_{k+1}$.

Let tags be the count of different type tags in P *a* $\mathcal{R}^{\mathsf{fin}}$ *plus one (for "•"). Let* $len = \mathrm{maxlen}(P)$. *Let maxpaths be the upper bound on different edge paths enumerated in Section 12.3. We have already argued in Section 12.3 that* $\mathrm{paths}(\Pi_k) \leqslant$ *maxpaths for all* k *and that* Π_k *contains only the tags from* $\mathcal{R}^{\mathsf{fin}}$ *and* P *and possibly "•". Now let us prove it in more details. Any action type* φ *from* `ProcessShape`(P) *is introduced at line 6 of* `ProcessShape`. *It is easy to see that* $\mathrm{maxlen}(\varphi) \leqslant len$ *and that* φ *contains only type tags from* P. `RestrictGraph` *does not introduce new action types and thus* $\mathrm{paths}(\Pi_0) \leqslant$ *maxpaths. There are four places where a new action type can be introduced to a shape graph during type inference: lines 6 and 12 in* `FlowClosureStep`, *and lines line 4 and 11 in* `RightRequired`. *It is easy to check that the newly introduced action type* φ *contains only type tags from* P *and* $\mathcal{R}^{\mathsf{fin}}$ *(and possibly "•"), and that* $\mathrm{maxlen}(\varphi) \leqslant len$. *All action types of* Π_{i+1} *which are not introduced by the above four lines are already present in* Π_i. *Thus* $\mathrm{paths}(\Pi_k) \leqslant$ *maxpaths for any* k *follows by induction.*

Now let us prove that $\mathrm{paths}(\Pi_k) \leqslant \mathrm{paths}(\Pi_{k+1})$ *for all* k. *Let* Γ_i *be the shape graph part* Π_i *for any* i, *that is, we have* $\Pi_i = \langle \Gamma_i, \chi_\mathsf{r} \rangle$. *Note that the all functions called from* `RestrictGraph` *preserve the root node. Let* $\Gamma_i^{\mathsf{local}}$ *be the set of edges newly introduced during the execution of* `LocalClosureStep`$(\Pi_i, \mathcal{R}^{\mathsf{fin}})$ *and let* Γ_i^{flow} *be the set of edges newly introduced by* `FlowClosureStep`$(\langle \Gamma_i \cup \Gamma_i^{\mathsf{local}}, \chi_\mathsf{r} \rangle)$. *Let* $\Gamma_i' = \Gamma_i \cup \Gamma_i^{\mathsf{local}} \cup \Gamma_i^{\mathsf{flow}}$. *By Lemma 12.7.10 there is* δ_i *such that* $\Pi_{i+1} = \delta_i(\langle \Gamma_i', \chi_\mathsf{r} \rangle)$. *It is clear that* $\mathrm{paths}(\langle \Gamma_i, \chi_\mathsf{r} \rangle) \leqslant \mathrm{paths}(\langle \Gamma_i', \chi_\mathsf{r} \rangle)$ *because every path in* Γ_i' *is a path in* Γ. *Thus by Lemma 12.7.11* $\mathrm{paths}(\Pi_i) \leqslant \mathrm{paths}(\Pi_{i+1})$.

Now let us prove that for every i *such that* $\mathrm{paths}(\Pi_i) = \mathrm{paths}(\Pi_{i+1})$ *there is* j *such that* $\Pi_{i+j} = \Pi_{i+j+1}$ *or* $\mathrm{paths}(\Pi_i) < \mathrm{paths}(\Pi_{i+j})$. *This is sufficient to prove the termination of the algorithm because the number* $\mathrm{paths}(\Pi_i)$ *can not grow over maxpaths. Let* $\mathrm{paths}(\Pi_i) = \mathrm{paths}(\Pi_{i+1})$ *but* $\Pi_i \neq \Pi_{i+1}$. *Thus we know that there is at least one new edge in* $\Gamma_i^{\mathsf{local}}$ *or* Γ_i^{flow} *because otherwise* $\Pi_i = \Pi_{i+1}$ *(because* δ_k *is an identity when* $\Gamma_i^{\mathsf{local}} = \Gamma_i^{\mathsf{flow}} = \emptyset$*).*

When there is some (non-flow) edge in $\Gamma_i^{\mathsf{local}}$ *or* Γ_i^{flow} *then let* $(\chi \xrightarrow{\varphi} \chi') \in \Gamma_i^{\mathsf{local}} \cup \Gamma_i^{\mathsf{flow}}$ *be the first edge newly introduced by the type inference algorithm. Thus we have* $(\chi \xrightarrow{\varphi} \chi') \notin \Gamma_i$ *but on the other hand we know that* χ *is a node in* Γ_i. *It is easy to prove by induction that there is a rooted path to every node in* Γ_i *and similarly for* Γ_i'. *By Lemma 12.7.12, there is a rooted path* $\{\chi_\mathsf{r} \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi\}$ *to* χ *in* Π_i *such that* $(\varphi_1, \ldots, \varphi_k)$ *is a disjoint edge path. Now* $(\varphi_1, \ldots, \varphi_k, \varphi)$ *is an edge path* Π_{i+1} *with at most one repetition. Let us prove that* $(\varphi_1, \ldots, \varphi_k, \varphi)$ *is not an edge path in* Π_i. *We know that* Π_i *is restricted and thus the edge path* $(\varphi_1, \ldots, \varphi_k)$ *uniquely determines the above rooted path* $\{\chi_\mathsf{r} \xrightarrow{\varphi_1} \chi_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} \chi\}$ *and its target node* χ. *Moreover we know that there is no* χ'' *such that* $(\chi \xrightarrow{\varphi} \chi'') \in \Gamma_i$ *because if there was some* χ'' *then the above mentioned* $(\chi \xrightarrow{\varphi} \chi')$ *would not be the first newly introduced edge (the algorithm would reuse* $(\chi \xrightarrow{\varphi} \chi'')$*). Thus* $(\varphi_1, \ldots, \varphi_k, \varphi)$ *is not an edge path in* Π_i *(not to say an edge path with at most one repetition). Thus* $\mathrm{paths}(\Pi_i) < \mathrm{paths}(\Pi_{i+1})$ *as required (we set* $j = 0$*).*

When there are only flow edges in $\Gamma_i^{\mathsf{local}}$ *and* Γ_i^{flow}, *then* Π_i *and* Π_{i+1} *have the same nodes. The set of nodes is preserved also for other shape predicates* Π_{i+2}, $\ldots$ *in the sequence as long as the number of edge paths with at most one repetition remains unchanged. With a fixed number of nodes there is clearly an upper bound on flow edges that can be added to the graph. Thus after finitely many steps no more flow edges can be added. Thus there is* j *such that after* j *steps either an ordinary non-flow edge is added to a shape graph (which increases the number of paths as proved above) or the shape graph remains unchanged (in which case the algorithm terminates).* ∎

Now we prove the correctness of `PrincipalType` discussed in Section 12.1.2.

PROPOSITION 12.10.2 (`PrincipalType` CORRECTNESS). *Let $\mathcal{R}$ be standard and $\Pi = \texttt{PrincipalType}(P, \mathcal{R})$. Then Π is a restricted $\mathcal{R}$-type of P.*

PROOF. *Let $\mathcal{R}$ be standard and $\Pi = \texttt{PrincipalType}(P, \mathcal{R})$. Let $\mathcal{R}^{\mathsf{fin}}$ be the value of variable $\mathcal{R}^{\mathsf{fin}}$. We see that the value of variable Π_0 in the last iteration of the* **repeat** *cycle is Π. Thus Π is restricted by Lemma 12.7.14. Moreover we can see that $\Pi = \texttt{FlowClosureStep}(\texttt{LocalClosureStep}(\Pi, \mathcal{R}^{\mathsf{fin}}))$ and thus $\Pi = \texttt{FlowClosureStep}(\Pi) = \texttt{LocalClosureStep}(\Pi, \mathcal{R}^{\mathsf{fin}})$ by Lemma 12.9.1. Furthermore by Lemma 12.9.2 and by Lemma 12.8.14 we obtain that Π is $\mathcal{R}^{\mathsf{fin}}$-type. When $\mathcal{R}$ is finite than $\mathcal{R}^{\mathsf{fin}} = \mathcal{R}$ and thus the result is $\mathcal{R}$-type. We know that $\mathcal{R}$ is monotonic and thus it is easy to verify that the type inference algorithm never introduces any type entity whose length is bigger than $\mathrm{maxlen}(P)$ into the shape graph. Thus clearly $\mathrm{maxlen}(\Pi) \leqslant \mathrm{maxlen}(P)$ and we can use Proposition 12.2.3 to prove that that Π is an $\mathcal{R}$-type whenever $\mathcal{R}$ is infinite and $\mathcal{R}^{\mathsf{fin}}$ is the rule description returned by* `SelectApplicableRules`.

Now we shall prove that $\vdash P : \Pi_0$ holds for Π_0 being the value of variable Π at any time of execution of the algorithm. This is true after evaluation of line 1 by Proposition 12.6.6. Each call to `RestrictGraph` *returns a shape predicate Π'_0 such that $\vdash P : \Pi'_0$ by Lemma 12.7.10 and Lemma 12.5.1. Now both* `FlowClosureStep` *and* `LocalClosureStep` *only add edges to Π'_0 preserving its root. Thus $\vdash P : \Pi_0$ hold for the value Π_0 of variable Π at any point of the execution of* `PrincipalType`. *Hence $\vdash P : \Pi$.* ∎

Finally the following proves the main property of `PrincipalType` which is completeness together with correctness. Of course, the correctness is proved using the previous Proposition 12.10.2. The completeness of `PrincipalType` was discussed in Section 12.1.3.

THEOREM 12.10.3 (`PrincipalType` COMPLETENESS). *Let $\mathcal{R}$ be standard and $\Pi = \texttt{PrincipalType}(P, \mathcal{R})$. Then Π is a principal restricted $\mathcal{R}$-type of P.*

PROOF. *Let $\mathcal{R}$ be standard and $\Pi = \texttt{PrincipalType}(P, \mathcal{R})$. Let $\mathcal{R}^{\mathsf{fin}}$ be the value of variable $\mathcal{R}^{\mathsf{fin}}$. By Proposition 12.10.2 we have that Π is a restricted $\mathcal{R}$-type of P. We also know that Π is a restricted $\mathcal{R}^{\mathsf{fin}}$-type of P. Firstly, let us prove that Π is a principal $\mathcal{R}^{\mathsf{fin}}$-type of P. Let us take the derivation $\Pi_0, \ldots, \Pi_k$ of Π, that is, the following sequence with $\Pi_k = \Pi$.*

$$\Pi_0 = \texttt{RestrictGraph}(\texttt{ProcessShape}(P))$$
$$\Pi_{i+1} = \texttt{RestrictGraph}(\texttt{FlowClosureStep}(\texttt{LocalClosureStep}(\Pi_i, \mathcal{R}^{\mathsf{fin}})))$$

Let Π' be a restricted $\mathcal{R}^{\mathsf{fin}}$-type of P. By Proposition 12.6.7 and Lemma 12.7.15 we obtain that there is some δ_0 such that $\delta_0 \therefore \Pi_0 \trianglelefteq \Pi'$. It is easy to prove by induction on i and by Lemma 12.8.15, Lemma 12.9.3, and Lemma 12.7.15 that there is δ_i such

that $\delta_i \therefore \Pi_i \trianglelefteq \Pi'$ *for all* $i \leqslant k$*. Thus* $\delta_k \therefore \Pi \trianglelefteq \Pi'$ *and by Lemma 12.5.2 we obtain that* $\Pi \preccurlyeq \Pi'$*. Hence* Π *is a principal* $\mathcal{R}^{\mathsf{fin}}$*-type of* P*.*

Now every $\mathcal{R}$*-type is an* $\mathcal{R}^{\mathsf{fin}}$*-type because* $\mathcal{R}^{\mathsf{fin}} \subseteq \mathcal{R}$*. Let* Π' *be a restricted* $\mathcal{R}$*-type of* P*. Then* Π' *is a restricted* $\mathcal{R}^{\mathsf{fin}}$*-type of* P *and by the above we know that* $\Pi \preccurlyeq \Pi'$*. Hence* Π *is also a principal* $\mathcal{R}$*-type of* P*.* ∎

Part III

Applications and Expressiveness of Shape Types

Chapter 13

General View of Analysis Systems

This section discusses type and flow analysis systems, how to use Poly✶ to achieve goals attained by other systems, how to use Poly✶ on its own, how and why to relate Meta✶ with other calculi, and several possibilities how a formal comparison of Poly✶ with other systems can be made.

13.1 General View of Analysis Systems

Different type and static analysis systems are designed for different purposes. A single system is usually intended to statically verify a specific fixed property of processes of a given calculus, for example, that a process does not execute an ill-formed instruction. Commonly, it is easy to verify this property for a specific process. On the other hand, to verify that the property in question is satisfied for a given process and for all of its successor states, is generally much more complicated task. Type and static analysis systems provide an effective solution of this problem.

Here we repeat general notations introduced in Section 1.5. A typical type or static analysis system S_C for the process calculus C works as follows. Firstly, it defines the set of predicates. Let ρ range over it. Predicates formally represent properties which the system reasons about and verifies. Secondly, the system defines a binary relation on processes and predicates. Let B range over processes of C. Let us write the relation as $\triangleright B : \rho$. This relation formally represents the statement "B has the property ρ". The relation is desired to be effectively verifiable. Thirdly, the system (usually) enjoys the subject reduction property, which states that the relation $\triangleright$ is preserved under rewriting of processes, that is, $\triangleright B_0 : \rho$ implies $\triangleright B_1 : \rho$ for any successor state B_1 of the process B_0.

13.2 How To Use Poly✶ ?

Consider some type/static analysis system S_C with the properties from the previous section. Furthermore suppose that the rewriting rules of the calculus C can be described in Meta✶ by the set of rewriting rules $\mathcal{R}$. This gives us the instantiation $C_\mathcal{R}$ of Meta✶ and the type system $S_\mathcal{R}$ provided by Poly✶. Now suppose that we would like to compare expressiveness of S_C and $S_\mathcal{R}$. We would like to know whether questions answerable by the relation $\rhd$ of S_C can be expressed and answered within $S_\mathcal{R}$. Being able to do this for several different systems from the literature would lead us to the conclusion that the generic concept of Poly✶ shape types is at least as expressive as the single-purpose predicates of the selected systems. We shall show how to do this for three systems from the literature. Several possible approaches how to do this are outlined in Section 13.4.

Some systems are designed to verify a certain fixed property of processes, for example, that a process does not execute an ill-formed instruction. When such a property is given we can use Poly✶ directly without referencing S_C. Let P denote the property in question. For these systems the following holds: When there exists some ρ such that $\rhd B : \rho$ then B has property P. Usually some over-approximation is encountered, which means that the opposite implication is not always satisfied. This is a commonly accepted trade-off between preciseness and time complexity of the verification of P. Suppose that we can formulate a condition on a Poly✶ shape type Π whose fulfillment implies that every process matching Π has property P. Then we can use Poly✶ to verify P directly. We can also choose property P directly without any references to other type or static analysis systems. We shall show how to use Poly✶ to verify communication safety of the π-calculus and Mobile Ambients processes. We show that Poly✶ provides better results, that is, it over-approximates communication safety less, than other two systems designed specifically for this purpose. Moreover, we show that Poly✶ can also exactly recognize whether or not $\rhd B : \rho$ holds for a given B and ρ in these two systems. This is important because the relation $\rhd B : \rho$ might be used by some applications of S_C for various purposes and not only to verify communication safety.

13.3 Relating Calculi C and $C_\mathcal{R}$

Calculi $C_\mathcal{R}$ and C are usually reasonably equivalent. Nevertheless, for the reason of a formal comparison of the systems S_C and $S_\mathcal{R}$ a reasonable relationship between the calculi C and $C_\mathcal{R}$ has to be proved. It is important to understand, however, that a similar relationship has to be established only for the reasons of formal comparisons, such are those presented in the next chapters, and it is not required for a standard use of Poly✶.

At first we need an encoding $(\!|\cdot|\!)$ which translates processes of C into META✶ processes. Because of a benevolent syntax of META✶, this encoding is in many cases almost an identity. To avoid technical problems, such as a different handling of α-conversion, we suppose that processes of C are built from META✶ names, and that C does α-conversion as META✶. One can easily construct an equivalent version of C which meets this requirement when necessary. We suppose that the encoding $(\!|\cdot|\!)$ preserves free names and type tags. The relationship between the rewriting relation $\rightarrow$ of C and the relation $\overset{\mathcal{R}}{\hookrightarrow}$ of META✶ has two parts. The first says that $B_0 \rightarrow B_1$ implies $(\!|B_0|\!) \overset{\mathcal{R}}{\hookrightarrow} (\!|B_1|\!)$. The second ensures that whenever $(\!|B_0|\!) \overset{\mathcal{R}}{\hookrightarrow} P_1$ then P_1 is the translation of some B_1 such that $B_0 \rightarrow B_1$. To handle subtle differences in structural congruences of different calculi, we formulate this property modulo structural congruences $\equiv$ of S_C and META✶. As mentioned, proving Property 13.3.1 is usually easy.

PROPERTY 13.3.1 (FAITHFUL ENCODING). *When $B_0 \rightarrow B_1$ then there are B_0' and B_1' such that $B_0 \equiv B_0'$ & $(\!|B_0'|\!) \overset{\mathcal{R}}{\hookrightarrow} (\!|B_1'|\!)$ & $B_1' \equiv B_1$. When $(\!|B_0|\!) \overset{\mathcal{R}}{\hookrightarrow} P_1$ then there is B_1 such that $B_0 \rightarrow B_1$ & $(\!|B_1|\!) \equiv P_1$.* ∎

13.4 Comparing Systems S_C and $S_{\mathcal{R}}$

A straightforward way to relate S_C and $S_{\mathcal{R}}$ is to define an embedding $\langle\!|\cdot|\!\rangle$ of predicates of S_C to POLY✶ types such that $\rhd B : \rho$ if and only if $\vdash (\!|B|\!) : \langle\!|\rho|\!\rangle$. This approach is not possible when the relation $\rhd$ of S_C is preserved under renaming of bound type tags of a process. Unfortunately, this is the case of majority of the systems in literature, especially of those we work with in this paper. Recall that we suppose that C builds processes from META✶ names (which include type tags). The problem is that bound type tags are used to build POLY✶ shape types. Thus, when a bound type tag in a process is changed then the new process does not need to match the same shape types as before (because typability in POLY✶ is not preserved under renaming of bound type tags).

To put it another way, a straightforward embedding of one type system in another can not be constructed when predicates of the systems differ too much and when they contain different information. For example POLY✶ shape types contain information about bound names which can appear in a process. These information contain their type tags and an upper bound on the count of bound names (with different type tags). These information are necessary to construct a shape type. On the other hand these information are not usually contained in predicates of other systems. Thus we can not construct a direct embedding of these other systems in POLY✶ because the information about bound names are missing.

In order to demonstrate explain this, let us suppose the encoding $\langle\!|\cdot|\!\rangle$ with the

above desired property, and let $\rhd(\nu x)B_0 : \rho$ for some B_0 and ρ such that $x \in \mathrm{fn}(B_0)$. Now, because both $(\!|(\nu x)(B_0)|\!)$ and $\langle\!|\rho|\!\rangle$ are finite objects, we can take some type tag ι which is in none of them. Let a be the basic name of x, that is, $x = a^{\iota_0}$ for some ι_0. Let us take $B' = (\nu a^{\iota})(B_0\{x \mapsto a^{\iota}\})$, that is, let us change the type tag of x from ι_0 to ι in all occurrences of x. Because $\rhd$ is preserved under renaming of bound type tags of S_C's processes, we have that also $\rhd B' : \rho$. But we can see that $(\!|B'|\!)$ can hardly match $\langle\!|\rho|\!\rangle$ because $\langle\!|\rho|\!\rangle$ does not contain any ι_0 necessary to match occurrences of a^{ι_0} in $(\!|B'|\!)$. A similar argument can be made even when we do not require $(\!|\cdot|\!)$ to preserve type tags.

We investigate another ways to compare S_C and $S_{\mathcal{R}}$ to avoid the above problem. The first one, which we use in Chapter 14 to compare Poly✶ with a typed version of the π-calculus, exploits the existence of principal types for processes in $S_{\mathcal{R}}$. We answer the question $\rhd B : \rho$ by performing a simple check on a Poly✶ principal type Π_B of $(\!|B|\!)$. Formally we define the relation $\rho \cong \Pi$, which says that ρ "agrees" with Π, and we prove that $\rhd B : \rho$ if and only if $\rho \cong \Pi_B$.

Another approach to compare S_C and $S_{\mathcal{R}}$ is an enhancement of the straightforward comparison outlined above. We equip a translation of S_C's predicate ρ into a Poly✶ type with necessary information I_B about bound names of the process B. We translate ρ and I_B into the Poly✶ type $\langle\!|\rho, I_B|\!\rangle$ and we prove that $\rhd B : \rho$ if and only if $\vdash (\!|B|\!) : \langle\!|\rho, I_B|\!\rangle$. We use this approach to compare Poly✶ with a typed version of Mobile Ambients in Chapter 16.

Yet another style of comparison is used to compare Poly✶ with a flow analysis system for BioAmbients in Chapter 18. For a given B, we compute a Poly✶ principal type Π_B of $(\!|B|\!)$ and use it to construct a predicate ρ_B of system S_C such that $\rhd B : \rho_B$. Then we take the actual result ρ of the analysis of B computed by S_C and we prove that ρ_B constructed from Π_B is at least as precise as ρ, let us write it as $\rho_B \subseteq \rho$. Our result says that $\rhd B : \rho$ implies $\rho_B \subseteq \rho$. The opposite implication would not give a meaningful result in this particular case. Additionally, we also show how to use the approach from the previous paragraph to compare Poly✶ with the same BioAmbients flow analysis system.

Chapter 14

Shape Types for the π-calculus

This chapter demonstrates how to use POLY✶ with a polyadic π-calculus and proves POLY✶ to be more expressive than a π-calculus type system from the literature. Section 14.1 introduces the π-calculus, Section 14.2 introduces a type system from the literature [Tur95, Chapter 3], Section 14.3 describes the type system for the π-calculus provided by POLY✶, Section 14.4 formally compares expressiveness of the above two systems, and finally Section 14.5 provides conclusions and discussion of a related work.

14.1 A Polyadic π-calculus

The π-calculus [MPW92a, Mil99] is a process calculus involving process mobility developed by Milner, Parrow, and Walker. Mobility is abstracted as channel-based communication whose objects are atomic names. Channel labels are not distinguished from names and can be passed by communication. This ability, referred as *link passing*, is the π-calculus feature that most distinguishes it from its predecessors. We use a polyadic version of the π-calculus which supports communication of tuples of names.

Figure 14.1 presents the syntax and semantics of the π-calculus. Processes are built from META✶ names which contain type tags. The process "$c(n_1, \ldots, n_k).B$", which (input)-binds the names n_i's, waits to receive a k-tuple of names over channel c and then behaves like B with the received values substituted for n_i's. The process "$c\langle n_1, \ldots, n_k\rangle.B$" sends the k-tuple n_1, …, n_k over channel c and then behaves like B. Other constructors have the meaning as in META✶ (Chapter 3). The sets of names and type tags $\mathrm{fn}(B)$, $\mathrm{ftags}(B)$, $\mathrm{itags}(B)$, $\mathrm{ntags}(B)$ are defined as in META✶.

Processes are identified up to α-conversion of bound names which preserves type tags. A substitution in the π-calculus is a finite function from names to names, and its application to B is written postfix, that is, "$B\{n \mapsto m\}$". We set SpecialTag = $\{\bullet\}$ and forbid "$\bullet$" to be used in processes because it is reserved for POLY✶. We

Syntax of the π-calculus processes:

$$\begin{array}{rcccl} c, n, m & \in & \text{PiName} & = & \text{Name} \\ N & \in & \text{PiAction} & ::= & c(n_1, \ldots, n_k) \mid c\langle n_1, \ldots, n_k \rangle \\ B & \in & \text{PiProcess} & ::= & \mathsf{0} \mid (B_0 \mid B_1) \mid N.B \mid !B \mid (\nu n)B \end{array}$$

Structural equivalence of the π-calculus:

$$\frac{}{B \equiv B} \qquad \frac{B_0 \equiv B_1}{B_1 \equiv B_0} \qquad \frac{B_0 \equiv B_1 \quad B_1 \equiv B_2}{B_0 \equiv B_2} \qquad \frac{B_0 \equiv B_1}{B_0 \mid B_2 \equiv B_1 \mid B_2}$$

$$\frac{B_0 \equiv B_1}{N.B_0 \equiv N.B_1} \qquad \frac{B_0 \equiv B_1}{!B_0 \equiv !B_1} \qquad \frac{B_0 \equiv B_1}{(\nu n)B_0 \equiv (\nu n)B_1} \qquad \frac{}{B_0 \mid B_1 \equiv B_1 \mid B_0}$$

$$\frac{}{B_0 \mid (B_1 \mid B_2) \equiv (B_0 \mid B_1) \mid B_2} \qquad \frac{}{B \equiv B \mid \mathsf{0}} \qquad \frac{}{!B \equiv B \mid !B}$$

$$\frac{n \notin \mathrm{fn}(B_1)}{(\nu n)B_0 \mid B_1 \equiv (\nu n)(B_0 \mid B_1)}$$

Rewriting relation of the π-calculus:

$$\frac{}{c(n_1, \ldots, n_k).B_0 \mid c\langle m_1, \ldots, m_k \rangle.B_1 \to B_0\{n_1 \mapsto m_1, \ldots, n_k \mapsto m_k\} \mid B_1}$$

$$\frac{B_0 \to B_1}{(\nu n)B_0 \to (\nu n)B_1} \qquad \frac{B_0 \to B_1}{B_0 \mid B_2 \to B_1 \mid B_2} \qquad \frac{B_0' \equiv B_0 \quad B_0 \to B_1 \quad B_1 \equiv B_1'}{B_0' \to B_1'}$$

Figure 14.1: The syntax and semantics of the π-calculus.

require all processes to be well formed according to the following definition. Well-formedness can be achieved by name renaming if necessary and it is preserved by rewriting.

Definition 14.1.1. *A process B is* **well formed** *iff all the following hold.*

(S1) $\mathrm{ftags}(B) \cup \mathrm{itags}(B)$ *is disjoint with* $\mathrm{ntags}(B)$

(S2) *for* $(n_1, \ldots, n_k).B_0$ *in* B, $\overline{n_i} \notin \mathrm{itags}(B_0)$ *and* $\overline{n_i} \neq \overline{n_j}$ *when* $i \neq j$

(S3) B *do not contain any type tag from* SpecialTag ∎

Example 14.1.2. *Let us consider the following process.*

$$B = !\mathsf{s(x,y).x\langle y\rangle.0} \mid \mathsf{s\langle a,n\rangle.0} \mid \mathsf{a(v).v(p).0} \mid \mathsf{n\langle o\rangle.0} \mid \mathsf{s\langle b,m\rangle.0} \mid \mathsf{b(w).w(q,r).0} \mid \mathsf{m\langle o,o\rangle.0}$$

Using the rewriting relation → sequentially four times we can obtain (among others) the process "$\mathsf{!s(x,y).x\langle y\rangle.0 \mid n(p).0 \mid n\langle o\rangle.0 \mid m(q,r).0 \mid m\langle o,o\rangle.0}$*".* □

Syntax of TPI *types:*

$$\begin{array}{rcll} \beta & \in & \text{PiTypeVariable} & ::= \ \iota \mid \iota' \mid \iota'' \mid \cdots \\ \kappa & \in & \text{PiType} & ::= \ \beta \mid \uparrow[\kappa_1, \ldots, \kappa_k] \\ \Delta & \in & \text{PiContext} & = \ \text{TypeTag} \rightarrow_{\text{fin}} \text{PiType} \end{array}$$

Typing rules of TPI*:*

$$\frac{}{\Delta \vdash \mathsf{0}} \qquad \frac{\Delta \vdash B_0 \quad \Delta \vdash B_1}{\Delta \vdash B_0 \mid B_1} \qquad \frac{\Delta \vdash B}{\Delta \vdash !B} \qquad \frac{\Delta[\overline{n} \mapsto \kappa] \vdash B}{\Delta \vdash (\nu n)B}$$

$$\frac{\Delta(\overline{c}) = \uparrow[\kappa_1, \ldots, \kappa_k] \quad \Delta[\overline{n_1} \mapsto \kappa_1, \ldots, \overline{n_k} \mapsto \kappa_k] \vdash B}{\Delta \vdash c(n_1, \ldots, n_k).B}$$

$$\frac{\Delta(\overline{c}) = \uparrow[\Delta(\overline{n_1}), \ldots, \Delta(\overline{n_k})] \quad \Delta \vdash B}{\Delta \vdash c\langle n_1, \ldots, n_k\rangle.B}$$

Figure 14.2: Syntax of TPI types and typing rules.

14.2 Types for the Polyadic π-calculus (TPI)

We compare POLY✶ with a simple type system [Tur95, Chapter 3] for the polyadic π-calculus presented by Turner which we name TPI. TPI is essentially Milner's sort discipline [Mil99]. In the polyadic settings, an arity mismatch error on channel c can occur when the lengths of the sent and received tuple do not agree, like in "$c(n).0 \mid c\langle m, m\rangle.0$". Processes which can never evolve to a state with a similar situation are called *communication safe*. TPI verifies communication safety of π-processes.

The syntax and typing rules of TPI are presented in Figure 14.2. Recall that $\overline{n}$ denotes the type tag of n. Types κ are assigned to names. Type variables β are types of names which are not used as channel labels. The type "$\uparrow[\kappa_1, \ldots, \kappa_k]$" describes a channel which can be used to communicate any k-tuple whose i-th name has type κ_i. A context Δ assigns types to free names of a process (via their type tags[1]). The relation $\Delta \vdash B$, which is preserved under rewriting, expresses that the actual usage of channels in B agrees with Δ. When $\Delta \vdash B$ for some Δ then B is communication safe. The opposite does not necessarily hold.

EXAMPLE 14.2.1. *Given B from Example 14.1.2 we can see that there is no Δ such that $\Delta \vdash B$. It is because the parts* s<a, n> *and* s<a, m> *imply that types of* n *and* m *must be equal while the parts* n<o> *and* m<o, o> *force them to be different. On the other hand B is communication safe. We check this using* POLY✶ *in Example 14.3.1.* □

[1]Turner's original system does not use META✶ type tags and assigns types directly to names. This technical variation simplifies the correspondence with POLY✶.

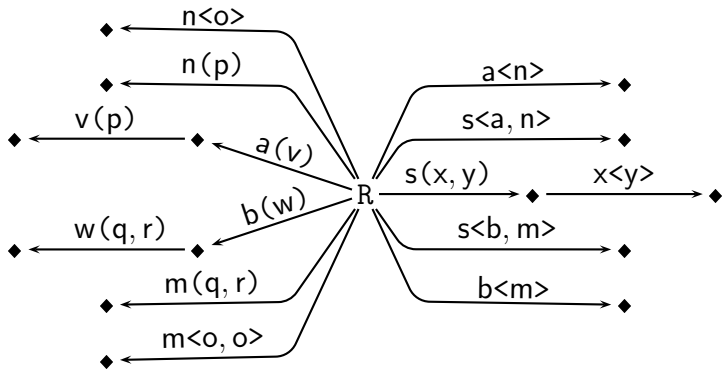
14.3 Instantiation of META✶ to the π-calculus

The π-calculus syntax from Section 14.1 already matches the META✶ syntax and thus only the following $\mathcal{P}$ is needed to instantiate META✶ to the calculus $C_\mathcal{P}$ and POLY✶ to its type system $S_\mathcal{P}$. Section 14.4 shows that $C_\mathcal{P}$ is essentially identical to the above π-calculus. The set $\mathcal{P}$ below is the set $\mathcal{P}_{\mathsf{poly}}$ from Section 10.3.

$$\mathcal{P} = \big\{\mathbf{rewrite}\{\, \mathring{\mathtt{c}}\mathtt{<}\mathring{\mathtt{a}}_1,\ldots,\mathring{\mathtt{a}}_n\mathtt{>.}\mathring{\mathtt{P}} \mid \mathring{\mathtt{c}}\mathtt{(}\mathring{\mathtt{x}}_1,\ldots,\mathring{\mathtt{x}}_n\mathtt{).}\mathring{\mathtt{Q}} \hookrightarrow \mathring{\mathtt{P}} \mid \{\mathring{\mathtt{x}}_1 \mathtt{:=} \mathring{\mathtt{a}}_1,\ldots \mathring{\mathtt{x}}_n \mathtt{:=} \mathring{\mathtt{a}}_n\mathtt{}\mathring{\mathtt{Q}}\,\}\mathtt{}\colon n \geqslant 0\big\}$$

Each communication prefix length has its own rule; in the type inference algorithm implementation, a single rule can uniformly handle all lengths, but the formal META✶ presentation is deliberately simpler. The next example shows how to check communication safety in $S_\mathcal{P}$ without using TPI.

EXAMPLE 14.3.1. *Let P be a* META✶ *equivalent of B from Example 14.1.2. The processes P and B share the syntax and differ only by a syntactic category. We can use the* POLY✶*'s type inference algorithm to compute a principal type Π_P of P which has with root* R *and the following shape graph (with flow edges removed).*



Names of non-root nodes are omitted because they are irrelevant. The type Π_P contains all computational futures of P in one place. Thus, because there are no two edges from the root node labeled by "$\iota(\iota_1,\ldots,\iota_i)$" and "$\iota\mathtt{<}\iota'_1,\ldots,\iota'_j\mathtt{>}$" with $i \neq j$, we can conclude that P is communication safe which Example 14.2.1 shows TPI *can not do. Our type inference implementation can be instructed (using an additional rule) to insert a special error name at the place of communication errors. Then checking communication safety is equivalent to checking the presence of the special error name.* □

$$
\begin{array}{ll}
(\!|\mathtt{0}|\!) = \mathtt{0} & (\!|B_0 \mid B_1|\!) = (\!|B_0|\!) \mid (\!|B_1|\!) \\
(\!|!B|\!) = !(\!|B|\!) & (\!|c(n_1,\ldots,n_k).B|\!) = c(n_1,\ldots,n_k).(\!|B|\!) \\
(\!|(\nu n)B|\!) = \nu n.(\!|B|\!) & (\!|c\langle n_1,\ldots,n_k\rangle.B|\!) = c\langle n_1,\ldots,n_k\rangle.(\!|B|\!)
\end{array}
$$

Figure 14.3: Encoding of π-calculus processes in META✶.

The set of expected and actual channel types of Γ*:*

$$\mathrm{chtypes}(\Delta,\Gamma) = \{(\Delta(\iota), \uparrow[\Delta(\iota_1),\ldots,\Delta(\iota_k)]) : (\chi \xrightarrow{\iota(\iota_1,\ldots,\iota_k)} \chi') \in \Gamma \vee (\chi \xrightarrow{\iota\langle\iota_1,\ldots,\iota_k\rangle} \chi') \in \Gamma\}$$

Context Δ *and shape type* Π *agreement relation* $\cong$*:*

Write $\Delta \cong \langle\Gamma,\chi\rangle$ when there is some Δ' with the domain disjoint from Δ such that $\mathrm{chtypes}(\Delta \cup \Delta', \Gamma)$ is defined and is an identity.

Figure 14.4: Property of shape types corresponding to $\vdash$ of TPI.

14.4 Embedding of TPI in POLY✶

Using the terminology from Section 13.1 we have that the calculus C is the π-calculus, S_C is TPI, predicates ρ of S_C are contexts Δ, and S_C's relation $\triangleright B : \rho$ is $\Delta \vdash B$. Moreover $\mathcal{R}$ is $\mathcal{P}$ which was introduced with $C_{\mathcal{P}}$ and $S_{\mathcal{P}}$ in Section 14.3. This section provides a formal comparison which shows how to, for a given B and Δ, answer the question $\Delta \vdash B$ using $S_{\mathcal{P}}$.

As stated in Section 13.3, to relate TPI and $S_{\mathcal{P}}$ we need to provide a faithful encoding $(\!|\cdot|\!)$ of π-processes in META✶. This $(\!|\cdot|\!)$, presented in Figure 14.3, is almost an identity because the π-calculus syntax (Figure 14.1) already agrees with META✶. Thus $(\!|\cdot|\!)$ mainly changes the syntactic category. Property 13.3.1 holds in the above context.

Given Δ, we define a shape type property which holds for the principal type Π_B of $(\!|B|\!)$ iff $\Delta \vdash B$. The property is given by the relation $\Delta \cong \Pi$ from Figure 14.4. The set $\mathrm{chtypes}(\Delta,\Gamma)$ contains pairs of TPI types extracted from Γ. Each pair corresponds to an edge of Γ labeled by a form type "$\iota(\iota_1,\ldots,\iota_k)$" or "$\iota\langle\iota_1,\ldots,\iota_k\rangle$". The first member of the pair is ι's type expected by Δ, and the second member computes ι's actual usage from the types of ι_i's. The set $\mathrm{chtypes}(\Delta,\Gamma)$ is undefined when some required value of Δ is not defined. The context Δ' from the definition of $\cong$ provides types of names originally bound in B. These are not mentioned by Δ but are in Γ. The following theorem shows how to answer $\Delta \vdash B$ by $\cong$.

THEOREM 14.4.1. *Let no two different binders in* B *bind the same type tag,* Π_B *be a principal (*$\mathcal{P}$*-)type of* $(\!|B|\!)$*, and* $\mathrm{dom}(\Delta) = \mathrm{ftags}(B)$*. Then* $\Delta \vdash B$ *iff* $\Delta \cong \Pi_B$*.*

The requirement on different binders (which can be achieved by renaming) is not preserved under rewriting because replication can introduce two same-named binders. However, when all binding type tags differ in B_0, then the theorem holds

for any successor B_1 of B_0 even when the requirement is not met for B_1. We want to ensure that the derivation of $\Delta \vdash B$ does not assign different types to different type tags. A slightly stronger assumption of Theorem 14.4.1 simplifies its formulation. The theorem uses principal types and does not necessarily hold for a non-principal $\mathcal{P}$-type Π of $(\!|B|\!)$ because Π's additional edges not needed to match $(\!|B|\!)$ can preclude $\Delta \cong \Pi$.

14.5 Conclusions

We showed a process (Example 14.1.2) that can not be proved communication safe by TPI (Example 14.2.1) but can be proved so by POLY✶ (Example 14.3.1). Theorem 14.4.1 implies that POLY✶ recognizes safety of all TPI-safe processes. Thus we conclude that POLY✶ is better in recognition of communication safety then TPI. Theorem 14.4.1 allows to recognize typability in TPI: B is typable in TPI iff $\emptyset \cong \Pi_B$. This is computable because a POLY✶ principal type can always be found (for $S_{\mathcal{P}}$ in polynomial time), and checking $\cong$ is easy.

Turner [Tur95, Ch. 5] presents also a polymorphic system for the π-calculus which recognizes B from Example 14.1.2 as safe. However, with respect to our best knowledge, it can not recognize safety of the process "B | s<n, a>.0" which POLY✶ can do. We are not aware of any process that can be recognized safe by Turner's polymorphic system but not by POLY✶. It must be noted, there are still processes which POLY✶ can not prove safe, for example, "a(x).a(y, z).0 | a<o>.a<o, o>.0".

Other π-calculus type systems are found in the literature. Kobayashi and Igarashi [IK01] present types for the π-calculus looking like simplified processes which can verify properties which are hard to express using shape types (race conditions, deadlock detection) but do not support polymorphism. One can expect applications where POLY✶ is more expressive as well as contrariwise. Shape types, however, work for many process calculi, not just the π-calculus.

Chapter 15

Details on the TPI Embedding

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

Definition 15.0.1 extends the definition from Figure 14.4 with some additional notations. Proposition 15.0.2 is the left-to-right implication of Theorem 14.4.1 and Proposition 15.0.3 is its right-to-left implication. The proofs use standard weakening and strengthening lemmas [Tur95, Lem. 3.8-9].

DEFINITION 15.0.1. *Write* $\Delta \cong \Gamma$ *when* $\Delta \cong \langle \Gamma, \chi \rangle$ *for an arbitrary* χ*. Moreover, we say that* $\Delta \cong \Gamma$ **holds via** Δ' *when* $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset$ *and* $\mathrm{chtypes}(\Delta \cup \Delta', \Gamma)$ *is defined and an identity.* ∎

PROPOSITION 15.0.2. *Let*

(1) B *be a* π*-process such that no two different binders bind the type tag,*
(2) Π_B *be a principal* $\mathcal{P}$*-type of* $(\!|B|\!)$*,*
(3) $\mathrm{dom}(\Delta) \subseteq \mathrm{ftags}(B)$*, and*
(4) $\Delta \vdash B$*.*

Then $\Delta \cong \Pi_B$*.*

PROOF. *By induction on the structure of* B*. Let* $\Pi = \langle \Gamma, \chi \rangle = \Pi_B$*. Let*

$B = \mathtt{0}$**:** *Then* $\Gamma \cong \Delta$ *holds trivially because* $\Gamma = \emptyset$*.*

$B = B_0 \mid B_1$**:** *Let* $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ *be a principal type of* $(\!|B_0|\!)$ *and* $\Pi_1 = \langle \Gamma_1, \chi_1 \rangle$ *be a principal type of* $(\!|B_1|\!)$*. Take*

$$\Delta_0 = \{(\iota \mapsto \kappa) \colon (\iota \mapsto \kappa) \in \Delta \;\&\; \iota \in \mathrm{ftags}(B_0)\}$$
$$\Delta_1 = \{(\iota \mapsto \kappa) \colon (\iota \mapsto \kappa) \in \Delta \;\&\; \iota \in \mathrm{ftags}(B_1)\}$$

Let us verify the assumptions of the induction step for B_0*,* Π_0*, and* Δ_0*:*

(1) Clear.
(2) Clear.
(3) $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta) \cap \mathrm{ftags}(B_0) \subseteq \mathrm{ftags}(B_0)$.
(4) Here $\Delta_0 \vdash B_0$ *follows from* $\Delta \vdash B_0$ *by strengthening.*

Similarly the assumptions are satisfied for B_1, Π_1, *and* Δ_1. *Thus by the induction hypothesis we have that* $\Delta_0 \cong \Pi_0$ *and* $\Delta_1 \cong \Pi_1$. *Let* $\Delta_0 \cong \Gamma_0$ *via* Δ'_0 *and let* $\Delta_1 \cong \Gamma_1$ *via* Δ'_1. *Because* (1) *we can suppose that* $\mathrm{dom}(\Delta'_0) \cap \mathrm{dom}(\Delta'_1) = \emptyset$. *Take* $\Delta' = \Delta'_0 \cup \Delta'_1$. *We shall proof that* $\Delta \cong \Gamma$ *via* Δ_0. *Denote* $\Delta^+ = \Delta \cup \Delta'$. *Although* Γ *can contain some additional edges not contained in* Γ_0 *and* Γ_1 *it can not introduce new type tags. When* Γ *contains a type substitution* $\mathfrak{o}$ *such that* $\mathfrak{o}(\iota) = \iota'$ *then it is not hard to that* $\Delta^+(\iota) = \Delta^+(\iota')$. *Thus all additional members in* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *are identities because they are constructed by application of type substitutions in* Γ. *Thus the claim.*

$B = c(n_1, \ldots, n_k).B_0$**:** *Let* $\iota = \overline{c}$ *and* $\iota_i = \overline{n_i}$ *for* $0 < i \leqslant k$. *Let* $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ *be a principal type of* $(\!|B_0|\!)$. *There are some* $\kappa_1, \ldots, \kappa_k$ *such that* $\Delta(\iota) = \uparrow[\kappa_1, \ldots, \kappa_k]$. *Take* Δ_0 *which does not contain* ι_i *not mentioned in* B_0 *as follows:*

$$\Delta_0 = \Delta[\iota_1 \mapsto \kappa_1, \ldots, \iota_k \mapsto \kappa_k] \backslash \{\iota_i \mapsto \kappa_i : 0 < i \leqslant k \;\&\; \iota_i \notin \mathrm{ftags}(B_0)\}$$

Now verify the assumptions of the induction step for B_0, Π_0, *and* Δ_0*:*

(1) Clear.
(2) Clear.
(3) $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta) \cup (\{b_1, \ldots, b_k\} \cap \mathrm{ftags}(B_0)) \subseteq \mathrm{ftags}(B) \cup \mathrm{ftags}(B_0) \subseteq \mathrm{ftags}(B_0)$.
(4) The assumption $\Delta \vdash B$ *implies* $\Delta[b_1 \mapsto \kappa_1, \ldots, b_k \mapsto \kappa_k] \vdash B_0$ *and thus* $\Delta_0 \vdash B_0$ *by strengthening.*

Thus by the induction hypothesis we have that $\Delta_0 \cong \Pi_0$. *Let* $\Delta_0 \cong \Gamma_0$ *via* Δ'_0. *We can find* Δ' *such that* $\Delta \cup \Delta' = \Delta_0 \cup \Delta'_0$ *and thus also* $\Delta \cong \Pi_0$. *Denote* $\Delta^+ = \Delta \cup \Delta'$. *It is easy to see that the principal type* Π *of* B *simply directly corresponds to the syntax tree of* B *because no rewriting rule can be applied to* B *standing alone. Also all form types contained in* Γ, *except of* $\varphi = a(b_1, \ldots, b_k)$ *which labels the only edge coming out of the root node* χ, *are contained in* Γ_0. *The member of* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *corresponding to the edge labeled by* φ *is identity because* $\Delta(\iota) = \uparrow[\kappa_0, \ldots, \kappa_k]$ *follows directly from the assumption* $\Delta \vdash B$. *All other members of* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *are present also in* $\mathrm{chtypes}(\Delta_0 \cup \Delta'_0, \Gamma_0)$ *and thus* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *is defined and identity. Thus* $\Delta \cong \Pi$.

$B = c\langle n_1, \ldots, n_k\rangle.B_0$**:** *Let* $\iota = \overline{c}$ *and* $\iota_i = \overline{n_i}$ *for* $0 < i \leqslant k$. *Let* $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ *be a principal type of* $(\!|B_0|\!)$. *Take* $\Delta_0 = \{(\iota' \mapsto \kappa) \in \Delta : \iota' \in \mathrm{ftags}(B_0)\}$. *Let us verify the assumptions of the induction step for* B_0, Π_0, *and* Δ_0.

(1) Clear.
(2) Clear.
(3) $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta) \cap \mathrm{ftags}(B_0) \subseteq \mathrm{ftags}(B_0)$.
(4) Here $\Delta_0 \vdash B_0$ *follows from* $\Delta \vdash B_0$ *by strengthening.*

By the induction hypothesis we obtain that $\Delta_0 \cong \Pi_0$. *Let* $\Delta_0 \cong \Gamma_0$ *via* Δ_0'. *Denote* $\Delta^+ = \Delta_0 \cup \Delta_0'$. *Now let us proof that* $\Delta \cong \Pi$. *It is easy to see that the principal type* Π *of* B *simply directly corresponds to the syntax tree of* B *because no rewriting rule can be applied to* B. *Also it is easy to see that all form types contained in* Γ, *except of* $\varphi = \iota(\iota_1, \ldots, \iota_k)$ *which labels the only edge coming out of the root node* χ, *are contained in* Γ_0. *The member of* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *corresponding to the edge labeled by* φ *is identity because* $\Delta(a) = \uparrow[\Delta(\iota_1), \ldots, \Delta(\iota_k)]$ *follows directly from the assumption* $\Delta \vdash P$. *All other members of* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *are present also in* $\mathrm{chtypes}(\Delta^+, \Gamma_0)$ *and thus* $\mathrm{chtypes}(\Delta^+, \Gamma)$ *is defined and identity. Thus the claim.*

$B = !B_0$**:** *Trivial.*

$B = (\nu n)B_0$**:** *Let* $\iota = \overline{n}$. *We know that* $\Delta \vdash B$. *Condition* (3) *implies* $\iota \notin \mathrm{dom}(\Delta)$. *Take*

$$\Delta_0 = \begin{cases} \Delta[\iota \mapsto \kappa] & \text{when } \iota \in \mathrm{ftags}(B_0) \\ \Delta & \text{when } \iota \notin \mathrm{ftags}(B_0) \end{cases}$$

We see that Π *is a principal type of* $(\!|B_0|\!)$ *as well. Let us verify the assumptions of the induction step for* B_0, Π, *and* Δ_0.

(1) Clear.
(2) Clear.
(3) When $\iota \in \mathrm{ftags}(B_0)$ *then* $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta) \cup \{a\} \subseteq \mathrm{ftags}(B) \cup \{a\} = \mathrm{ftags}(B_0)$. *When* $\iota \notin \mathrm{ftags}(B_0)$ *then* $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta) \subseteq \mathrm{ftags}(B) = \mathrm{ftags}(B_0)$.
(4) When $\iota \in \mathrm{ftags}(B_0)$ *then* $\Delta_0 \vdash B_0$ *follows from* $\Delta \vdash P$. *When* $\iota \notin \mathrm{ftags}(B_0)$ *then* $\Delta \vdash P$ *implies* $\Delta[\iota \mapsto \kappa] \vdash B_0$ *and thus* $\Delta_0 \vdash B_0$ *by strengthening.*

Thus by the induction hypothesis we have that $\Delta_0 \cong \Pi$. *Let* $\Delta_0 \cong \Gamma$ *via* Δ_0'. *It is easy to see that we can take* Δ' *such that* $\Delta \cup \Delta' = \Delta_0 \cup \Delta_0'$ *and thus the claim* $\Delta \cong \Pi$ *holds.* ∎

Proposition 15.0.3. *Let B be a π-process and let*

(1) Π_B *be a principal $\mathcal{P}$-type of $(\!|B|\!)$,*
(2) $\mathrm{ftags}(B) \subseteq \mathrm{dom}(\Delta)$*, and*
(3) $\Delta \cong \Pi_B$*.*

Then $\Delta \vdash B$*.*

Proof. *By induction on the structure of B. Let $\Pi = \langle \Gamma, \chi \rangle = \Pi_B$. Let*

$B = \mathtt{0}$**:** *Trivial.*

$B = B_0 \mid B_1$**:** *Let $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ be a principal type of $(\!|B_0|\!)$ and let $\Pi_1 = \langle \Gamma_1, \chi_1 \rangle$ be a principal type of $(\!|B_1|\!)$. It has to hold that any action type φ contained in Γ_0 (respectively Γ_1) as an edge label is also contained in Γ. Thus we have $\Delta \cong \Pi_0$ and $\Delta \cong \Pi_1$. It shows that the assumptions of the induction step are satisfied and by the induction hypothesis we have that $\Delta \vdash B_0$ and $\Delta \vdash B_1$ which proof the claim $\Delta \vdash B$.*

$B = c(n_1, \ldots, n_k).B_0$**:** *Let $\iota = \overline{c}$ and $\iota_i = \overline{n_i}$ for $0 < i \leqslant k$. Let $\Delta \cong \Gamma$ via Δ'. Denote $\Delta^+ = \Delta \cup \Delta'$. Take*

$$\Delta_0 = \Delta[\iota_1 \mapsto \Delta^+(\iota_1), \ldots, \iota_k \mapsto \Delta^+(\iota_k)]$$

It is easy to see that Δ_0 is defined. Let $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ be a principal type of $(\!|B_0|\!)$. Now let us verify the assumptions of the induction step for B_0, Π_0, and Δ_0.

(1) Clear.
(2) $\mathrm{ftags}(B_0) \subseteq \mathrm{ftags}(B) \cup \{\iota_1, \ldots, \iota_k\} \subseteq \mathrm{dom}(\Delta) \cup \{\iota_1, \ldots, \iota_k\} = \mathrm{dom}(\Delta_0)$*.*
(3) We can take Δ_0' such that $\Delta^+ = \Delta \cup \Delta' = \Delta_0 \cup \Delta_0'$. Note that Γ_0 could contain additional edges which are not in Γ. Those are edges introduced by the type inference algorithm to make the shape graph flow closed. But we can observe that whenever Γ_0 contains some flow edge labeled with type substitution $\mathfrak{o}$ such that $\mathfrak{o}(\iota') = \iota''$, then it has to hold that $\Delta^+(\iota') = \Delta^+(\iota'')$. Thus $\Delta_0 \cong \Pi_0$.

By the induction hypothesis we obtain $\Delta_0 \vdash B_0$. We see that it holds $\Delta(\iota) = \uparrow[\Delta^+(\iota_1), \ldots, \Delta^+(\iota_k)]$ and thus the claim.

$B = c\langle n_1, \ldots, n_k \rangle.B_0$**:** *Let $\iota = \overline{c}$ and $\iota_i = \overline{n_i}$ for $0 < i \leqslant k$. Let $\Delta \cong \Gamma$ via Δ'. Denote $\Delta^+ = \Delta \cup \Delta'$. Let $\Pi_0 = \langle \Gamma_0, \chi_0 \rangle$ be a principal type of $(\!|B_0|\!)$. Now let us verify the assumptions of the induction step for B_0, Π_0, and Δ.*

(1) Clear.

(2) $\mathrm{ftags}(B_0) \subseteq \mathrm{ftags}(B) \subseteq \mathrm{dom}(\Delta)$.

(3) To prove $\Delta \cong \Pi_0$ *use the same argument as in the proof of assumption* 3 *of the previous case concerning an input-binder.*

By the induction hypothesis we obtain $\Delta \vdash B_0$. *We see that* $\Delta \cong \Pi$ *implies* $\Delta(\iota) = \uparrow[\Delta(\iota_1), \ldots, \Delta(\iota_k)]$ *and thus the claim.*

$B = !B_0$**:** *Trivial.*

$B = (\nu n)B_0$**:** *Let* $\iota = \overline{n}$. *We can suppose that* $\iota \in \mathrm{ftags}(B_0)$ *because otherwise we can directly use the induction hypothesis (for* B_0, Π, *and* Δ*) and weakening to proof the claim. Let* $\Delta \cong \Gamma$ *via* Δ'. *Because* $\iota \in \mathrm{ftags}(B_0)$ *we have that there is* $\kappa = (\Delta \cup \Delta')(\iota)$. *Take* $\Delta_0 = \Delta[\iota \mapsto \kappa]$. *Let us verify the assumptions of the induction step.*

(1) Π *is a principal type of* $(\!|B_0|\!)$ *as well*

(2) $\mathrm{ftags}(B_0) = \mathrm{ftags}(B) \cup \{\iota\} \subseteq \mathrm{dom}(\Delta) \cup \{\iota\} = \mathrm{dom}(\Delta_0)$

(3) When $a \notin \mathrm{dom}(\Delta)$ *then* $\Delta_0 \cong \Gamma$ *via* $\Delta' \backslash \{\iota \mapsto \kappa\}$. *When* $\iota \in \mathrm{dom}(\Delta)$ *then* $\Delta = \Delta_0$.

By the induction hypothesis we have that $\Delta_0 \vdash B_0$ *and thus the claim.* ∎

Chapter 16

Shape Types for Mobile Ambients

This chapter shows how to instantiate Poly✶ to make a type system for Mobile Ambients [CG98] (Ma). Furthermore it proves this Poly✶ instantiation to be more expressive than an Ma type system from the literature [CG99] which we call Tma, shows how to embed Tma predicates in Poly✶ types, and discusses possible extensions of the embedding.

16.1 Mobile Ambients (Ma)

Mobile Ambients (Ma), introduced by Cardelli and Gordon [CG98], is a process calculus for representing process mobility. Processes are placed inside named bounded locations called *ambients* which form a tree hierarchy. Processes can change the hierarchy and send messages to nearby processes. Messages contain either ambient names or hierarchy change instructions. In order to simplify the presentation we build Ma processes from Meta✶ names with type tags preserved under α-renaming as in Meta✶.

Figure 16.1 describes Ma process syntax. Capabilities are ambient hierarchy change instructions. Executing a capability consumes it and instructs the surrounding ambient to change the hierarchy. The capability "in n" causes an ambient to move itself into a sibling ambient named n. The capability "out n" causes an ambient to move out of the parent ambient n and become its sibling The capability "open n" causes an ambient to dissolve the boundary of a child ambient n. Although the syntax allows an arbitrary N after capability name ("in N") so that substituting a capability for a name yields valid syntax, capabilities where N is not a single name are inert and meaningless. In capability sequences ($N.N'$), the left-most capability will be executed first.

The process constructors "0", "|", ".", "!", and "ν" have standard meanings. Binders contain explicit type annotations (Section 16.2 below). The expression $n[B]$ describes the process B running inside ambient n. As above, the syntax also allows

Syntax of MA *processes:*

$$
\begin{array}{rcll}
n, m & \in & \text{AName} & = \text{Name} \\
N & \in & \text{ACapability} & ::= \varepsilon \mid n \mid \mathsf{in}\ N \mid \mathsf{out}\ N \mid \mathsf{open}\ N \mid N.N' \\
\omega & \in & \text{AMsgType} & ::= \text{definition postponed to Fig. 16.3} \\
B & \in & \text{AProcess} & ::= \mathsf{0} \mid (B_0 \mid B_1) \mid N[\![B]\!] \mid N.B \mid !B \mid (\nu n{:}\omega)B \mid \\
 & & & \quad \langle N_1, \ldots, N_k \rangle \mid (n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B
\end{array}
$$

Structural Equivalence of MA*:*

$$\frac{}{B \equiv B} \qquad \frac{B_0 \equiv B_1}{B_1 \equiv B_0} \qquad \frac{B_0 \equiv B_1 \quad B_1 \equiv B_2}{B_0 \equiv B_2} \qquad \frac{B_0 \equiv B_1}{B_0 \mid B_2 \equiv B_1 \mid B_2}$$

$$\frac{B_0 \equiv B_1}{N[\![B_0]\!] \equiv N[\![B_1]\!]} \qquad \frac{B_0 \equiv B_1}{N.B_0 \equiv N.B_1} \qquad \frac{B_0 \equiv B_1}{!B_0 \equiv !B_1} \qquad \frac{B_0 \equiv B_1}{(\nu n{:}\omega)B_0 \equiv (\nu n{:}\omega)B_1}$$

$$\frac{B_0 \equiv B_1}{(n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B_0 \equiv (n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B_1} \qquad \frac{}{B_0 \mid B_1 \equiv B_1 \mid B_0}$$

$$\frac{}{B_0 \mid (B_1 \mid B_2) \equiv (B_0 \mid B_1) \mid B_2} \qquad \frac{}{\varepsilon.B \equiv B} \qquad \frac{}{!\mathsf{0} \equiv \mathsf{0}} \qquad \frac{}{B \mid \mathsf{0} \equiv B}$$

$$\frac{}{!B \equiv B \mid !B} \qquad \frac{}{B_0 \mid B_1 \equiv B_1 \mid B_0} \qquad \frac{}{(\nu n{:}\mathsf{Amb}[\![\kappa]\!])\mathsf{0} \equiv \mathsf{0}}$$

$$\frac{}{B_0 \mid (B_1 \mid B_2) \equiv (B_0 \mid B_1) \mid B_2} \qquad \frac{}{(N.N').B \equiv N.(N'.B)}$$

$$\frac{n \neq m}{(\nu n{:}\omega)(m[\![B]\!]) \equiv m[\![(\nu n{:}\omega)B]\!]} \qquad \frac{n \notin \mathrm{fn}(B_0)}{B_0 \mid (\nu n{:}\omega)B_1 \equiv (\nu n{:}\omega)(B_0 \mid B_1)}$$

$$\frac{n \neq m}{(\nu n{:}\omega_0)(\nu m{:}\omega_1)B \equiv (\nu m{:}\omega_1)(\nu n{:}\omega_0)B}$$

Figure 16.1: Syntax and structural equivalence of MA processes.

inert meaningless constructions with non-name N at the position of n. Capabilities can be communicated in messages. $\langle N_1, \ldots, N_k \rangle$ is a process that sends a k-tuple of messages. $(n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B$ is a process that receives a k-tuple of messages, substitutes them for appropriate n's in B, and continues as this new process. The name n_i is said to be (input-)bound in $(n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B$ and it comes with an explicit type annotation.

Bound type tags and free names of a process are defined like in META✶. Processes are identified up to α-conversion of bound names which preserves type tags. A substitution is a finite function from names to capabilities and its application to B is written postfix, for example, $B\{n \mapsto N\}$. Figure 16.1 defines also structural equivalence and Figure 16.2 describes semantics of MA processes. The only thing the semantics does with type annotations is copy them around. We set SpecialTag =

$$\frac{}{n[\mathsf{in}\ m.B_0 \mid B_1] \mid m[B_2] \to m[n[B_0 \mid B_1] \mid B_2]}$$

$$\frac{}{m[n[\mathsf{out}\ m.B_0 \mid B_1] \mid B_2] \to n[B_0 \mid B_1] \mid m[B_2]}$$

$$\frac{}{\mathsf{open}\ n.B_0 \mid n[B_1] \to B_0 \mid B_1}$$

$$\frac{}{(n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B \mid \langle N_1, \ldots, N_k \rangle \to B\{n_1 \mapsto N_1, \ldots, n_k \mapsto N_k\}}$$

$$\frac{B_0 \to B_1}{(\nu n{:}\omega)B_0 \to (\nu n{:}\omega)B_1} \qquad \frac{B_0 \to B_1}{n[B_0] \to n[B_1]} \qquad \frac{B_0 \to B_1}{B_0 \mid B_2 \to B_1 \mid B_2}$$

$$\frac{B_0' \equiv B_0 \quad B_0 \to B_1 \quad B_1 \equiv B_1'}{B_0' \to B_1'}$$

Figure 16.2: Semantics of Ma.

$\{\bullet, \mathsf{in}, \mathsf{out}, \mathsf{open}\}$ in order to prevent type tags with a special meaning to be bound. We require all processes to be well formed according to the following definition. Well-formedness can be achieved by name renaming if necessary and it is preserved by rewriting.

Definition 16.1.1. *A process B is* **well formed** *iff all the following hold.*

(S1) $\mathrm{ftags}(B) \cup \mathrm{itags}(B)$ *is disjoint with* $\mathrm{ntags}(B)$

(S2) *for* $(n_1 : \omega_1, \ldots, n_k : \omega_k).B_0$ *in* B, $\overline{n_i} \notin \mathrm{itags}(B_0)$ *and* $\overline{n_i} \neq \overline{n_j}$ *when* $i \neq j$

(S3) *bound names with the same type tag have the same type*

(S4) B *do not contain any type tags from* SpecialTag ∎

Example 16.1.2. *In this* Ma *process, packet ambient* p *delivers a synchronization message to destination ambient* d *by following instructions* x *received from the top level. As we have not yet properly defined message types, we only suppose* $\omega_\mathsf{p} = \mathsf{Amb}[\kappa]$ *for some* κ.

$$\begin{aligned} B = \ & \langle \mathsf{in\ d} \rangle \mid (\nu \mathsf{p}{:}\omega_\mathsf{p})(\mathsf{d}[\mathsf{open\ p}.0] \mid (\mathsf{x} : \omega_\mathsf{x}).\mathsf{p}[\mathsf{x}.\langle\rangle]) \quad \to \\ & (\nu \mathsf{p}{:}\omega_\mathsf{p})(\mathsf{d}[\mathsf{open\ p}.0] \mid \mathsf{p}[\mathsf{in\ d}.\langle\rangle]) \quad \to \\ & (\nu \mathsf{p}{:}\omega_\mathsf{p})(\mathsf{d}[\mathsf{open\ p}.0 \mid \mathsf{p}[\langle\rangle]]) \quad \to \\ & \mathsf{d}[\langle\rangle] \end{aligned}$$

This example is also used in the sections to follow. □

Syntax of TMA *types:*

$$\begin{array}{rcllcl} \omega & \in & \text{AMsgType} & ::= & \mathsf{Amb}[\kappa] \mid \mathsf{Cap}[\kappa] \\ \kappa & \in & \text{AExchangeType} & ::= & \mathsf{Shh} \mid \omega_1 \otimes \cdots \otimes \omega_k \\ \Delta & \in & \text{AEnvironment} & = & \text{TypeTag} \rightarrow_{\text{fin}} \text{AMsgType} \end{array}$$

Typing rules of TMA*:*

$$\frac{\Delta(n) = \omega}{\Delta \vdash n : \omega} \qquad \frac{}{\Delta \vdash \varepsilon : \mathsf{Cap}[\kappa]} \qquad \frac{\Delta \vdash N : \mathsf{Cap}[\kappa] \quad \Delta \vdash N' : \mathsf{Cap}[\kappa]}{\Delta \vdash N.N' : \mathsf{Cap}[\kappa]}$$

$$\frac{\Delta \vdash N : \mathsf{Amb}[\kappa']}{\Delta \vdash \mathsf{in}\ N : \mathsf{Cap}[\kappa]} \qquad \frac{\Delta \vdash N : \mathsf{Amb}[\kappa']}{\Delta \vdash \mathsf{out}\ N : \mathsf{Cap}[\kappa]} \qquad \frac{\Delta \vdash N : \mathsf{Amb}[\kappa]}{\Delta \vdash \mathsf{open}\ N : \mathsf{Cap}[\kappa]}$$

$$\frac{}{\Delta \vdash \mathsf{0} : \kappa} \qquad \frac{\Delta \vdash B : \kappa}{\Delta \vdash !B : \kappa} \qquad \frac{\Delta \vdash N : \mathsf{Cap}[\kappa] \quad \Delta \vdash B : \kappa}{\Delta \vdash N.B : \kappa}$$

$$\frac{\Delta \vdash N : \mathsf{Amb}[\kappa'] \quad \Delta \vdash B : \kappa'}{\Delta \vdash N[B] : \kappa} \qquad \frac{\Delta \vdash B_0 : \kappa \quad \Delta \vdash B_1 : \kappa}{\Delta \vdash B_0 \mid B_1 : \kappa}$$

$$\frac{\forall i\colon 0 < i \leqslant k \quad \Delta \vdash N_i : \omega_i}{\Delta \vdash \mathsf{<}N_1, \ldots, N_k\mathsf{>} : \omega_1 \otimes \cdots \otimes \omega_k} \qquad \frac{\Delta[n \mapsto \mathsf{Amb}[\kappa']] \vdash B : \kappa}{\Delta \vdash (\nu n\!:\!\mathsf{Amb}[\kappa'])B : \kappa}$$

$$\frac{\Delta[n_1 \mapsto \omega_1, \ldots, n_k \mapsto \omega_k] \vdash B : \omega_1 \otimes \cdots \otimes \omega_k}{\Delta \vdash (n_1\!:\!\omega_1, \ldots, n_k\!:\!\omega_k).B : \omega_1 \otimes \cdots \otimes \omega_k}$$

Figure 16.3: Syntax of TMA types and typing rules.

16.2 Types for Mobile Ambients (TMA)

An arity mismatch error, like in "<a, b>.0 | (x).in x.0", can occur in polyadic MA. Another communication error can be encountered when a sender sends a capability while a receiver expects a single name. For example "<in a>.0 | (x).out x.0" can rewrite to a meaningless "out (in a).0". Yet another error happens when a process is to execute a single name capability, like in "a.0". Processes which can never evolve to a state with any of the above errors are called *communication safe*. A typed MA introduced by Cardelli and Gordon [CG99], which we name TMA, verifies communication safety.

TMA assigns an allowed communication topic to each ambient location and Figure 16.3 describes TMA type syntax. Exchange types, which describe communication topics, are assigned to processes and ambient locations. The type Shh indicates silence (no communication). $\omega_1 \otimes \cdots \otimes \omega_k$ indicates communication of k-tuples of messages whose i-th member has the message type ω_i. For $k = 0$ we write $\mathbf{1}$ which allows only synchronization actions <> and (). Message types describe communication objects (names and capability sequences). Amb[κ] is the type of an ambient

where communication described by κ is allowed. $\mathsf{Cap}[\kappa]$ describes capabilities whose execution can unleash exchange κ (by opening some ambient). Environments assign message types to free names (via type tags). Figure 16.3 also describes the TMA typing rules. Types from conclusions not mentioned in the assumption can be arbitrary. For example, the type of $N[B]$ can be arbitrary provided B is well-typed. It reflects the fact that the communication inside N does not directly interact with N's outside. Subject reduction holds in TMA. When there are some Δ and κ such that Δ does not assign a Cap-type to any type tag, then $\Delta \vdash B : \kappa$ implies that B is communication safe. For more details about TMA see [CG99].

EXAMPLE 16.2.1. *Consider the process B from Example 16.1.2. Let us take*

$$\Delta = \{\mathsf{d} \mapsto \mathsf{Amb}[\mathbf{1}]\} \qquad \omega_{\mathsf{p}} = \mathsf{Amb}[\mathbf{1}] \qquad \omega_{\mathsf{x}} = \mathsf{Cap}[\mathbf{1}]$$

We can see that $\Delta \vdash B : \mathsf{Cap}[\mathbf{1}]$ but, for example, $\Delta \not\vdash B : \mathbf{1}$. □

16.3 Instantiation of META✶ to MA

When we omit type annotations, add "0" after output actions, and write capability prefixes always in a right associative manner (like "in a.(out b.(in c.0))"), we see that the MA syntax is included in the META✶ syntax. The following set $\mathcal{A}$ instantiates META✶ to MA.

$$\begin{aligned}\mathcal{A} = \big\{&\mathbf{active}\{\,\mathring{\mathtt{P}}\ \mathbf{in}\ \mathring{\mathtt{a}}[\mathring{\mathtt{P}}]\,\},\\ &\mathbf{rewrite}\{\,\mathring{\mathtt{a}}[\mathsf{in}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{b}}[\mathring{\mathtt{a}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}]\,\},\\ &\mathbf{rewrite}\{\,\mathring{\mathtt{a}}[\mathring{\mathtt{b}}[\mathsf{out}\ \mathring{\mathtt{a}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{a}}[\mathring{\mathtt{R}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}]\,\},\\ &\mathbf{rewrite}\{\,\mathsf{open}\ \mathring{\mathtt{a}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{a}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{P}} \mid \mathring{\mathtt{R}}\,\}\big\} \cup\\ \bigcup_{k=0}^{\infty} \big\{&\mathbf{rewrite}\{\,\mathtt{<}\mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{M}}_k\mathtt{>}.0 \mid (\mathring{\mathtt{a}}_1,\ldots,\mathring{\mathtt{a}}_k).\mathring{\mathtt{Q}} \hookrightarrow\ \{\mathring{\mathtt{a}}_1 := \mathring{\mathtt{M}}_1,\ldots,\mathring{\mathtt{a}}_k := \mathring{\mathtt{M}}_k\}\,\mathring{\mathtt{Q}}\,\}\ \big\}\end{aligned}$$

The **active** rule lets rewriting be done inside ambients. It corresponds to the MA rule "$B_0 \to B_1 \Rightarrow n[B_0] \to n[B_1]$". Each communication prefix length has its own rule as described in Section 10.3. $\mathcal{A}$ defines the calculus $C_{\mathcal{A}}$ and the type system $S_{\mathcal{A}}$.

Communication safety of P can be checked on an $\mathcal{A}$-type as follows. Two edges with the same source labeled by $(a_1,\ldots,a_k)$ and $\mathtt{<}b_1,\ldots,b_j\mathtt{>}$ with $k \neq j$ indicates an arity mismatch error (but only at active positions). Every label containing $\bullet$ (introduced by a substitution) indicates that a capability was sent instead of a name. Moreover, an edge labeled with a name $a \notin \mathrm{itags}(P)$ at active position indicates an execution of a single name capability. A type of P not indicating any error proves P's safety. Checking safety this way is easy.

$$\widetilde{N} = \begin{cases} n & \text{if } N = n \in \text{AName} \\ \bullet & \text{otherwise} \end{cases} \qquad (\!|\varepsilon|\!) = \mathsf{0} \qquad (\!|N_0.N_1|\!) = (\!|N_0|\!).(\!|N_1|\!)$$

$$(\!|\mathsf{in}\ N|\!) = \mathsf{in}\ \widetilde{N} \qquad (\!|\mathsf{out}\ N|\!) = \mathsf{out}\ \widetilde{N} \qquad (\!|\mathsf{open}\ N|\!) = \mathsf{open}\ \widetilde{N}$$

$$(\!|\mathsf{0}|\!) = \mathsf{0} \qquad (\!|(n_1{:}\omega_1, \ldots, n_k{:}\omega_k).B|\!) = (n_1, \ldots, x_k).(\!|B|\!)$$

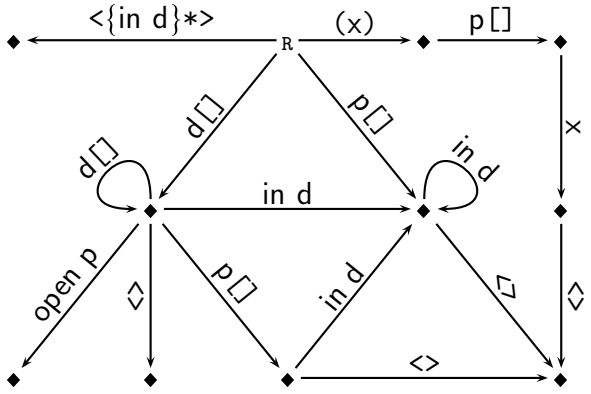
$$(\!|!B|\!) = !(\!|B|\!) \qquad (\!|\mathsf{<}N_1, \ldots, N_k\mathsf{>}|\!) = \mathsf{<}(\!|N_1|\!), \ldots, (\!|N_k|\!)\mathsf{>.0}$$

$$(\!|N.B|\!) = (\!|N|\!)_*(\!|B|\!) \qquad (\!|(\nu n : \omega)B|\!) = \nu n.(\!|B|\!)$$

$$(\!|N\,\mathsf{[}B\mathsf{]}|\!) = \widetilde{N}\,\mathsf{[}(\!|B|\!)\mathsf{]} \qquad (\!|B_0 \mid B_1|\!) = (\!|B_0|\!) \mid (\!|B_1|\!)$$

Figure 16.4: Encoding of TMA processes in META✶.

EXAMPLE 16.3.1. *Let us consider process B from Example 16.1.2 whose $C_{\mathcal{A}}$ equivalent is "P = <in d>.0 | νp.(d[open p.0] | (x).p[x.<>.0])". Its $S_{\mathcal{A}}$ principal type (with flow edges removed) has root* R *and the following shape graph.*



The names of non-root nodes are omitted. We can easily conclude that P is communication safe by simply checking the labels of edges as described above. □

16.4 Embedding of TMA in POLY✶

Using the notation from Section 13.1 we have that C is MA, S_C is TMA, predicates ρ are pairs (Δ, κ), and S_C's relation $\triangleright B : \rho$ is $\Delta \vdash B : \kappa$. Moreover $\mathcal{R}$ is $\mathcal{A}$ which was introduced with $C_{\mathcal{A}}$ and $S_{\mathcal{A}}$ in Section 16.3. The current section provides an embedding which shows how to, for a given B, Δ, and κ, answer the question $\Delta \vdash B : \kappa$ using $S_{\mathcal{A}}$. We stress that it is primarily a theoretical embedding for proving greater expressiveness which is not intended for use in practice.

Following the general discussion in Section 13.3 we need to provide an encoding $(\!|\cdot|\!)$ of MA processes in META✶. This encoding, presented in Figure 16.4, is straightforward due to the flexibility of META✶ syntax. The encoding $(\!|\cdot|\!)$ translates capabilities to META✶ messages and MA processes to META✶ processes. Meaningless expressions allowed by MA's syntax are translated using the auxiliary mapping $\widetilde{\cdot}$ and the special name "•". For example "$(\!|\mathsf{in}\ (\mathsf{out}\ \mathsf{a})|\!) = \mathsf{in}\ \bullet$". Recall that in META✶ "$x\,\mathsf{[}P\mathsf{]}$" is an abbreviation for "$x\,\mathsf{[]}.P$", and that "$_*$" linearizes composed messages

(like $((\mathsf{a}.\mathsf{b}).\mathsf{c})_* P) = \mathsf{a}.\mathsf{b}.\mathsf{c}.P$). The encoding erases type annotations; this is okay because TMA's rewriting rules only copy type annotations around without any other effect. The type embedding in Section 16.4 will recover type information by different means. Property 13.3.1 in the context given by MA and $\mathcal{A}$ becomes the following theorem.

THEOREM 16.4.1. *The following holds.*

$$\begin{array}{lll} B_0 \to B_1 & \textit{implies} & \exists B_0', B_1' \colon B_0 \equiv B_0' \;\&\; (\!|B_0'|\!) \overset{\mathcal{A}}{\hookrightarrow} (\!|B_1'|\!) \;\&\; B_1' \equiv B_1 \\ (\!|B_0|\!) \overset{\mathcal{A}}{\hookrightarrow} P_1 & \textit{implies} & \exists B_1 \colon (\!|B_1|\!) \equiv P_1 \;\&\; B_0 \to B_1 \end{array}$$

Because the TMA relation $\vdash$ is preserved under a consistent renaming of type tags of processes, we can not translate (Δ, κ) to a POLY$\star$ shape type with an equivalent meaning as discussed in Section 13.4. Nevertheless this becomes possible when we specify the sets of allowed input- and ν-bound type tags and their types. These can be easily extracted from a given process B. An environment Δ_B^ν (resp. Δ_B^{in}) from the top part of Figure 16.5 describes ν-bound (resp. input-bound) type tags of B. The definition reflects that ν-bound names in typable processes can only have Amb-types. For a given Δ, B, and κ we construct the shape type $(\!|\Delta \cup \Delta_B^\nu, \Delta_B^{\mathsf{in}}, \kappa|\!)$ such that $\Delta \vdash B : \kappa$ iff $\vdash (\!|B|\!) : (\!|\Delta \cup \Delta_B^\nu, \Delta_B^{\mathsf{in}}, \kappa|\!)$. The construction needs to know which names are input-bound and thus they are separated from the other names. The well-formedness rules S1-S4 ensure that there is no ambiguity in using only type tags to refer to typed names in a process. The type information I (Figure 16.5, 2nd part) collects what is needed to construct a shape type. For $I = (\Delta \cup \Delta_B^\nu, \Delta_B^{\mathsf{in}}, \kappa)$ we define Δ_I, Δ_I^{in}, and κ_I such that Δ_I describes types of all names in Δ and B, and Δ_I^{in} describes types of B's input-bound names, and κ_I is simply κ.

EXAMPLE 16.4.2. *Δ, B, and κ from the previous examples (Example 16.1.2 and Example 16.2.1) gives us $I = (\Delta \cup \Delta_B^\nu, \Delta_B^{\mathsf{in}}, \mathsf{Cap}[\mathbf{1}])$ and we have:*

$$\Delta \cup \Delta_B^\nu = \{\mathsf{d} \mapsto \mathsf{Amb}[\mathbf{1}], \mathsf{p} \mapsto \mathsf{Amb}[\mathbf{1}]\} \quad \Delta_I^{\mathsf{in}} = \{\mathsf{x} \mapsto \mathsf{Cap}[\mathbf{1}]\}$$

The main idea of the construction of the shape type $(\!|I|\!)$ from I is that $(\!|I|\!)$ contains exactly one node for every exchange type of some ambient location, that is, one node for the top-level type κ_I, and one node for κ' whenever $\mathsf{Amb}[\kappa']$ is in I. The top-level type corresponds to the shape type root. Each node corresponding to some κ has self-loops which describe all capabilities and communication actions which a process of the type κ can execute. When $\Delta_I(\mathsf{d}) = \mathsf{Amb}[\mathbf{1}]$ then every node would have a self-loop labeled by "$\mathsf{in}\ \mathsf{d}$" because in-capabilities can be executed by any process. On the other hand only the node corresponding to $\mathbf{1}$ would allow "$\mathsf{open}\ \mathsf{d}$" because only processes of type $\mathbf{1}$ can legally execute it. Finally, following an edge labeled with "$\mathsf{d}[]$" means entering d. Thus the edge has led to the node

Extraction of types of bound names:

$$\Delta^{\mathsf{in}}_B(\iota) = \omega \quad \text{iff} \quad B \text{ has a subprocess } (\ldots, a^\iota : \omega, \ldots).B_0$$
$$\Delta^{\nu}_B(\iota) = \omega \quad \text{iff} \quad \omega = \mathsf{Amb}[\kappa] \;\&\; B \text{ has a subprocess } (\nu a^\iota{:}\omega)B_0$$

Type information:

$$I \in \text{TypeInfo} = \text{AEnvironment} \times \text{AEnvironment} \times \text{AExchangeType}$$

Let $I = (\Delta_0, \Delta_1, \kappa)$. Write Δ_I for $\Delta_0 \cup \Delta_1$, and Δ^{in}_I for Δ_1, and κ_I for κ.

Set of nodes of a shape graph (and correspondence functions):

$$\text{types}_I = \{\kappa_I\} \cup \{\kappa\colon \mathsf{Amb}[\kappa] \in \text{rng}(\Delta_I)\} \qquad \text{nodeof}_I = \text{typeof}_I^{-1}$$

Let nodes_I be an arbitrary but fixed set of nodes such that there exist the bijection typeof_I from nodes_I into types_I.

Form types describing legal capabilities:

$$\text{namesof}_I(\omega) = \{\iota\colon \Delta_I(\iota) = \omega\}$$
$$\text{allowedin}_I(\kappa) = \text{moves}_I \cup \text{opens}_I(\kappa) \cup \text{comms}_I(\kappa)$$
$$\text{moves}_I = \{\mathsf{in}\ \iota, \mathsf{out}\ \iota\colon \exists\kappa.\, \iota \in \text{namesof}_I(\mathsf{Amb}[\kappa])\}$$
$$\text{opens}_I(\kappa) = \{\mathsf{open}\ \iota\colon \iota \in \text{namesof}_I(\mathsf{Amb}[\kappa])\} \cup \text{namesof}_I(\mathsf{Cap}[\kappa])$$
$$\text{msgs}_I(\mathsf{Amb}[\kappa]) = \text{namesof}_I(\mathsf{Amb}[\kappa])$$
$$\text{msgs}_I(\mathsf{Cap}[\kappa]) = \text{namesof}_I(\mathsf{Cap}[\kappa]) \cup \{(\text{moves}_I \cup \text{opens}_I(\kappa))*\}$$
$$\text{comms}_I(\mathsf{Shh}) = \emptyset$$
$$\text{comms}_I(\omega_1 \otimes \cdots \otimes \omega_k) = \{\langle \mu_1, \ldots, \mu_k \rangle\colon \mu_i \in \text{msgs}_I(\omega_i)\} \cup \{(\iota_1, \ldots, \iota_k)\colon \Delta^{\mathsf{in}}_I(\iota_i) = \omega_i \;\&\; (i \neq j \Rightarrow \iota_i \neq \iota_j)\}$$

Construction of shape predicates and embedding of type judgments:

$$\langle\!|I|\!\rangle = \{\chi \xrightarrow{\varphi} \chi\colon \varphi \in \text{allowedin}_I(\text{typeof}_I(\chi)) \;\&\; \chi \in \text{nodes}_I\} \cup \{\chi \xrightarrow{\iota[]} \chi'\colon \iota \in \text{namesof}_I(\mathsf{Amb}[\text{typeof}_I(\chi')]) \;\&\; \chi, \chi' \in \text{nodes}_I\}$$
$$[\![I]\!] = \langle \langle\!|I|\!\rangle, \text{nodeof}_I(\kappa_I)\rangle$$

Figure 16.5: Embedding of TPI in POLY✶.

χ_{d} that corresponds to $\mathbf{1}$. In the above example with $\Delta_I(\mathsf{d}) = \mathsf{Amb}[\mathbf{1}]$, the shape graph would contain edges labeled with "$\mathsf{d}[]$" from any node to χ_{d}.

The construction starts by building the node set of a shape predicate (Figure 16.5, 3rd part). All the exchange types of ambient locations are gathered in the set types_I. These types are put in bijective correspondence with the set nodes_I.

EXAMPLE 16.4.3. *Our example gives us* $\text{types}_I = \{\mathsf{Cap}[\mathbf{1}], \mathbf{1}\}$. *Let us choose* $\text{nodes}_I = \{\mathtt{R}, \mathtt{1}\}$ *and define the bijections such that* $\text{nodeof}_I(\mathsf{Cap}[\mathbf{1}]) = \mathtt{R}$ *and* $\text{nodeof}_I(\mathbf{1}) = \mathtt{1}$. □

The 4th part of Figure 16.5 defines some auxiliary functions. The set $\text{namesof}_I(\omega)$ contains all type tags declared with the type ω by I. The set $\text{allowedin}_I(\kappa)$ contains all POLY✶ form types which describe (translations of) all capabilities and communication action prefixes which are allowed to be legally executed by a process of the type κ. The set $\text{allowedin}_I(\kappa)$ consists of three parts: moves_I, $\text{opens}_I(\kappa)$, and

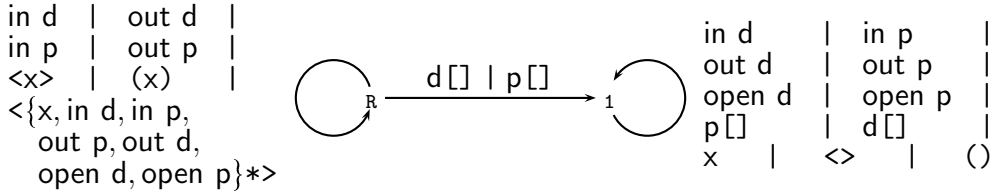
$\mathrm{comms}_I(\kappa)$. The form types in moves_I describe all in/out capabilities constructible from ambient type tags in I. The set does not depend on κ because in/out capabilities can be executed by any process. The set $\mathrm{opens}_I(\kappa)$ describe open-capabilities which can be executed by a process of the type κ. It consists of open-capabilities constructible from ambient names in I and from those type tags which have the type $\mathsf{Cap}[\kappa]$ in I. The second part of $\mathrm{opens}_I(\kappa)$ describes names of the type $\mathsf{Cap}[\kappa]$ which might be instantiated to some executable capabilities. The set $\mathrm{comms}_I(\kappa)$ describes communication actions which can be executed by a process of the type κ. Its first part describes output- and the second input-actions. The auxiliary set $\mathrm{msgs}_I(\omega)$ describes all capabilities (sometimes called messages) of the type ω constructible from names in I.

EXAMPLE 16.4.4. *Relevant sets for our example are as follows.*

$$\begin{array}{ll}
\mathrm{namesof}_I(\mathsf{Amb}[\mathbf{1}]) = \{\mathsf{d},\mathsf{p}\} & \mathrm{opens}_I(\mathbf{1}) = \{\mathsf{open\ d},\mathsf{open\ p},\mathsf{x}\} \\
\mathrm{namesof}_I(\mathsf{Cap}[\mathbf{1}]) = \{\mathsf{x}\} & \mathrm{opens}_I(\mathsf{Cap}[\mathbf{1}]) = \emptyset \\
\mathrm{comms}_I(\mathbf{1}) = \{\mathsf{<>},\mathsf{()}\} & \mathrm{moves}_I = \{\mathsf{in\ d},\mathsf{in\ p},\mathsf{out\ d},\mathsf{out\ p}\} \\
\multicolumn{2}{l}{\mathrm{comms}_I(\mathsf{Cap}[\mathbf{1}]) = \{\mathsf{<x>},\mathsf{<\{in\ d},\mathsf{in\ p},\mathsf{out\ d},\mathsf{out\ p},\mathsf{open\ d},\mathsf{open\ p},\mathsf{x\}*>},\mathsf{(x)}\}}
\end{array}$$

The bottom part of Figure 16.5 constructs the shape graph $\langle\!|I|\!\rangle$ and the shape predicate $[\!|I|\!]$ from I. The first part of $\langle\!|I|\!\rangle$ describes self-loops of χ which describe actions allowed to be executed by a process of $\mathrm{typeof}_I(\chi)$. The second part of $\langle\!|I|\!\rangle$ describe transitions among nodes. Any edge labeled by "a[]" always leads to the node which corresponds to the exchange type allowed inside a.

EXAMPLE 16.4.5. *The resulting shape predicate* $[\!|I|\!] = \langle\Gamma, \mathtt{R}\rangle$ *in our example has the root* R *and its shape graph* Γ *is below. We merge edges with the same source and destination into one using "|".*



Correctness of the translation is expressed by Theorem 16.4.6. The assumptions ensure that no ν-bound name is mentioned by Δ or has a Cap-type assigned by an annotation.

THEOREM 16.4.6. *Let* $\mathrm{dom}(\Delta) \cap \mathrm{ntags}(B) = \emptyset$ *and* $\mathrm{dom}(\Delta^\nu_B) = \mathrm{ntags}(B)$ *then*

$$\Delta \vdash B : \kappa \quad \textit{iff} \quad \vdash [\!|B|\!] : \langle\!|(\Delta \cup \Delta^\nu_B, \Delta^{\mathsf{in}}_B, \kappa)|\!\rangle$$

THEOREM 16.4.7. $\langle\!|(\Delta \cup \Delta^\nu_B, \Delta^{\mathsf{in}}_B, \kappa)|\!\rangle$ *is an $\mathcal{A}$-type without flow edges, that is, it can be completed to an $\mathcal{A}$-type by adding only flow edges.* ∎

16.5 Conclusions and Further Possibilities

We embedded TMA's typing relation in $S_{\mathcal{A}}$ (Section 16.4) and showed how to recognize communication safety in $S_{\mathcal{A}}$ directly (Section 16.3). The type $\langle\!|I|\!\rangle$ constructed in Section 16.4 can also be used to prove the safety of B. But then, it follows from the properties of principal types, that the safety of B can be recognized directly from its principal $\mathcal{A}$-type. Thus any process proved safe by TMA can be proved safe by $S_{\mathcal{A}}$ on its own.

Some processes are recognized safe by $S_{\mathcal{A}}$ but not by TMA. For example, "(x : ω).x.0 | <in a>" is not typable in TMA but it is trivially safe. Another examples show polymorphic abilities of shape types, for example, the $C_{\mathcal{A}}$ process

$$\begin{array}{llllll} !(x,y,m).x[\text{in } y.\langle m\rangle.0] & | & \langle p,a,c\rangle.0 & | & a[\text{open } p.0] & | \\ & & \langle q,b,\text{in } a\rangle.0 & | & b[\text{open } q.0] & \end{array}$$

can be proved safe by POLY$\star$ but it constitutes a challenge for TMA-like non-polymorphic type systems. We are not aware of other type systems for MA and its successor that can handle this kind of polymorphism.

The expressiveness of shape types $\langle\!|I|\!\rangle$ from Section 16.4 can be improved. In subsequent work [CGG99], Cardelli, Ghelli, and Gordon define a type system which can ensure that some ambients stay immobile or that their boundaries are never dissolved. This can be achieved easily by removing appropriate self loops of nodes. We can also assign nodes to (groups of) ambients instead of to exchange types. This gives us similar possibilities as another TMA successor [CGG00]. Moreover, we can use shape type polymorphism to express location-dependent properties of ambients, like that ambient a can be opened only inside ambient b.

Chapter 17

Details of the TMA Embedding

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

17.1 Faithfulness of MA Encoding in META✶

In this thesis we provide proof of Property 13.3.1 only for the case of MA. Proof of Property 13.3.1 for the π-calculus and BA are analogous. The proof for MA is the most complicated because the encoding of MA processes in META✶ is the less straightforward. All important ideas of the proofs for the π-calculus and BA are shown on the proof for MA.

We use the following names for the rewriting rules of MA from Figure 16.2 in the left-right and top-down order: AIN, AOUT, AOPEN, ACOM, ANU, AAMB, APAR, and ASTR. The following lemma states the relationship between META✶ and TMA substitutions.

LEMMA 17.1.1. *It holds that*

$$(\!|B\{n_1 \mapsto N_1, \ldots, n_k \mapsto N_k\}|\!) = \overline{\{n_1 \mapsto (\!|N_1|\!), \ldots, n_k \mapsto (\!|N_k|\!)\}}((\!|B|\!))$$

PROOF. *By induction on the structure of* B. ∎

The following proposition is the left-to-right implication of Property 13.3.1 for MA. Additionally we let C range over MA processes AProcess in the proof.

PROPOSITION 17.1.2. *Let* $B_0 \to B_1$. *Then there exist* B_0' *and* B_1' *such that*

$$B_0 \equiv B_0' \;\&\; (\!|B_0'|\!) \overset{\mathcal{A}}{\hookrightarrow} (\!|B_1'|\!) \;\&\; B_1' \equiv B_1$$

PROOF. *By induction on the derivation of* $B_0 \to B_1$. *Let it be derived by*

(AIN): *Then, for some* n, m, C_0, C_1, *and* C_1 *we have* $B_0 = n[\mathsf{in}\ m.C_0 \mid C_1] \mid m[C_1]$ *and* $B_1 = m[n[C_0 \mid C_1] \mid C_1]$. *Take the instantiation* $\mathbb{P} = \{\mathring{\mathtt{a}} \mapsto n, \mathring{\mathtt{b}} \mapsto m, \mathring{\mathtt{P}} \mapsto (\!|C_0|\!), \mathring{\mathtt{Q}} \mapsto (\!|C_1|\!), \mathring{\mathtt{R}} \mapsto (\!|C_1|\!)\}$. *Now, we know that* $\mathbf{rewrite}\{\, \mathring{\mathtt{a}}[\mathsf{in}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{b}}[\mathring{\mathtt{a}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}] \,\} \in \mathcal{A}$. *Moreover it is easy to see that* $(\!|B_0|\!) = \mathbb{P}[\![\mathring{\mathtt{a}}[\mathsf{in}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{R}}]]\!]$ *and* $(\!|B_1|\!) = \mathbb{P}[\![\mathring{\mathtt{b}}[\mathring{\mathtt{a}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}]]\!]$. *Take directly* $B_0' = B_0$ *and* $B_1' = B_1$ *and we have* $(\!|B_0'|\!) \overset{\mathcal{A}}{\hookrightarrow} (\!|B_1'|\!)$ *by* RREW.

(AOUT): *Like case* AIN.

(AOPEN): *Like case* AIN.

(ACOM): *Similarly to case* AIN *but with the following changes.*

$$
\begin{aligned}
&B_0 = (n_1{:}\omega_1, \ldots, n_k{:}\omega_k).C \mid \texttt{<}N_1, \ldots, N_k\texttt{>} \\
&B_1 = C\{n_1 \mapsto N_1, \ldots, n_k \mapsto N_k\} \\
&\mathbb{P} = \{\mathring{\mathtt{a}}_1 \mapsto n_1, \ldots, \mathring{\mathtt{a}}_k \mapsto n_k, \mathring{\mathtt{M}}_1 \mapsto (\!|N_1|\!), \ldots, \mathring{\mathtt{M}}_k \mapsto (\!|N_k|\!), \mathring{\mathtt{P}} \mapsto \mathtt{0}, \mathring{\mathtt{Q}} \mapsto (\!|C|\!)\} \\
&P_0 = \mathbb{P}[\![(\mathring{\mathtt{a}}_1, \ldots, \mathring{\mathtt{a}}_k).\mathring{\mathtt{Q}} \mid \texttt{<}\mathring{\mathtt{M}}_1, \ldots, \mathring{\mathtt{M}}_k\texttt{>}.\mathring{\mathtt{P}}]\!] = \\
&\quad = (n_1, \ldots, n_k).(\!|C|\!) \mid \texttt{<}(\!|N_1|\!), \ldots, (\!|N_k|\!)\texttt{>}.\mathtt{0} \\
&P_1 = \mathbb{P}[\![\mathring{\mathtt{P}} \mid \{\mathring{\mathtt{a}}_1 := \mathring{\mathtt{M}}_1, \ldots, \mathring{\mathtt{a}}_k := \mathring{\mathtt{M}}_k\}\, \mathring{\mathtt{Q}}]\!] = \mathtt{0} \mid \{n_1 \mapsto (\!|N_1|\!), \ldots, n_k \mapsto (\!|N_k|\!)\}(\!|C|\!) \\
&B_0' = B_0 \\
&B_1' = \mathtt{0} \mid B_1
\end{aligned}
$$

We have $(\!|B_0'|\!) = P_0$ *directly and* $(\!|B_1'|\!) = P_1$ *by Lemma 17.1.1. By* TMA *structure equivalence we have* $B_0 \equiv B_0'$ *and* $B_1 \equiv B_1'$. *Thus* $(\!|B_0'|\!) \overset{\mathcal{A}}{\hookrightarrow} (\!|B_1'|\!)$ *by* RREW.

(AAMB): *Here simply use the induction hypothesis and then instantiate the rule* $\mathbf{active}\{\, \mathring{\mathtt{P}}\ \mathbf{in}\ \mathring{\mathtt{a}}[\mathring{\mathtt{P}}] \,\}$ *in* RACT *by* $\mathbb{P} = \{\mathring{\mathtt{a}} \mapsto n\}$ *where* n *is the ambient name obtained from the assumptions. Here we have to verify that* $n \neq \bullet$ *which is clear because* $\bullet$ *is forbidden to be used by* TMA *processes.*

(ANU): *Again use the induction hypothesis and verify that* ν*-bound type tag* $\overline{n}$ *is not in* $\mathrm{tags}(\mathcal{A}) = \{\mathsf{in}, \mathsf{out}, \mathsf{open}, \mathtt{[]}\}$. *This is satisfied for these name are excluded from* AName. *Then use* RNU *to prove the claim.*

(APAR): *Use the induction hypothesis and* RPAR *to prove the claim.*

(ASTR): *Use the induction hypothesis and* RSTR *to prove the claim.*

The following proposition is the right-to-left implication of Property 13.3.1 for TMA.

PROPOSITION 17.1.3. *Let* $(\!|B_0|\!) \overset{\mathcal{A}}{\hookrightarrow} P_1$. *Then there exists some* B_1 *such that* $(\!|B_1|\!) \equiv P_1$ *and* $B_0 \to B_1$.

PROOF. *By induction on the derivation of* $(\!|B_0|\!) \overset{\mathcal{A}}{\hookrightarrow} P_1$. *Let it be derived by*

(RREW): *using the rule*

(1) $\mathbf{rewrite}\{\, \mathring{P} \hookrightarrow \mathring{Q} \,\} = \mathbf{rewrite}\{\, \mathring{\mathtt{a}}[\mathsf{in}\ \mathring{\mathtt{b}}.\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{b}}[\mathring{\mathtt{R}}] \hookrightarrow \mathring{\mathtt{b}}[\mathring{\mathtt{a}}[\mathring{\mathtt{P}} \mid \mathring{\mathtt{Q}}] \mid \mathring{\mathtt{R}}] \,\}$. *We also know that there is some instantiation* $\mathbb{P}$ *with all the variables mentioned by the rule in its range. We define* $x = \mathbb{P}[\![\mathring{\mathtt{a}}]\!]$, $y = \mathbb{P}[\![\mathring{\mathtt{b}}]\!]$, $P'_0 = \mathbb{P}[\![\mathring{\mathtt{P}}]\!]$, $P'_1 = \mathbb{P}[\![\mathring{\mathtt{Q}}]\!]$, $P'_2 = \mathbb{P}[\![\mathring{\mathtt{R}}]\!]$. *Now we can deduce that* $(\!|B_0|\!) = x[\mathsf{in}\ y.P'_0 \mid P'_1] \mid y[P'_2]$ *and* $P_1 = y[x[P'_0 \mid P'_1] \mid P'_2]$. *Now there have to be* B'_0, B'_1, B'_2 *such that* $(\!|B'_0|\!) = P'_0$, $(\!|B'_1|\!) = P'_1$, $(\!|B'_2|\!) = P'_2$, *and* $B_0 \equiv x[\mathsf{in}\ y.B'_0 \mid B'_1] \mid y[B'_2]$ *It holds that both* x *and* y *are in* AName *because (1)* $\mathbb{P}$ *can not map a name variable to* $\bullet$ *and (2)* $\mathsf{in}, \mathsf{out}, \mathsf{open}, []$ *can not appear in* B_0. *Now we just take* $B_1 = y[x[B'_0 \mid B'_1] \mid B'_2]$ *and thus we have* $(\!|B_1|\!) = P_1$. *Finally we proof* $B_0 \to B_1$ *by* AIn *and* AStr.

(2) Proof for the other three rules (out, open, *and the communication one) is similar as case (1).*

(RAct): *using the rule* $\mathbf{active}\{\, \mathring{\mathtt{P}}\ \mathbf{in}\ \mathring{\mathtt{a}}[\mathring{\mathtt{P}}] \,\}$. *Let* $x = \mathbb{P}[\![\mathring{\mathtt{a}}]\!]$. *In this case we have that there are some* P *and* Q *such that* $P \overset{\mathcal{A}}{\hookrightarrow} Q$. *We also have that* $(\!|B_0|\!) = x[P]$ *and* $P_1 = x[Q]$. *Thus we see that there is some* B'_0 *such that* $(\!|B'_0|\!) = P$ *and* $B_0 = x[B'_0]$. *It also implies that* $x \in$ AName. *Thus we obtain* $(\!|B'_0|\!) \overset{\mathcal{A}}{\hookrightarrow} Q$ *and by the induction hypothesis we have that there exists* B'_1 *such that* $(\!|B'_1|\!) \equiv Q$ *and* $B'_0 \to B'_1$. *Take* $B_1 = x[B'_1]$. *We have* $(\!|B_1|\!) = x[(\!|B'_1|\!)] \equiv x[Q] = P_1$. *Finally* $B_0 \to B_1$ *by* AAmb.

(RNu): *Thus there are* x, P, *and* Q, *such that* $(\!|B_0|\!) = \nu x.P$ *and* $P_1 = \nu x.Q$, *and* $P \overset{\mathcal{A}}{\hookrightarrow} Q$. *Here we see that* $x \in$ AName *and thus* $x \notin \mathrm{fn}(\mathcal{A})$. *From* $(\!|B_0|\!) = \nu x.P$ *we can conclude that there are some* ω *and* B'_0 *such that* $B_0 = (\nu n : \omega)B'_0$ *and* $(\!|B'_0|\!) = P$. *Thus we have* $(\!|B'_0|\!) \overset{\mathcal{A}}{\hookrightarrow} Q$ *and by the induction hypothesis we obtain that there exists* B'_1 *such that* $(\!|B'_1|\!) \equiv Q$ *and* $B'_0 \to B'_1$. *Let us take* $B_1 = (\nu x : \omega)B'_1$. *Now* $(\!|B_1|\!) = \nu x.(\!|B'_1|\!) \equiv \nu x.Q = P_1$. *Finally* $B_0 \to B_1$ *by* ANu.

(RPar): *Proof is similar to case* RNu.

(RStr): *The problem to deal with in this case is the difference in structural equivalences of* Meta✶ *and* Tma, *in particular, the* Meta✶ *rule present which allows a* ν*-binder to skip an arbitrary action. On contrary* Tma *allows* ν*-binders to skip ambient boundaries only. For example for* $B_0 = ().(\nu \mathsf{a} : \omega)\mathsf{in}\ \mathsf{a}.0$ *and* $B_1 = (\nu \mathsf{a} : \omega)().\mathsf{in}\ \mathsf{a}.0$ *we have* $(\!|B_0|\!) \equiv (\!|B_1|\!)$ *in* Meta✶ *but not* $B_0 \equiv B_1$ *in* Tma. *The key observation here is that whenever in* Meta✶ *some rewriting is inferred by* RStr *using* Meta✶ *structural equivalence in a way that is not allowed in* Tma *then the same rewriting statement can be inferred in* Meta✶ *using a derivation that uses structural equivalence only in a* Tma*-compatible way. Then is a simple application of the induction hypothesis.* ∎

17.2 Correctness of TMA Embedding in POLY✶

Proposition 17.2.1 is the left-to-right implication of Theorem 16.4.6 and Proposition 17.2.2 is the right-to-left implication. The assumption of the propositions that $\Delta(\iota) = \Delta^{\nu}_{B}(\iota)$ for every $\iota \in \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$ follows from the assumption of Theorem 16.4.6 that $\mathrm{dom}(\Delta) \cap \mathrm{ftags}(B) = \emptyset$.

PROPOSITION 17.2.1. *Let* $\Delta(\iota) = \Delta^{\nu}_{B}(\iota)$ *for every* $\iota \in \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$. *Then*

$$\Delta \vdash B : \kappa \quad \textit{implies} \quad \vdash (\!|B|\!) : \langle\!|(\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa)|\!\rangle$$

PROOF. *Let* $\Delta \vdash B : \kappa$. *Let* $I = (\Delta \cup \Delta^{\nu}_{B}, \Delta^{\mathsf{in}}_{B}, \kappa)$ *and* $\Pi = \langle\!|I|\!\rangle$. *Prove* $\vdash (\!|B|\!) : \Pi$ *by induction on the structure of* B. *Let*

$B = \mathsf{0}$**:** *Clear.*

$B = (B_0 \mid B_1)$**:** *We know that* $\Delta \vdash B_0 : \kappa$. *We see that* $\mathrm{ntags}(B_0) \subseteq \mathrm{ntags}(B)$. *Thus the assumption of the induction step for* B_0 *is satisfied. Let* $\Pi_0 = \langle\!|(\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\rangle$. *By the induction hypothesis we obtain* $\vdash (\!|B_0|\!) : \Pi_0$. *Because* $\mathrm{itags}(B_0) \subseteq \mathrm{itags}(B)$ *and* $\mathrm{ntags}(B_0) \subseteq \mathrm{ntags}(B)$ *we see that* Π *contains all the edges of* Π_0. *Thus also* $\vdash (\!|B_0|\!) : \Pi$ *by weakening. Similarly we obtain* $\vdash (\!|B_1|\!) : \Pi$. *Thus the claim.*

$B = N\,\mathsf{[}B_0\mathsf{]}$**:** *We know that there is some* κ' *such that* $\Delta \vdash N : \mathsf{Amb[}\kappa'\mathsf{]}$ *and* $\Delta \vdash B_0 : \kappa'$. *Thus it is clear that there is some* n *such that* $N = n$. *Let* $\iota = \overline{n}$. *We have* $\mathrm{ntags}(B_0) = \mathrm{ntags}(B)$ *and thus the assumption of the induction step for* B_0 *and* κ' *is satisfied. Let* $\Pi_0 = \langle\!|(\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa')|\!\rangle$. *By the induction hypothesis we obtain* $\vdash (\!|B_0|\!) : \Pi_0$. *We see that* $\Delta^{\mathsf{in}}_{B_0} = \Delta^{\mathsf{in}}_{B}$ *and* $\Delta^{\nu}_{B_0} = \Delta^{\nu}_{B}$ *and thus* Π_0 *and* Π *differ only in the root nodes. Let* χ *be the root node of* Π *and let* χ_0 *be the root node of* Π_0. *It is clear that* $\chi = \mathrm{nodeof}_I(\kappa)$ *and* $\chi_0 = \mathrm{nodeof}_I(\kappa')$. *(Note that for* $I_0 = (\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa')$ *it does not need to hold that* $\kappa \in \mathrm{types}_{I_0}$.*)* *We can see that* $\Delta(a) = \mathsf{Amb[}\kappa'\mathsf{]}$ *and thus* $\iota \in \mathrm{namesof}_I(\mathsf{Amb[}\kappa'\mathsf{]})$. *Thus* $(\chi \xrightarrow{\iota\mathsf{[]}} \chi_0) \in \Gamma$. *Hence the claim* $\vdash n\,\mathsf{[}(\!|B_0|\!)\mathsf{]} : \Pi$ *because* $\vdash n\,\mathsf{[]} : \iota\,\mathsf{[]}$.

$B = N.B_0$**:** *We see that* $\Delta \vdash N : \mathsf{Cap[}\kappa\mathsf{]}$ *and* $\Delta \vdash B_0 : \kappa$. *The assumption of the induction step for* B_0 *is satisfied because* $\mathrm{ntags}(B_0) = \mathrm{ntags}(B)$. *Let* $\Pi_0 = \langle\!|(\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\rangle$. *By the induction hypothesis we obtain* $\vdash (\!|B_0|\!) : \Pi_0$. *Clearly* Π *contains all the edges as* Π_0 *and thus also* $\vdash (\!|B_0|\!) : \Pi$ *by weakening. Let* $\chi = \mathrm{nodeof}_I(\kappa)$ *be the root node of* Π. *We see that* χ *is the root of* Π_0 *as well. Let us prove the claim by induction on the structure of* N. *(We are proving that* $\vdash (\!|B_0|\!) : \Pi$ *and* $\Delta \vdash N : \kappa$ *implies* $\vdash (\!|N.B_0|\!) : \Pi$.*)*

$N = n$**:** *Let* $\iota = \overline{n}$. *Then it is clear that* $\Delta(\iota) = \mathsf{Cap[}\kappa\mathsf{]}$. *Thus it holds that* $\iota \in \mathrm{namesof}_I(\mathsf{Cap[}\kappa\mathsf{]})$ *and* $\iota \in \mathrm{opens}_I(\kappa)$. *Now we see that* Π *contains the edge* $\chi \xrightarrow{\iota} \chi$ *and because* $\vdash n : \iota$ *we see that* $\vdash n.(\!|B_0|\!) : \Pi$. *Thus the claim.*

$N = \mathsf{in}\ N$**:** *We see that there is some* κ' *such that* $\Delta \vdash N : \mathsf{Amb}[\kappa']$ *and thus there is some* n *such that* $N = n$*. Let* $\iota = \overline{n}$*. We have* $\Delta(a) = \mathsf{Amb}[\kappa']$ *and thus* $\iota \in \mathrm{namesof}_I(\mathsf{Cap}[\kappa'])$ *and* $\mathsf{in}\ \iota \in \mathrm{moves}_I$*. Now we see that* Π *contains the edge* $\chi \xrightarrow{\mathsf{in}\ \iota} \chi$ *and because* $\vdash \mathsf{in}\ n : \mathsf{in}\ \iota$ *we see that* $\vdash \mathsf{in}\ n.(\!|B_0|\!) : \Pi$*. Thus the claim.*

$N = \mathsf{out}\ N$**:** *As in the case for "*$\mathsf{in}\ N$*".*

$N = \mathsf{open}\ N$**:** *As in the case for "*$\mathsf{in}\ N$*" but here* $\kappa' = \kappa$ *and* $\mathsf{open}\ \iota \in \mathrm{opens}_I(\kappa)$*.*

$N = N_0.N_1$**:** *We have that* $\Delta \vdash N_0 : \kappa$ *and* $\Delta \vdash N_1 : \kappa$*. By the induction hypothesis for* N_1 *and* B_0 *we obtain that* $\vdash (\!|N_1.B_0|\!) : \Pi$*. By the induction hypothesis for* N_0 *and* $N_1.B_0$ *(which is still structurally smaller than* N*) we obtain that* $\vdash (\!|N_0.(N_1.B_0)|\!) : \Pi$*. Now we see that* $(\!|N_0.(N_1.B_0)|\!) = (\!|(N_0.N_1).B_0|\!) = (\!|B|\!)$*. Hence the claim.*

$B = !B_0$**:** *We know that* $\Delta \vdash B : \Pi$*. The assumption of the induction step is clearly satisfied. Thus the claim follows from the induction hypothesis because* $\Delta^{\nu}_{B_0} = \Delta^{\nu}_{B}$ *and* $\Delta^{\mathsf{in}}_{B_0} = \Delta^{\mathsf{in}}_{B}$*.*

$B = (\nu n{:}\omega)B_0$**:** *Let* $\iota = \overline{n}$*. We know that there is some* κ' *such that* $\omega = \mathsf{Amb}[\kappa']$*. Thus* $\Delta^{\nu}_{B}(\iota) = \omega$*. Let* $\Delta_0 = \Delta[\iota \mapsto \omega]$*. We know that* $\Delta_0 \vdash B_0 : \kappa$*. Let* $\iota' \in \mathrm{dom}(\Delta_0) \cap \mathrm{ntags}(B_0)$*. When* $\iota' \neq \iota$ *then obviously* $\iota' \in \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$ *and thus* $\Delta_0(\iota') = \Delta^{\nu}_{B_0}(\iota')$*. When* $\iota' = \iota$ *we have that* $\Delta_0(\iota') = \omega = \Delta^{\nu}_{B}(\iota')$*. Now because* $\iota' \in \mathrm{ntags}(B_0)$ *we have that* $\Delta^{\nu}_{B_0}(\iota') = \omega$ *by well-scopedness condition* S4*. Thus the assumption of the induction step for* Δ_0 *and* B_0 *is satisfied.*

Let $\Pi_0 = (\!|(\Delta_0 \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!)$*. By the induction hypothesis we obtain that* $\vdash (\!|B_0|\!) : \Pi_0$*. By the same arguments used to prove the assumption of the induction step we can prove that* $\Delta_0 \cup \Delta^{\nu}_{B_0} = \Delta \cup \Delta^{\nu}_{B}$*. (When* $\iota \in \mathrm{dom}(\Delta)$ *then* $\Delta(\iota) = \Delta^{\nu}_{B}(\iota) = \omega = \Delta_0(\iota)$*.) Obviously* $\Delta^{\mathsf{in}}_{B_0} = \Delta^{\mathsf{in}}_{B}$ *and thus* $\Pi_0 = \Pi$*. Thus* $\vdash \nu n.(\!|B_0|\!) : \Pi$*. Hence the claim.*

$B = \langle N_1, \ldots, N_k \rangle$**:** *We know that* $\kappa = \omega_1 \otimes \cdots \otimes \omega_k$ *and* $\Delta \vdash N_i : \omega_i$ *for all* $0 < i \leqslant k$*. Let us prove for any* i*, by the induction on the structure of* N_i *that there is some* $\mu_i \in \mathrm{msgs}_I(\omega_i)$ *such that* $\vdash (\!|N_i|\!) : \mu_i$*. Let*

$N_i = n$**:** *Let* $\iota = \overline{n}$*. Take* $\mu_i = \iota$*. It is clear that* $\iota \in \mathrm{namesof}_I(\omega_i)$ *and thus* $\iota \in \mathrm{msgs}_I(\omega_i)$*. Hence the claim.*

$N_i = \mathsf{in}\ N$**:** *It is clear that there is some* n *such that* $N = n$*. Let* $\iota = \overline{n}$*. We can see that* $\omega = \mathsf{Amb}[\kappa']$ *for some* κ' *and* $\Delta(\iota) = \mathsf{Amb}[\kappa']$*. Take* $\mu_i = \mathsf{in}\ \iota$*. Thus* $\mu_i \in \mathrm{moves}_I$*. Hence the claim.*

$N_i = \mathsf{out}\ N$**:** *As in the case for "*$\mathsf{in}\ N$*".*

$N_i = \mathsf{open}\ N$**:** *As in the case for "*$\mathsf{in}\ N$*" but here* $\kappa' = \kappa$ *and* $\mu_i = \mathsf{open}\ \iota \in \mathrm{opens}_I(\kappa)$*.*

$N_i = N.N'$**:** *It is cleat that* $\omega = \mathsf{Cap}[\kappa']$ *for some* κ'*. From the induction hypothesis we have* μ *and* μ' *such that* $\vdash (\!|N|\!){:}\mu$ *and* $\vdash (\!|N'|\!){:}\mu'$*. When both*

μ and μ' are message types of the form $\Sigma*$ then $\mu = \mu'$ and thus $\vdash (\!|N_i|\!) : \mu$. When both μ and μ' are type tags then we have $N, N' \in \mathrm{namesof}_I(\omega)$. But we know that $\mathrm{msgs}_I(\omega)$ contains exactly one message type of the shape $\Sigma*$ and because $\mathrm{namesof}_I(\omega) = \mathrm{namesof}_I(\mathsf{Cap}\,[\kappa'\,]) \subseteq \mathrm{opens}_I(\kappa')$ we have that $\mu, \mu' \in \Sigma$. Thus $\vdash (\!|N_i|\!) : \Sigma*$. A similar situation is when only one of μ and μ' is a type tag. Then the second one is the same $\Sigma*$ as above and the first type tag is in Σ. Thus the claim.

Now let $\varphi = \texttt{<}\mu_1, \ldots, \mu_k\texttt{>}$. Let $\chi = \mathrm{nodeof}_I(\kappa)$ be the root node of Π. We see that Π contains $\chi \xrightarrow{\varphi} \chi$ because $\varphi \in \mathrm{allowedin}_I(\omega)$. Thus we can prove that $\vdash \texttt{<}(\!|N_1|\!), \ldots, (\!|N_k|\!)\texttt{>.0} : \Pi$. Hence the claim.

$B = (n_1 : \omega_1, \ldots, n_k : \omega_k).B_0$**:** *Let $\iota_i = \overline{n_i}$ for $0 < i \leqslant k$. Let $\Delta_0 = \Delta[n_1 \mapsto \omega_1, \ldots, n_k \mapsto \omega_k]$. We know that $\kappa = \omega_1 \otimes \cdots \otimes \omega_k$ and $\Delta_0 \vdash B_0 : \kappa$. By the well-scopedness condition* S1 *(the part that ν- and input-bound type tags do not intersect) we have for any i that $\iota_i \notin \mathrm{ntags}(B_0)$. Thus $\mathrm{dom}(\Delta_0) \cap \mathrm{ntags}(B_0) = \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$ and the assumption of the induction step is satisfied.*

Let $\Pi_0 = \{\!|(\Delta_0 \cup \Delta^\nu_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\}$. By the induction hypothesis we obtain that $\vdash (\!|B_0|\!) : \Pi_0$. Let $\varphi = (\iota_1, \ldots, \iota_k)$ and let $\chi = \mathrm{nodeof}_I(\kappa)$ be the root node of Π. We can see that χ is the root of Π_0 as well. Moreover we can see that Π contains all the edges of Π_0 by weakening. Thus also $\vdash (\!|B_0|\!) : \Pi$. For any i we have $\Delta^\nu_B(\iota_i) = \omega_i$ and thus $\varphi \in \mathrm{allowedin}_I(\omega_1 \otimes \cdots \otimes \omega_k)$. Thus the shape graph of Π additionally contains the edge $\chi \xrightarrow{\varphi} \chi$. Thus $\vdash (n_1, \ldots, n_k).(\!|B_0|\!) : \Pi$. Hence the claim. ∎

Proposition 17.2.2. *Let $\mathrm{ntags}(B) = \mathrm{dom}(\Delta^\nu_B)$. Let $\Delta(\iota) = \Delta^\nu_B(\iota)$ for every $\iota \in \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$. Then*

$$\vdash (\!|B|\!) : \{\!|(\Delta \cup \Delta^\nu_B, \Delta^{\mathsf{in}}_B, \kappa)|\!\} \quad \textit{implies} \quad \Delta \vdash B : \kappa.$$

Proof. *Let $I = (\Delta \cup \Delta^\nu_B, \Delta^{\mathsf{in}}_B, \kappa)$ and $\Pi = \{\!|I|\!\}$ and $\langle \Gamma, \chi \rangle = \Pi$. Thus we have $\chi = \mathrm{nodeof}_I(\kappa)$. Let $\vdash (\!|B|\!) : \Pi$. Prove $\Delta \vdash B : \kappa$ by induction on the structure of B. Let*

$B = \mathtt{0}$**:** *Clear.*

$B = (B_0 \mid B_1)$**:** *We know $\vdash (\!|B_0|\!) : \Pi$. Let $\Pi_0 = \{\!|(\Delta \cup \Delta^\nu_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\}$. Now Π contains additional edges which are not present in Π_0. These comes from type tags present in B but not in B_0. Thus we can prove $\vdash (\!|B_0|\!) : \Pi_0$ applying strengthening for each of the above type tags not present in B_0. The other two assumptions of the induction step for B_0 are clearly satisfied. By the induction hypothesis we obtain $\Delta \vdash B_0 : \kappa$. Similarly we obtain $\Delta \vdash B_1 : \kappa$. Thus the claim.*

$B = N\,[\,B_0\,]$**:** *We have* $\vdash (\!|N\,[\,B_0\,]\,|\!) : \Pi$ *and thus there is some* n *such that* $n = N$ *and* $n \neq \bullet$ *(for* $\bullet$ *is not in* Π*). Thus* $(\!|n\,[\,B_0\,]\,|\!) = n\,[\,].(\!|B_0|\!)$*. Let* $\iota = \overline{n}$*. There are some* φ *and* χ_0 *such that* $\vdash n\,[\,] : \varphi$*, and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$*, and moreover* $\vdash (\!|B_0|\!) : \langle \Gamma, \chi_0 \rangle$*. Thus* $\varphi = \iota\,[\,]$*. Let* $\kappa' = \mathrm{typeof}_I(\chi_0)$*. Take* $\Pi_0 = \{\!|(\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa')|\!\}$*. Now* Π *and* Π_0 *differ only in the root node and* Π *can contain one additional node (its root* χ*). But we can observe that all paths of* Π *which start at* χ_0 *are also present in* Π_0*. Thus* $\vdash (\!|B_0|\!) : \Pi_0$*. The assumptions of the induction step are satisfied because* $\Delta^{\nu}_{B'_0} = \Delta^{\nu}_{B}$ *and* $\Delta^{\mathsf{in}}_{B'_0} = \Delta^{\mathsf{in}}_{B}$*. By the induction hypothesis we obtain* $\Delta \vdash B'_0 : \kappa'$*.*

Let us prove $\Delta \vdash n : \mathsf{Amb}\,[\,\kappa'\,]$*. We know that* $\varphi \in \Gamma$ *and thus it holds that* $\iota \in \mathrm{namesof}_I(\mathsf{Amb}\,[\,\mathrm{typeof}_I(\chi_0)\,])$*. Because* $\iota \in \mathrm{ftags}(B)$ *it has to be the case that* $\Delta(\iota) = \mathsf{Amb}\,[\,\kappa'\,]$*. Hence the claim.*

$B = N.B_0$**:** *Take* $N'.B'_0 \equiv N.B_0$ *such that* N' *is not a composed message, that is, it is either* n*,* $\mathsf{in}\ N_0$*,* $\mathsf{out}\ N_0$*, or* $\mathsf{open}\ N_0$*. We can see that* $\vdash (\!|N'.B'_0|\!) : \Pi$ *because* $\vdash (\!|N.B_0|\!) : \Pi$*. We have that* $(\!|N'.B'_0|\!) = (\!|N'|\!)_*(\!|B'_0|\!) = (\!|N'|\!).(\!|B'_0|\!)$*. Thus there are some* φ *and* χ_0 *such that* $\vdash (\!|N'|\!) : \varphi$*, and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$*, and moreover* $\vdash (\!|B'_0|\!) : \langle \Gamma, \chi_0 \rangle$*. It is clear that* $\chi_0 = \chi$ *because* φ *is not of the shape* $\iota\,[\,]$*. Thus* $\vdash (\!|B'_0|\!) : \Pi$*. Take* $\Pi'_0 = \{\!|(\Delta \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\}$*. Obviously* $\Delta^{\nu}_{B'_0} = \Delta^{\nu}_{B}$ *and* $\Delta^{\mathsf{in}}_{B'_0} = \Delta^{\mathsf{in}}_{B}$ *and thus* $\Pi = \Pi'_0$*. The other two assumptions of the induction step for* B_0 *are clearly satisfied. By the induction hypothesis we obtain* $\Delta \vdash B'_0 : \kappa$*.*

Now let us prove $\Delta \vdash N' : \mathsf{Cap}\,[\,\kappa\,]$*. Distinguish the following cases:*

- $N' = n$**:** *Let* $\iota = \overline{n}$*. We know* $\vdash (\!|N'|\!) : \varphi$ *and thus* $\varphi = \iota$*. Also we know that* $\varphi \in \mathrm{allowedin}_I(\kappa)$*. It has to be the case that* $\varphi \in \mathrm{opens}_I(\kappa)$ *and* $\iota \in \mathrm{namesof}_I(\mathsf{Cap}\,[\,\kappa\,])$*. Because* $\iota \in \mathrm{ftags}(B)$ *it has to be the case that* $\Delta(\iota) = \mathsf{Cap}\,[\,\kappa\,]$*. Thus the claim* $\Delta \vdash \mathsf{in}\ n : \mathsf{Cap}\,[\,\kappa\,]$ *holds.*
- $N' = \mathsf{in}\ N_0$**:** *Because* $\vdash (\!|N'|\!) : \varphi$ *we can see that there has to be some* n *(*$n \neq \bullet$*) such that* $n = N_0$*. Let* $\iota = \overline{n}$*. Thus it has to be* $\varphi = \mathsf{in}\ \iota$*. Also we know that* $\varphi \in \mathrm{allowedin}_I(\kappa)$*. It has to be the case that* $\varphi \in \mathrm{moves}_I$*. Thus there is some* κ' *such that* $\iota \in \mathrm{namesof}_I(\mathsf{Amb}\,[\,\kappa'\,])$*. Because* $\iota \in \mathrm{ftags}(B)$ *it has to be the case that* $\Delta(\iota) = \mathsf{Amb}\,[\,\kappa'\,]$*. Thus* $\Delta \vdash n : \mathsf{Amb}\,[\,\kappa'\,]$ *and the claim* $\Delta \vdash \mathsf{in}\ n : \mathsf{Cap}\,[\,\kappa\,]$ *holds.*
- $N' = \mathsf{out}\ N_0$**:** *As in the case for "*$\mathsf{in}\ N$*".*
- $N' = \mathsf{open}\ N_0$**:** *As in the case for "*$\mathsf{in}\ N$*" but here* $\kappa' = \kappa$ *and* $\mathsf{open}\ \iota \in \mathrm{opens}_I(\kappa)$*.*
- **otherwise:** *Other possibilities are not allowed by the choice of* N'*.*

Thus we have $\Delta \vdash N'.B'_0 : \kappa$*. Hence the claim* $\Delta \vdash N.B_0 : \kappa$ *because* $N'.B'_0 \equiv N.B_0$*.*

$B = !B_0$**:** *Clear.*

$B = (\nu n{:}\omega)B_0$**:** *Let* $\iota = \overline{n}$. *We see* $\iota \in \mathrm{ntags}(B)$ *and thus* $\Delta^\nu_B(\iota) = \omega$. *That is why* $\omega = \mathsf{Amb}[\kappa']$ *for some* κ'. *Let* $\Delta_0 = \Delta[\iota \mapsto \omega]$. *Let* $\iota' \in \mathrm{dom}(\Delta_0) \cap \mathrm{ntags}(B_0)$. *When* $\iota' \neq \iota$ *then obviously* $\iota' \in \mathrm{dom}(\Delta) \cap \mathrm{ntags}(B)$ *and thus* $\Delta_0(\iota') = \Delta^\nu_{B_0}(\iota')$. *When* $\iota' = \iota$ *we have that* $\Delta_0(\iota') = \omega = \Delta^\nu_B(\iota')$. *Now because* $\iota' \in \mathrm{ntags}(B_0)$ *we have that* $\Delta^\nu_{B_0}(\iota') = \omega$ *by well-scopedness condition* S4.

We have that $\vdash (\!|B_0|\!) : \Pi$. *Let* $\Pi_0 = \langle\!|(\Delta_0 \cup \Delta^\nu_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\rangle$. *Now we can see that* $\Delta_0 \cup \Delta^\nu_{B_0} = \Delta \cup \Delta^\nu_B$. *(When* $\iota \in \mathrm{dom}(\Delta)$ *then* $\Delta(\iota) = \Delta^\nu_B(\iota) = \omega = \Delta_0(\iota)$.*) Thus* $\Pi_0 = \Pi$ *and* $\vdash (\!|B_0|\!) : \Pi_0$. *Moreover we see that* $\mathrm{ntags}(B_0) = \mathrm{dom}(\Delta^\nu_{B_0})$ *is satisfied as well. Thus the assumptions of the induction step for* Π_0 *and* Δ_0 *and* B_0 *is satisfied. By the induction hypothesis we obtain that* $\Delta_0 \vdash B_0 : \kappa$. *Hence the claim because we have already shown that* $\omega = \mathsf{Amb}[\kappa']$ *for some* κ'.

$B = \mathsf{<}N_1, \ldots, N_k\mathsf{>}$**:** *Let* $F = \mathsf{<}(\!|N_1|\!), \ldots, (\!|N_k|\!)\mathsf{>}$. *We see* $(\!|B|\!) = F.\mathsf{0}$. *Now because know that* $\vdash F.\mathsf{0} : \Pi$ *we have that there are* φ *and* χ_0 *such that* $\vdash F : \varphi$ *and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$. *Thus there are some* μ_1, ..., μ_k *such that* $\varphi = \mathsf{<}\mu_1, \ldots, \mu_k\mathsf{>}$ *and* $\vdash (\!|N_i|\!) : \mu_i$. *Also clearly* $\varphi \in \mathrm{allowedin}_I(\kappa)$ *and* $\varphi \in \mathrm{comms}_I(\kappa)$. *It implies that there are some* ω_1, ..., ω_k *such that* $\kappa = \omega_1 \otimes \cdots \otimes \omega_k$ *and* $\mu_i \in \mathrm{msgs}_I(\omega_i)$ *for all* i.

Let us prove $\Delta \vdash N_i : \omega_i$ *for all* i. *When* $\mu_i = \iota$ *for some* ι *(we know* $\iota \neq \bullet$*) then* $\vdash (\!|N_i|\!) : \iota$ *implies that there is some* n *such that* $n = N_i$ *and* $\iota = \overline{n}$. *Now* $\mu_i \in \mathrm{msgs}_I(\omega_i)$ *implies* $\iota \in \mathrm{namesof}_I(\omega_i)$. *We see* $\iota \in \mathrm{ftags}(B)$ *and thus it has to be the case that* $\Delta(\iota) = \omega_i$. *Hence* $\Delta \vdash N_i : \omega_i$. *When* $\mu_i = \Sigma*$ *we know that* $\omega_i = \mathsf{Cap}[\kappa']$ *for some* κ' *and also we see that* $\Sigma = \mathrm{moves}_I \cup \mathrm{opens}_I(\kappa')$. *Let us prove the claim* $\Delta \vdash N_i : \omega_i$ *by the induction of the structure of* N_i. *Let*

$N_i = n$**:** *Let* $\iota = \overline{n}$. *We have* $\vdash n : \Sigma*$ *and thus* $\iota \in \Sigma$. *Thus we see* $\iota \in \mathrm{namesof}_I(\mathsf{Cap}[\kappa'])$. *Now* $\iota \in \mathrm{ftags}(B)$ *implies that* $\Delta(\iota) = \mathsf{Cap}[\kappa']$. *Hence the claim* $\Delta \vdash n : \omega_i$.

$N_i = \mathsf{in}\ N'$**:** *Because* $\vdash (\!|N_i|\!) : \mu_i$ *and* μ_i *does not contain* $\bullet$ *we know that there is some* n *such that* $n = N'$. *Let* $\iota = \overline{n}$. *Thus* $(\!|M_i|\!) = \mathsf{in}\ n$ *and thus* $\mathsf{in}\ \iota\Sigma$. *It implies that* $\mathsf{in}\ \iota \in \mathrm{moves}_I$ *and thus there is some* κ'' *such that* $\iota \in \mathrm{namesof}_I(\mathsf{Amb}[\kappa''])$. *Because* $\iota \in \mathrm{ftags}(B)$ *we see that it must be the case* $\Delta(\iota) = \mathsf{Amb}[\kappa'']$. *Hence* $\Delta \vdash \mathsf{in}\ n : \mathsf{Cap}[\kappa']$.

$N_i = \mathsf{out}\ N'$**:** *As in the case for "*$\mathsf{in}\ N'$*".*

$N_i = \mathsf{open}\ N'$**:** *As in the case for "*$\mathsf{in}\ N'$*" but here* $\kappa' = \kappa''$ *and* $\mathsf{open}\ \iota \in \mathrm{opens}_I(\kappa')$.

$N_i = N'.N''$**:** *By the induction hypothesis we have* $\Delta \vdash N' : \mathsf{Cap}[\kappa']$ *and* $\Delta \vdash N'' : \mathsf{Cap}[\kappa']$. *Hence the claim.*

Hence the claim $\Delta \vdash B : \omega_1 \otimes \cdots \otimes \omega_k$ *holds.*

$B = (n_1 : \omega_1, \ldots, n_k : \omega_k).B_0$**:** *We know that it holds* $\vdash (\!|B|\!) : \Pi$ *and we have* $(\!|B|\!) = (n_1, \ldots, n_k).(\!|B_0|\!)$*. Thus there are some* φ *and* χ_0 *such that* $\vdash (n_1, \ldots, n_k) : \varphi$*, and* $(\chi \xrightarrow{\varphi} \chi_0) \in \Gamma$*, and* $\vdash (\!|B_0|\!) : \langle \Gamma, \chi_0 \rangle$*. We see* $\chi_0 = \chi$*. Thus* $\vdash (\!|B_0|\!) : \Pi$*. Take* $\Delta_0 = \Delta[\overline{n_1} \mapsto \omega_1, \ldots, \overline{n_k} \mapsto \omega_k]$*. Let* $\Pi_0 = \{\!|(\Delta_0 \cup \Delta^{\nu}_{B_0}, \Delta^{\mathsf{in}}_{B_0}, \kappa)|\!\}$*. We can see that* Π_0 *contains all the edges of* Π *but the edge* $(\chi \xrightarrow{\varphi} \chi_0)$ *This is because all the names* $\overline{n_i}$ *from* Δ^{in}_B *have just moved to* Δ_0*. Also by well-scopedness condition* S1 *we know that* $\overline{n_i} \notin \mathrm{dom}(\Delta)$ *for any* i*. By well-scopedness condition* S2 *we have that no* $\overline{n_i} \notin \mathrm{itags}(B_0)$ *for any* i *and thus the above edge* $(\chi \xrightarrow{\varphi} \chi_0)$ *is not used when matching* $(\!|B_0|\!)$ *against* Π*. Thus also* $\vdash (\!|B_0|\!) : \Pi_0$*. The assumptions of the induction step are satisfied. By the induction hypothesis we have that* $\Delta_0 \vdash B_0 : \kappa$*. Hence the claim.* ∎

Chapter 18

Shape Types for BioAmbients

We show how to instantiate Poly✶ to a type system for BioAmbients [RPS+04] and how to use it for flow analysis. Moreover we compare results achieved by Poly✶ with a flow analysis system for BioAmbients [NNPR07] from the literature which we call Faba.

18.1 BioAmbients (Ba)

BioAmbients is a process calculus for modeling biomolecular systems introduced by Regev, Panina, Silverman, Cardelli, and Shapiro [RPS+04]. Regev et al. present BioAmbients with the choice operator to express computation options and with replication. We work with a choice-free variant of BioAmbients with replication which we name Ba. Poly✶ can handle choice in a way that achieves the same results as Faba but we omit it to simplify the presentation.

Ba is similar to Ma but it differs in several ways. Ambients are anonymous, that is, are not labeled with names. It implies that capabilities can no longer use names to refer to ambients. Thus capabilities come in require/allow pairs synchronized by names, for example, "enter a/accept a". Then an appropriate action is performed when two ambients containing corresponding parts are found in a required position. The open capability is replaced by an operation that merges two sibling ambients. Communication is channel-based, that is, both a sender and receiver have to agree on a channel name for communication to happen. Moreover, communication is allowed also across some ambient boundaries, and only single names are exchanged.

Figure 18.1 gives the syntax of Ba. As in the case of the π-calculus and Ma, we build processes from Meta✶ names to ease the presentation. The capability "enter n" instructs an ambient to enter a sibling containing a corresponding "accept n". The capability "exit n" instructs an ambient to exit its parent ambient provided it allows it with the "expel n" capability. Finally, "merge+ n" instructs an ambient to merge with a sibling containing "merge− n". Communication is in four

Syntax of BA*:*

$$
\begin{array}{rcll}
l & \in & \text{BioLabel} & \subset \text{ TypeTag} \\
n, m & \in & \text{BioName} & ::= \text{ Name} \\
d & \in & \text{BioDirection} & ::= \mathsf{local} \mid \mathsf{p2c} \mid \mathsf{c2p} \mid \mathsf{s2s} \\
N & \in & \text{BioCapability} & ::= \mathsf{enter}\ n \mid \mathsf{exit}\ n \mid \mathsf{merge{+}}\ n \mid \\
 & & & \quad \mathsf{accept}\ n \mid \mathsf{expel}\ n \mid \mathsf{merge{-}}\ n \\
B & \in & \text{BioProcess} & ::= \mathsf{0} \mid B_0 \mid B_1 \mid [B]^l \mid N.B \mid !B \mid (\nu n)B \mid \\
 & & & \quad d\ n?\{m\}.B \mid d\ n!\{m\}.B
\end{array}
$$

Structural equivalence of BA*:*

$$\frac{}{B \equiv B} \qquad \frac{B_0 \equiv B_1}{B_1 \equiv B_0} \qquad \frac{B_0 \equiv B_1 \quad B_1 \equiv B_2}{B_0 \equiv B_2} \qquad \frac{B_0 \equiv B_1}{B_0 \mid B_2 \equiv B_1 \mid B_2}$$

$$\frac{B_0 \equiv B_1}{[B_0]^l \equiv [B_1]^l} \qquad \frac{B_0 \equiv B_1}{N.B_0 \equiv N.B_1} \qquad \frac{B_0 \equiv B_1}{!B_0 \equiv !B_1} \qquad \frac{B_0 \equiv B_1}{(\nu n)B_0 \equiv (\nu n)B_1}$$

$$\frac{B_0 \equiv B_1}{d\ n?\{m\}.B_0 \equiv d\ n?\{m\}.B_1} \qquad \frac{B_0 \equiv B_1}{d\ n!\{m\}.B_0 \equiv d\ n!\{m\}.B_1} \qquad \frac{}{B \mid \mathsf{0} \equiv B}$$

$$\frac{}{B_0 \mid B_1 \equiv B_1 \mid B_0} \qquad \frac{}{B_0 \mid (B_1 \mid B_2) \equiv (B_0 \mid B_1) \mid B_2} \qquad \frac{}{!\mathsf{0} \equiv \mathsf{0}} \qquad \frac{}{!B \equiv B \mid !B}$$

$$\frac{}{(\nu n)\mathsf{0} \equiv \mathsf{0}} \qquad \frac{}{(\nu n)([B]^l) \equiv [(\nu n)B]^l} \qquad \frac{n \notin \mathrm{fn}(B_0)}{B_0 \mid (\nu n)B_1 \equiv (\nu n)(B_0 \mid B_1)}$$

$$\frac{}{(\nu n)(\nu m)B \equiv (\nu m)(\nu n)B}$$

Figure 18.1: Syntax and structural equivalence of BA.

directions: between processes in the same ambient (local), between processes in sibling ambients (s2s), from a parent ambient to its child (p2c), and from a child to the parent (c2p). Only a single name (not capabilities) can be sent. The output action syntax is "$d\ n\ !\{m\}$" where n is the channel name, d is the desired direction, and m is the name being sent. The input prefix "$d\ n\ ?\{m\}$" is similar and (input-)binds the name m.

Flow analysis must refer to ambients to track changes, so following the approach of FABA, our syntax introduces ambient labels with no influence on the semantics. We use META✶ type tags as labels and write $[B]^l$ for an ambient labeled l. Bound type tags and free names of a process are defined like in META✶. The set tags(B) do not contain ambient labels. Processes are identified up to α-conversion of bound names which preserves type tags. We set SpecialTag = $\{\bullet, \mathsf{enter}, \mathsf{exit}, \mathsf{merge{+}}, \mathsf{accept}, \mathsf{expel}, \mathsf{merge{-}}, \mathsf{local}, \mathsf{p2c}, \mathsf{c2p}, \mathsf{local}\}$ in order to prevent type tags with a special meaning to be bound. We require all processes to be well formed according to the following definition. Well-formedness can be achieved by

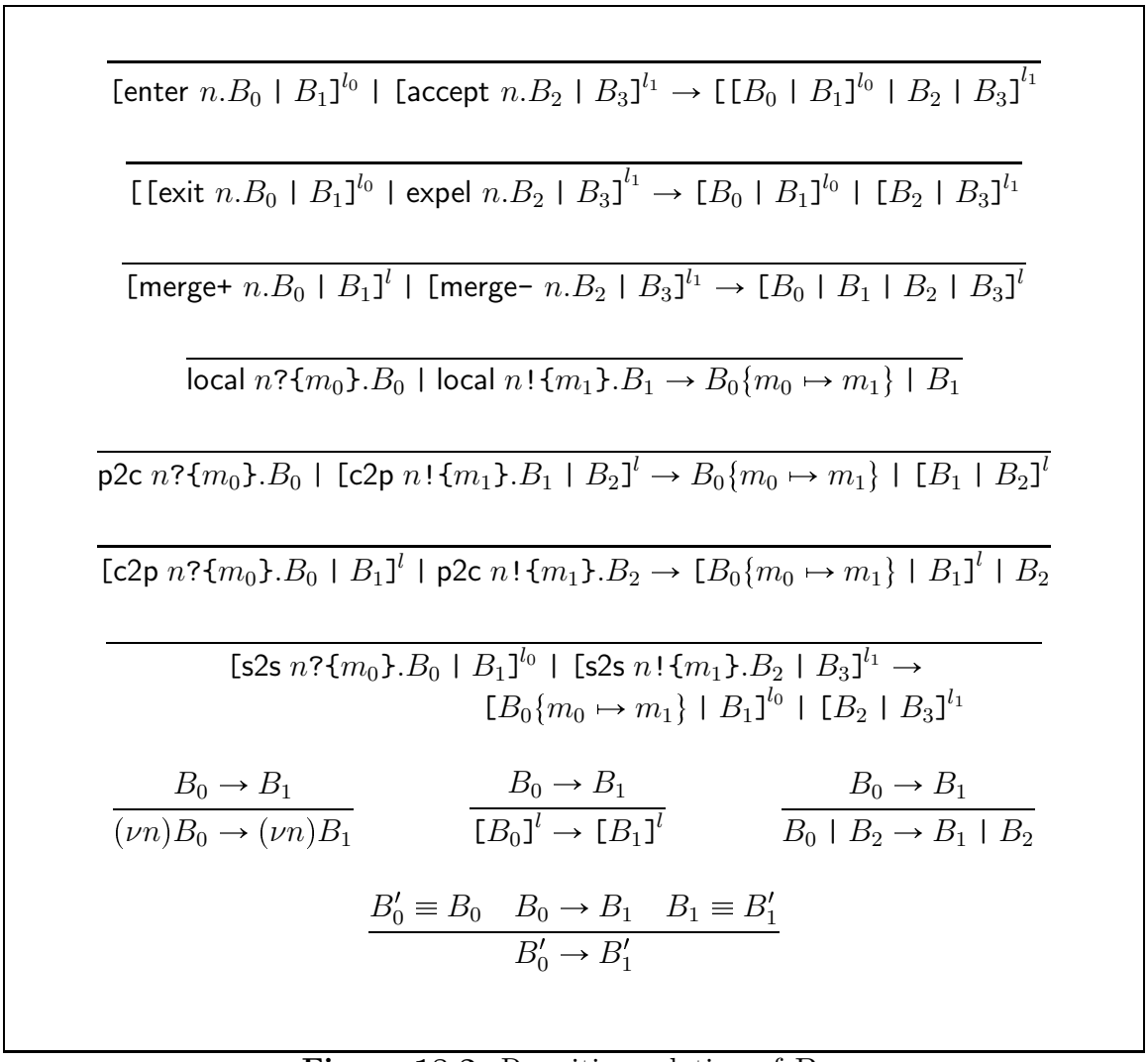


Figure 18.2: Rewriting relation of BA.

name renaming if necessary and it is preserved by rewriting.

DEFINITION 18.1.1. *A process B is* **well formed** *iff all the following hold.*

(S1) $\mathrm{ftags}(B) \cup \mathrm{itags}(B)$ *is disjoint with* $\mathrm{ntags}(B)$

(S2) *for "d* $n?\{m\}.B_0$*" in* B, $\overline{m} \notin \mathrm{itags}(B_0)$

(S3) *type tags of names from* B *are distinct from ambient labels from* B

(S4) B *do not contain any type tags from* SpecialTag ∎

Figure 18.1 also presents BA structural equivalence. The semantics of BA is in Figure 18.2.

EXAMPLE 18.1.2. *Consider the following simple* BA *process.*

$$B = \begin{array}{l}[\mathsf{enter\ n.accept\ x.0} \mid \mathsf{enter\ m.merge\text{-}\ y.0}]^{\mathsf{a}} \mid \\ [\mathsf{accept\ n.0}]^{\mathsf{b}} \mid [\mathsf{accept\ m.0}]^{\mathsf{c}}\end{array}$$

The following two different rewritings can be proved.

$$B \to [[\mathsf{accept\ x.0} \mid \mathsf{enter\ m.merge\text{-}\ y.0}]^{\mathsf{a}}]^{\mathsf{b}} \mid [\mathsf{accept\ m.0}]^{\mathsf{c}}$$
$$B \to [\mathsf{accept\ n.0}]^{\mathsf{b}} \mid [[\mathsf{enter\ n.accept\ x.0} \mid \mathsf{merge\text{-}\ y.0}]^{\mathsf{a}}]^{\mathsf{c}}$$

18.2 Flow Analysis of BioAmbients (FABA)

Nielson, Nielson, Priami, and Rosa [NNPR07] designed a flow analysis system for BioAmbients (hereafter FABA) which conservatively over-approximates the states that a system can evolve to. The original FABA works for a version of BA with the μ (also called **rec**) operator instead of replication. Here we suppose only a restricted usage of μ which can be expressed by replication because META✶ does not support μ at the current moment. We could emulate the μ operator using additional rules as described in Section 9.3.1 but we prefer to work with replication in order to simplify the presentation.

The original FABA does α-conversion similarly to META✶. It assigns a canonical name to every name that is preserved by α-conversion. We identify these canonical names with META✶ type tags. Canonical names are used in canonical capabilities and communication prefixes, which we map into POLY✶ form types.

FABA takes a BA process as an input and its output collects information about possible contents of ambients in any process that the input process can evolve to. A result of FABA analysis is a pair $(\mathcal{S}, \mathcal{N})$ where $\mathcal{S} \subseteq \text{BioLabel} \times \text{FormType}$, and $\mathcal{N} \subseteq \text{TypeTag} \times \text{TypeTag}$. For every ambient, $\mathcal{S}$ collects information about possible child ambients, capabilities, and communication prefixes contained in it. For example $(\mathsf{a}, \mathsf{b}[\,]) \in \mathcal{S}$ says[1] that the ambient (with the label) a can have a child ambient b, while $(\mathsf{a}, \mathsf{enter\ n}) \in \mathcal{S}$ says that an ambient with the label a can possibly contain (and execute) the capability "$\mathsf{enter}\ a^{\mathsf{n}}$" for any a. Note that members of $\mathcal{S}$ are built from type tags. In order to match the syntax of action types we write "$d\ a(b)$" instead of "$d\ a?\{b\}$", and "$d\ a\langle b\rangle$" instead of "$d\ a!\{b\}$".

Input-bound names are handled in a special way. Capabilities built from input-bound names are not contained in $\mathcal{S}$. Instead, $\mathcal{S}$ contains all their actual instantiations introduced by communication. For example, for the input process "$\mathsf{local\ a?\{x\}.enter\ x.0} \mid \mathsf{local\ a!\{b\}.0}$", the $\mathcal{S}$ part of the result contains "$\mathsf{enter\ b}$" but not "$\mathsf{enter\ x}$". The set $\mathcal{N}$ describes possible name instantiations invoked by communication. For example $(\mathsf{x}, \mathsf{b}) \in \mathcal{N}$ says that communication can instantiate x to b.

FABA defines the predicate $(\mathcal{S}, \mathcal{N}) \models^{l} B$ meaning that B matches the structure allowed by $(\mathcal{S}, \mathcal{N})$ inside the ambient l. The name "$\star$" is used to refer to the top

[1] In the original paper [NNPR07] the set $\mathcal{S}$ contains (ι, ι_0) instead of $(\iota, \iota_0[\,])$. This technical change we make allows easier formulation of our expressiveness evaluation.

$$
\begin{array}{lll}
(\mathcal{S},\mathcal{N}) \models^l \mathsf{0} & \text{iff} & \text{true} \\
(\mathcal{S},\mathcal{N}) \models^l B_0 \mid B_1 & \text{iff} & (\mathcal{S},\mathcal{N}) \models^l B_0 \;\&\; (\mathcal{S},\mathcal{N}) \models^l B_1 \\
(\mathcal{S},\mathcal{N}) \models^l [B]^{l_0} & \text{iff} & \mathcal{S}(l, l_0\,[\,]) \;\&\; (\mathcal{S},\mathcal{N}) \models^{l_0} B \\
(\mathcal{S},\mathcal{N}) \models^l N.B & \text{iff} & (\mathcal{S},\mathcal{N}) \models^l N \;\&\; (\mathcal{S},\mathcal{N}) \models^l B \\
(\mathcal{S},\mathcal{N}) \models^l d\ n?\{m\}.B & \text{iff} & (\mathcal{S},\mathcal{N}) \models^l d\ n?\{m\} \;\&\; (\mathcal{S},\mathcal{N}) \models^l B \\
(\mathcal{S},\mathcal{N}) \models^l d\ n!\{m\}.B & \text{iff} & (\mathcal{S},\mathcal{N}) \models^l d\ n!\{m\} \;\&\; (\mathcal{S},\mathcal{N}) \models^l B \\
(\mathcal{S},\mathcal{N}) \models^l !B & \text{iff} & (\mathcal{S},\mathcal{N}) \models^l B \\
(\mathcal{S},\mathcal{N}) \models^l (\nu a^i)B & \text{iff} & \mathcal{N}(a,a) \;\&\; (\mathcal{S},\mathcal{N}) \models^l B
\end{array}
$$

$$
\begin{array}{lll}
(\mathcal{S},\mathcal{N}) \models^l \mathsf{enter}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{enter}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l \mathsf{accept}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{accept}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l \mathsf{exit}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{exit}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l \mathsf{expel}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{expel}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l \mathsf{merge+}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{merge+}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l \mathsf{merge-}\ n & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, \mathsf{merge-}\ \iota) \\
(\mathcal{S},\mathcal{N}) \models^l d\ n?\{m\} & \text{iff} & \forall \iota : \mathcal{N}(\overline{n},\iota) \Rightarrow \mathcal{S}(l, d\ \iota(\overline{m})) \\
(\mathcal{S},\mathcal{N}) \models^l d\ n!\{m\} & \text{iff} & \forall \iota_0,\iota_1 : \mathcal{N}(\overline{n},\iota_0) \;\&\; \mathcal{N}(\overline{m},\iota_1) \Rightarrow \mathcal{S}(l, d\ \iota_0\langle\iota_1\rangle)
\end{array}
$$

Figure 18.3: FABA analysis of BA processes.

level location. $(\mathcal{S},\mathcal{N}) \models^l B$Figure 18.3 defines the relation $(\mathcal{S},\mathcal{N}) \models^l B$. When an input process B is given, this figure gives us the set of conditions on $(\mathcal{S},\mathcal{N})$ that has to be satisfied for $(\mathcal{S},\mathcal{N}) \models^\star B$ to hold. For example, $(\mathcal{S},\mathcal{N}) \models^\star \mathsf{enter}\ a^\iota.B$ holds iff $(\mathcal{S},\mathcal{N}) \models^\star B$ and $\mathcal{S}(\mathsf{enter}\ \iota', \star)$ for all ι' to which ι can be renamed to (that is, such that $\mathcal{N}(\iota,\iota')$).

Figure 18.4 specifies conditions which a FABA result has to satisfy to be closed under rewriting. These conditions directly correspond to the BA rewriting rules. For example for the local communication rule, $\mathcal{S}(\star, \mathsf{local}\ \iota(\iota_0))$ and $\mathcal{S}(\star, \mathsf{local}\ \iota\langle\iota_1\rangle)$ has to imply $\mathcal{N}(\iota_0,\iota_1)$ because the rewriting can result in a corresponding renaming. The FABA result for B is the smallest pair $(\mathcal{S},\mathcal{N})$ such $(\mathcal{S},\mathcal{N}) \models^l B$ and which satisfies all the closure conditions from Figure 18.4. FABA ensures that the structure described by a valid result is closed under rewritings.

EXAMPLE 18.2.1. *The* FABA *resutl for the process B from Example 18.1.2 is as follows.*

$$
\begin{array}{ll}
\mathcal{N} = \{ & (\mathsf{n},\mathsf{n}), (\mathsf{m},\mathsf{m}), (\mathsf{x},\mathsf{x}), (\mathsf{y},\mathsf{y}) \;\} \\
\mathcal{S} = \{ & (\star, \mathsf{a}\,[\,]), (\mathsf{a}, \mathsf{enter}\ \mathsf{n}), (\mathsf{a}, \mathsf{enter}\ \mathsf{m}), (\mathsf{a}, \mathsf{accept}\ \mathsf{x}), (\mathsf{a}, \mathsf{merge-}\ \mathsf{y}), \\
& (\star, \mathsf{b}\,[\,]), (\mathsf{b}, \mathsf{a}\,[\,]), (\mathsf{b}, \mathsf{accept}\ \mathsf{n}), (\star, \mathsf{c}\,[\,]), (\mathsf{c}, \mathsf{a}\,[\,]), (\mathsf{c}, \mathsf{accept}\ \mathsf{m}) \;\}
\end{array}
$$

18.3 Instantiation of META⋆ to BioAmbients

We can express BA prefixes "$d\ n\ !\{m\}$" and "$d\ n\ ?\{m\}$" as 3-length META⋆ forms "$d\ n\langle m\rangle$" and "$d\ n(m)$" respectively. Ambient labels can be translated using an

$$
\begin{array}{ll}
\forall l, l_1, l_2, \iota: & \mathcal{S}(l_1, \mathsf{enter}\ \iota)\ \&\ \mathcal{S}(l, l_1\,[\,])\ \&\ \mathcal{S}(l_2, \mathsf{accept}\ \iota)\ \&\ \mathcal{S}(l, l_2\,[\,]) \\
& \qquad \Rightarrow \mathcal{S}(l_2, l_1) \\
\forall l, l_1, l_2, \iota: & \mathcal{S}(l_2, \mathsf{exit}\ \iota)\ \&\ \mathcal{S}(l_1, l_2\,[\,])\ \&\ \mathcal{S}(\mathsf{expel}\ \iota, l_1)\ \&\ \mathcal{S}(l, l_1\,[\,]) \\
& \qquad \Rightarrow \mathcal{S}(l, l_2) \\
\forall l, l_1, l_2, \iota: & \mathcal{S}(l_1, \mathsf{merge{+}}\ \iota)\ \&\ \mathcal{S}(l, l_1\,[\,])\ \&\ \mathcal{S}(l_2, \mathsf{merge{-}}\ \iota)\ \&\ \mathcal{S}(l, l_2\,[\,]) \\
& \qquad \Rightarrow (\forall \varphi : \mathcal{S}(l_2, \varphi) \Rightarrow \mathcal{S}(l_1, \varphi)) \\
\forall l, \iota, \iota_0, \iota_1: & \mathcal{S}(\mathsf{local}\ \iota(\iota_0), l)\ \&\ \mathcal{S}(\mathsf{local}\ \iota{<}\iota_1{>}, l) \\
& \qquad \Rightarrow \mathcal{N}(\iota_0, \iota_1) \\
\forall l, l_0, \iota, \iota_0, \iota_1: & \mathcal{S}(l, \mathsf{p2c}\ \iota{<}\iota_1{>})\ \&\ \mathcal{S}(l, l_0\,[\,])\ \&\ \mathcal{S}(l_0, \mathsf{c2p}\ \iota(\iota_0)) \\
& \qquad \Rightarrow \mathcal{N}(\iota_0, \iota_1) \\
\forall l, l_0, \iota, \iota_0, \iota_1: & \mathcal{S}(l, \mathsf{p2c}\ \iota(\iota_0))\ \&\ \mathcal{S}(l, l_0\,[\,])\ \&\ \mathcal{S}(l_0, \mathsf{c2p}\ \iota{<}\iota_1{>}) \\
& \qquad \Rightarrow \mathcal{N}(\iota_0, \iota_1) \\
\forall l, l_0, l_1, \iota, \iota_0, \iota_1: & \mathcal{S}(l_0, \mathsf{s2s}\ \iota(\iota_0))\ \&\ \mathcal{S}(l, l_0\,[\,])\ \&\ \mathcal{S}(l_1, \mathsf{s2s}\ \iota{<}\iota_1{>})\ \&\ \mathcal{S}(l, l_1\,[\,]) \\
& \qquad \Rightarrow \mathcal{N}(\iota_0, \iota_1)
\end{array}
$$

Figure 18.4: Closure conditions valid for FABA results.

ambient syntactic sugar as in MA, that is "$\mathtt{[0]}^l$" as "$l^l\,\mathtt{[0]}$". Then the syntax of BA matches the syntax of META✶.

Recall that

$$\text{SpecialTag} = \{\bullet, \mathsf{enter}, \mathsf{exit}, \mathsf{merge{+}}, \mathsf{accept}, \mathsf{expel}, \mathsf{merge{-}}, \mathsf{local}, \mathsf{p2c}, \mathsf{c2p}, \mathsf{local}\}.$$

The set $\mathcal{B}$ of META✶ rewriting rules looks as follows.

$$
\begin{array}{ll}
\mathcal{B} = \{ & \mathbf{active}\{\,\mathring{\mathtt{P}}\ \mathbf{in}\ \mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{P}}\mathtt{]}\,\}, \\
& \mathbf{rewrite}\{\,\mathring{\mathtt{a}}\mathtt{[enter}\ \mathring{\mathtt{n}}.\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{b}}\mathtt{[accept}\ \mathring{\mathtt{n}}.\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]} \hookrightarrow \mathring{\mathtt{b}}\mathtt{[}\mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]}\,\}, \\
& \mathbf{rewrite}\{\,\mathring{\mathtt{b}}\mathtt{[}\mathring{\mathtt{a}}\mathtt{[exit}\ \mathring{\mathtt{n}}.\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathsf{expel}\ \mathring{\mathtt{n}}.\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]} \hookrightarrow \mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{b}}\mathtt{[}\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]}\,\}, \\
& \mathbf{rewrite}\{\,\mathring{\mathtt{a}}\mathtt{[}\mathsf{merge{+}}\ \mathring{\mathtt{n}}.\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{b}}\mathtt{[}\mathsf{merge{-}}\ \mathring{\mathtt{n}}.\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]} \hookrightarrow \mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\ \mathtt{|}\ \mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]}\,\}, \\
& \mathbf{rewrite}\{\,\mathsf{local}\ \mathring{\mathtt{n}}(\mathring{\mathtt{x}}).\mathring{\mathtt{P}}\ \mathtt{|}\ \mathsf{local}\ \mathring{\mathtt{n}}{<}\mathring{\mathtt{M}}{>}.\mathring{\mathtt{Q}} \hookrightarrow \{\mathring{\mathtt{x}} := \mathring{\mathtt{M}}\}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\,\}, \\
& \mathbf{rewrite}\{\,\mathsf{p2c}\ \mathring{\mathtt{n}}(\mathring{\mathtt{x}}).\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{a}}\mathtt{[}\mathsf{c2p}\ \mathring{\mathtt{n}}{<}\mathring{\mathtt{M}}{>}.\mathring{\mathtt{Q}}\ \mathtt{|}\ \mathring{\mathtt{R}}\mathtt{]} \hookrightarrow \{\mathring{\mathtt{x}} := \mathring{\mathtt{M}}\}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{a}}\mathtt{[}\mathring{\mathtt{Q}}\ \mathtt{|}\ \mathring{\mathtt{R}}\mathtt{]}\,\}, \\
& \mathbf{rewrite}\{\,\mathring{\mathtt{a}}\mathtt{[}\mathsf{c2p}\ \mathring{\mathtt{n}}(\mathring{\mathtt{x}}).\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathsf{p2c}\ \mathring{\mathtt{n}}{<}\mathring{\mathtt{M}}{>}.\mathring{\mathtt{R}} \hookrightarrow \mathring{\mathtt{a}}\mathtt{[}\{\mathring{\mathtt{x}} := \mathring{\mathtt{M}}\}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{R}}\,\}, \\
& \mathbf{rewrite}\{\,\mathring{\mathtt{a}}\mathtt{[}\mathsf{s2s}\ \mathring{\mathtt{n}}(\mathring{\mathtt{x}}).\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{b}}\mathtt{[}\mathsf{s2s}\ \mathring{\mathtt{n}}{<}\mathring{\mathtt{M}}{>}.\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]} \hookrightarrow \mathring{\mathtt{a}}\mathtt{[}\{\mathring{\mathtt{x}} := \mathring{\mathtt{M}}\}\mathring{\mathtt{P}}\ \mathtt{|}\ \mathring{\mathtt{Q}}\mathtt{]}\ \mathtt{|}\ \mathring{\mathtt{b}}\mathtt{[}\mathring{\mathtt{R}}\ \mathtt{|}\ \mathring{\mathtt{S}}\mathtt{]}\,\}\ \}
\end{array}
$$

The following set $\mathcal{B}$ instantiates META✶ to BA and POLY✶ to BA's type system $S_{\mathcal{B}}$.

EXAMPLE 18.3.1. POLY✶ *principal type* Π_B *for a* META✶ *equivalent of* B *from*

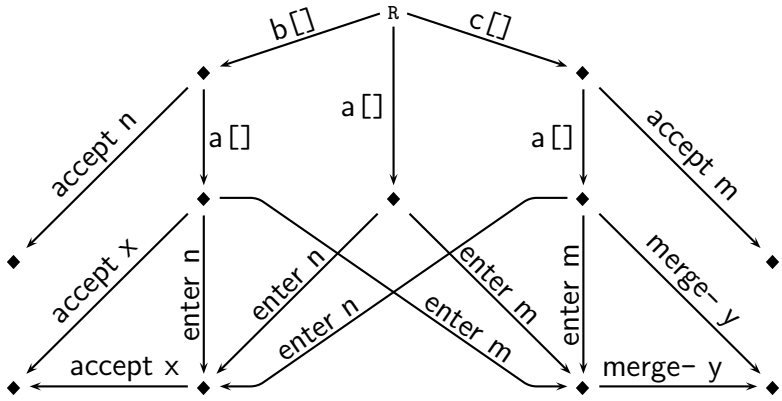
$$
\begin{array}{ll}
(\!|\mathsf{local}|\!) = \mathsf{local} & (\!|\mathsf{p2c}|\!) = \mathsf{p2c} \\
(\!|\mathsf{c2p}|\!) = \mathsf{c2p} & (\!|\mathsf{s2s}|\!) = \mathsf{s2s}
\end{array}
$$

$$
\begin{array}{l}
(\!|\mathsf{enter}\ n|\!) = \mathsf{enter}\ n \\
(\!|\mathsf{accept}\ n|\!) = \mathsf{accept}\ n \\
(\!|\mathsf{exit}\ n|\!) = \mathsf{exit}\ n \\
(\!|\mathsf{expel}\ n|\!) = \mathsf{expel}\ n \\
(\!|\mathsf{merge{+}}\ n|\!) = \mathsf{merge{+}}\ n \\
(\!|\mathsf{merge{-}}\ n|\!) = \mathsf{merge{-}}\ n
\end{array}
$$

$$
\begin{array}{l}
(\!|0|\!) = 0 \\
(\!|B_0 \mid B_1|\!) = (\!|B_0|\!) \mid (\!|B_1|\!) \\
(\!|\,[B]^{l}\,|\!) = l^{l}\,[(\!|B|\!)] \\
(\!|N.B|\!) = (\!|N|\!).(\!|B|\!) \\
(\!|d\ n?\{m\}|\!).B = (\!|d|\!)\ n(m).(\!|B|\!) \\
(\!|d\ n!\{m\}|\!).B = (\!|d|\!)\ n\langle m\rangle.(\!|B|\!) \\
(\!|!B|\!) = !(\!|B|\!) \\
(\!|(\nu n)B|\!) = \nu n.(\!|B|\!)
\end{array}
$$

Figure 18.5: Encoding of BA processes in META✶.

Example18.1.2 looks as follows.



Contents of ambients can be easily read from it. It also shows POLY✶*'s spatial polymorphism in action: ambient* a *can execute "*accept x*" only when contained inside ambient* b*, and similarly for "*merge- y*" and* c. □

18.4 POLY✶ Types and FABA Results

Using the notation from Section 13.1 we have that C is BA, S_C is FABA, predicates ρ are triples $(\mathcal{S}, \mathcal{N}, l)$, and S_C's relation $\rhd B : \rho$ is $(\mathcal{S}, \mathcal{N}) \models^{l} B$. Moreover, $\mathcal{B}$, $C_{\mathcal{B}}$, and $S_{\mathcal{B}}$ were discussed in Section 18.3. This section shows that POLY✶ can provide the same information as FABA and can do better. The encoding $(\!|\cdot|\!)$ of BA processes in META✶ is presented in Figure 18.5. Property 13.3.1 holds.

18.4.1 FABA Result from Shape Type

Information provided by FABA results are contained in POLY✶ principal types as well. For example when the shape graph contains "$\chi_0 \xrightarrow{l_0\,[\,]} \chi_1 \xrightarrow{l_1\,[\,]} \chi_2$" then it means that ambient l_0 can possibly contain ambient l_1. The above two edges can be possibly separated by other edges. We use the following two predicates to extract relevant information from a shape predicate $\Pi = \langle \Gamma, \chi_{\mathsf{r}} \rangle$. A form type φ is said to be

under the root of Π, written $\mathrm{inroot}_\Pi(\varphi)$, when Γ contains the path $\{\chi_\mathsf{r} \xrightarrow{\varphi_1} \chi_0 \cdots \xrightarrow{\varphi_k} \chi_{k-1} \xrightarrow{\varphi} \chi_k\}$ of edges starting at the root χ_r where no φ_i has the shape "$l\,[\,]$". The condition on the shape of φ_i's expresses that φ is not inside any ambient. Similarly, the predicate $\mathrm{inamb}_\Pi(l, \varphi)$ holds when φ is contained directly inside the ambient l in Γ. That is, when Γ contains the path $\{\chi \xrightarrow{l\,[\,]} \chi_0 \xrightarrow{\varphi_1} \chi_1 \cdots \xrightarrow{\varphi_k} \chi_k \xrightarrow{\varphi} \chi_{k+1}\}$ starting this time at any node and where no φ_i can have the shape "$l_1\,[\,]$". We write $\mathrm{inamb}_\Pi(\star, \varphi)$ for $\mathrm{inroot}_\Pi(\varphi)$.

The following predicate is used to recognize non-instantiated capabilities, that is, those that contain type tags which are bound in some other action types of the shape graph. Let $\mathrm{itags}(\Gamma)$ be the set of all type tags which appear as one of ι_i's in some $(\iota_1, \ldots, \iota_k)$ in Γ. Write $\mathrm{instant}_\Pi(\varphi)$ when φ labels some edge in the shape graph Γ of Π and $\mathrm{ftags}(\varphi) \cap \mathrm{itags}(\Gamma) = \emptyset$. Then a Faba-like result is constructed from a shape predicate $\Pi = \langle \Gamma, \chi \rangle$ as follows.

$$\mathcal{S}_\Pi = \{(l, \varphi) : \mathrm{inamb}_\Pi(l, \varphi)\ \&\ \mathrm{instant}_\Pi(\varphi)\}$$
$$\mathcal{N}_\Pi = \{(\iota, \iota') : (\chi_0 \overset{\{\iota \mapsto \iota'\}}{\cdots\cdots\rightarrow} \chi_1) \in \Gamma\} \cup \{(\iota, \iota) : \exists \varphi.\iota \in (\mathrm{ftags}(\varphi) \backslash \mathrm{BioLabel})\ \&\ \mathrm{instant}_\Pi(\varphi)\}$$

The set $\mathcal{N}_\Pi$ is constructed from Poly★ flow-edges. Theorem 18.4.1 describes the relation between native Faba results and those constructed from Poly★: Poly★ principal types contain the information provided by Faba. When (l, φ) is in $\mathcal{S}$ but not in $\mathcal{S}_\Pi$, then subject reduction of Poly★ ensures the situation predicted by Faba can never happen, in which case Poly★ is more precise.

Theorem 18.4.1. *Let $(\mathcal{S}, \mathcal{N})$ be the result of* Faba *analysis for B Let Π be a* Poly★ *restricted principal $\mathcal{B}$-type of $(\!|B|\!)$. The following holds.*

$$\mathcal{S}_\Pi \subseteq \mathcal{S} \quad \& \quad \mathcal{N}_\Pi \subseteq \mathcal{N} \quad \& \quad (\mathcal{S}_\Pi, \mathcal{N}_\Pi) \models^\star B$$

Example 18.4.2. *The sets $\mathcal{S}_\Pi$ and $\mathcal{N}_\Pi$ constructed for process B (Example 18.1.2) from the shape type Π_B (Example 18.3.1) gives exactly the same result as* Faba *(Example 18.2.1) because of the simplicity of our example. However, Example 18.3.1 shows how* Poly★ *can express more detailed information not contained in* Faba *results.* □

18.4.2 Shape Type from Faba Result

This section shows how to construct a Poly★ shape type which exactly correspond to a given Faba result. To be able to do this we need an upper bound on input-bound names allowed in the examined process. The reasons for this limitation were discussed in Section 13.4.

We do not need to be able to construct a shape predicate for every possible Faba predicate but only for those which are valid Faba results. We could require this

Sets of labels and nodes; bejections between them:

$\text{labels}_{\mathcal{S}} = \text{dom}(\mathcal{S}) \cup (\text{rng}(\mathcal{S}) \cap \text{BioLabel}) \cup \{\star\}$
$\text{nodes}_{\mathcal{S}} =$ arbitrary but fixed nodes set of the same size as $\text{labels}_{\mathcal{S}}$
$\text{nodeof}_{\mathcal{S}} = \text{labelof}_{\mathcal{S}}^{-1}$ ... bijections from $\text{labels}_{\mathcal{S}}$ into $\text{nodes}_{\mathcal{S}}$ and reversely

Sets of form types describing legal actions:

$$\text{activecaps}_{(\mathcal{S},\mathcal{N})}(l) = \{d\ \iota\!<\!\iota_0\!>\colon \mathcal{S}(l, d\ \iota'\!<\!\iota_0'\!>)\ \&\ \mathcal{N}(\iota,\iota')\ \&\ \mathcal{N}(\iota_0,\iota_0')\} \cup \{\overline{\{\iota' \mapsto \iota\}}\varphi\colon \mathcal{S}(l,\varphi)\ \&\ \iota' \in \text{ftags}(\varphi)\ \&\ \mathcal{N}(\iota,\iota')\ \&\ \varphi \notin \text{labels}_{\mathcal{S}}\}$$

$$\text{inertcaps}_{(\mathcal{N},Z)} = \{d\ \iota(\iota_0) : d \in \text{BioDirection}\ \&\ \iota \in Z\backslash\text{dom}(\mathcal{N})\ \&\ \iota_0 \in Z\} \cup \{d\ \iota\!<\!\iota_0\!>, d\ \iota\!<\!\iota_0\!>\colon d \in \text{BioDirection}\ \&\ \iota \in Z\backslash\text{dom}(\mathcal{N})\ \&\ \iota_0 \in Z \cup \text{dom}(\mathcal{N})\} \cup \{\mathsf{enter}\ \iota, \mathsf{accept}\ \iota, \mathsf{exit}\ \iota, \mathsf{expel}\ \iota, \mathsf{merge{+}}\ \iota, \mathsf{merge{-}}\ \iota\colon \iota \in Z\backslash\text{dom}(\mathcal{N})\}$$

$$\text{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l) = \text{activecaps}_{(\mathcal{S},\mathcal{N})}(l) \cup \text{inertcaps}_{(\mathcal{N},Z)}$$

Construction of a shape graph:

$$\Gamma_{(\mathcal{S},\mathcal{N},Z)} = \{\text{nodeof}_{\mathcal{S}}(l) \xrightarrow{l_0\,[\,]} \text{nodeof}_{\mathcal{S}}(l_0)\colon (l, l_0\,[\,]) \in \mathcal{S}\ \&\ l, l_0 \in \text{labels}_{\mathcal{S}}\} \cup \{\chi \xrightarrow{\varphi} \chi\colon \chi \in \text{nodes}_{\mathcal{S}}\ \&\ \varphi \in \text{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(\text{labelof}_{\mathcal{S}}(\chi))\}$$

Figure 18.6: Construction of a shape graph corresponding to a FABA result.

directly but it useful to explicitly state a specific condition on a FABA result. This condition is required for our construction to be correct and it is satisfied for all valid FABA results. The condition on $(\mathcal{S},\mathcal{N})$ is as follows.

DEFINITION 18.4.3. *We say that* $(\mathcal{S},\mathcal{N})$ *is* **closed** *when all of the following hold for an arbitrary* $l, \iota, \iota', \iota_0, \iota_0', d$.

(1) $\mathcal{N}(\iota,\iota')\ \&\ \mathcal{S}(l, \mathsf{enter}\ \iota') \Rightarrow (\forall \iota''\colon \mathcal{N}(\iota,\iota'') \Rightarrow \mathcal{S}(l, \mathsf{enter}\ \iota''))$

(2)-(6) as case (1) but for accept, ..., merge-

(7) $\mathcal{N}(\iota,\iota')\ \&\ \mathcal{S}(l, d\ \iota'(\iota_0)) \Rightarrow (\forall \iota''\colon \mathcal{N}(\iota,\iota'') \to \mathcal{S}(l, d\ \iota''(\iota_0)))$

(8) $\mathcal{N}(\iota,\iota')\ \&\ \mathcal{N}(\iota_0,\iota_0')\ \&\ \mathcal{S}(l, d\ \iota'\!<\!\iota_0'\!>) \Rightarrow (\forall \iota'', \iota_0''\colon \mathcal{N}(\iota,\iota'')\ \&\ \mathcal{N}(\iota_0,\iota_0'') \to \mathcal{S}(l, d\ \iota''\!<\!\iota_0''\!>))$ ∎

The condition above has eight parts, one for each possible action prefix. It reflects how input-bound names are handled in FABA. Let us describe the case for enter. It says that when an ambient labeled by l can contain "enter ι'" and some ι can be instantiated to ι' by communication then l can also contain all other instantiations of "enter ι". Other cases are similar. Note that when $(\mathcal{S},\mathcal{N})$ is a valid FABA result for B than the following two claims hold. (1) When $\iota \in \text{ftags}(B) \cup \text{ntags}(B)$ then $\mathcal{N}(\iota,\iota)$, and for any ι' such that $\mathcal{N}(\iota,\iota')$ it holds $\iota' = \iota$. (2) When $\iota \in \text{itags}(B)$ then $\iota \notin \text{rng}(\mathcal{N})$.

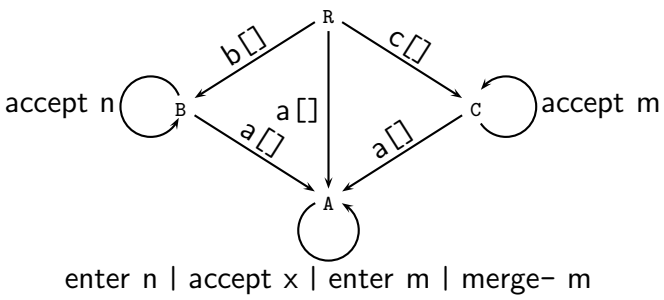
The construction of a shape type which correspond to a FABA predicate $(\mathcal{S},\mathcal{N})$ is presented in Figure 18.6. The set $\text{labels}_{\mathcal{S}}$ is the set of labels contained in $\mathcal{S}$. The set $\text{nodes}_{\mathcal{S}}$ is a set of nodes with the same number of members like $\text{labels}_{\mathcal{S}}$. Two mutually inverse bijections on these two sets are introduced. The set $\text{activecaps}_{(\mathcal{S},\mathcal{N})}(l)$

describes all action prefixes allowed in an ambient labeled by l. Note that we have to construct also original prefixes from their instantiations. As already noted, the construction requires an upper bound on input-bound names allowed in a BA process. This is given by the set of type tags Z. The set $\text{inertcaps}_{(\mathcal{N},Z)}$ describes all action prefixes constructed from those input-bound tags which are never instantiated by communication to any actual value, that is, from communication inert input-bound type tags. Such actions are not contained in FABA results but a shape type needs to describe them. For example, for the FABA result $\mathcal{S} = \{(\star, \mathsf{s2s\ a(b)})\}$ and $\mathcal{N} = \{(\mathsf{a}, \mathsf{a})\}$ it holds that $(\mathcal{S}, \mathcal{N}) \models^{\star} \mathsf{s2s\ a?\{b\}.enter\ b.0}$. But note that "$\mathsf{enter\ b}$" is not contained in $\mathcal{S}$. In fact an arbitrary number of actions constructed from b can be present under "$\mathsf{s2s\ a?\{b\}}$" and the process is still correctly described by $(\mathcal{S}, \mathcal{N})$ (as long as $\mathsf{b} \notin \text{dom}(\mathcal{N})$). The list of inert actions is added to $\text{activecaps}_{(\mathcal{S},\mathcal{N})}(l)$ to form the set $\text{allowedin}_{(\mathcal{N},\mathcal{S},Z)}(l)$. The shape graph $\Gamma_{(\mathcal{S},\mathcal{N},Z)}$ connects the nodes from $\text{nodes}_{\mathcal{S}}$ accordingly to the ambient hierarchy described by $\mathcal{S}$. Finally the action types from $\text{allowedin}_{(\mathcal{N},\mathcal{S},Z)}(l)$ are added as labels of loops of the node which correspond to l.

EXAMPLE 18.4.4. *Let us demonstrate the construction on the process B from Example 18.1.2 and the* FABA *result for B from Example 18.2.1. We have* $\text{labels}_{\mathcal{S}} = \{\star, \mathsf{a}, \mathsf{b}, \mathsf{c}\}$. *Let us take* $\text{nodes}_{\mathcal{S}} = \{\mathtt{R}, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$ *and* $\text{nodeof}_{\mathcal{S}}$ *such that* $\text{nodeof}_{\mathcal{S}}(\star) = \mathtt{R}$, $\text{nodeof}_{\mathcal{S}}(\mathsf{a}) = \mathtt{A}$, $\text{nodeof}_{\mathcal{S}}(\mathsf{b}) = \mathtt{B}$, *and* $\text{nodeof}_{\mathcal{S}}(\mathsf{c}) = \mathtt{C}$. *The situation with input-bound names is simple because we know that* $\text{itags}(B) = \emptyset$ *and thus we can take* $Z = \emptyset$. *Thus* $\text{inertcaps}_{(\mathcal{N},Z)} = \emptyset$. *We have that*

$$\begin{aligned}
&\text{activecaps}_{(\mathcal{S},\mathcal{N})}(\star) = \emptyset \\
&\text{activecaps}_{(\mathcal{S},\mathcal{N})}(\mathsf{a}) = \{\mathsf{enter\ n}, \mathsf{accept\ x}, \mathsf{enter\ m}, \mathsf{merge\text{-}\ m}\} \\
&\text{activecaps}_{(\mathcal{S},\mathcal{N})}(\mathsf{b}) = \{\mathsf{merge\text{-}\ n}\} \\
&\text{activecaps}_{(\mathcal{S},\mathcal{N})}(\mathsf{c}) = \{\mathsf{accept\ m}\}
\end{aligned}$$

The shape graph looks as follows.



Labels of multiple loop edges of node $\mathtt{A}$ *are merged together by "|". We can see that this graph is slightly less precise that the graph of the principal type presented in Example 18.3.1.* □

The correctness of the construction is expressed by the following theorem. The root node of a constructed shape graph is of course the node $\text{nodeof}_{\mathcal{S}}(\star)$. We see that the theorem allows us to exactly emulate Faba relation $(\mathcal{S},\mathcal{N}) \models^{\star} B$.

Theorem 18.4.5. *Let* $(\mathcal{S},\mathcal{N})$ *be closed. Let* $\text{itags}(B) \subseteq Z$ *and* $\forall \iota \in \text{ftags}(B) \cup \text{ntags}(B)$ *it holds that* $\mathcal{N}(\iota,\iota)$*. Then the following holds.*

$$(\mathcal{S},\mathcal{N}) \models^{\star} B \quad \textit{iff} \quad \vdash (\!|B|\!) : \langle \Gamma_{(\mathcal{S},\mathcal{N},Z)}, \text{nodeof}_{\mathcal{S}}(\star)\rangle$$

18.5 Conclusions and Further Discussions

We showed how to use Poly✶ for flow analysis of Ba. Theorem 18.4.1 says that Poly✶ provides at least the same precision of information as a flow analysis system Faba from the literature. We showed how to exactly emulate the Faba's relation $(\mathcal{S},\mathcal{N}) \models^{\star} B$ in Poly✶ which is important because the relation can potentially be used by some application of Faba.

The original Faba works with a version of Ba containing the choice operator ("+") used to express computation options and with the μ (**rec**) operator to express recursive behavior. These are not currently supported as builtin operators in Meta✶ and Poly✶ but Section 9.3.2 and Section 9.3.1 shows how they can be emulated. Whit this emulations we could extend both embeddings to work with the full Faba and we would achieve the same results as for the restricted Faba. However, the main idea would be the same as in the embeddings in this chapter.

Chapter 19

Details on the Faba Embedding

This chapter contains technical details related to the previous chapter. It can be skipped for the first reading and looked up later, either the whole chapter or just some particular part.

In all the proofs in this section we consider BioDirection $\subseteq$ TypeTag. At first we prove Theorem 18.4.1. The following definition defines a binary relation $\mathrm{nodeunder}_\Pi(l,\chi)$ which is similar to $\mathrm{inamb}_\Pi(l,\iota)$ but its second argument is χ rather that a label.

Definition 19.0.1. *For χ, ι, and the shape predicate $\Pi = \langle \Gamma, \chi' \rangle$ write*

(1) $\mathrm{nodeunder}_\Pi(\star,\chi)$ *when* $\{\chi' \xrightarrow{\varphi_0} \cdots \xrightarrow{\varphi_k} \chi\} \subseteq \Gamma$ *and no* φ_i *contains* `[]`

(2) $\mathrm{nodeunder}_\Pi(\iota,\chi)$ *when* $\{(\chi_0 \xrightarrow{\iota\texttt{[]}} \chi_1 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} \chi)\} \subseteq \Gamma$ *and no* φ_i *contains* `[]` ∎

Theorem (Proof of Theorem 18.4.1). *Let $(\mathcal{S},\mathcal{N})$ be the result of* Faba *analysis for B. Let Π be a* Poly★ *restricted principal $\mathcal{B}$-type of $(\!|B|\!)$. The following holds.*

$$\mathcal{S}_\Pi \subseteq \mathcal{S} \quad \& \quad \mathcal{N}_\Pi \subseteq \mathcal{N} \quad \& \quad (\mathcal{S}_\Pi, \mathcal{N}_\Pi) \models^\star B$$

Proof. *We suppose that the principal restricted $\mathcal{B}$-type Π of $(\!|B|\!)$ was computed using the type inference algorithm from Chapter 11. Let us take the derivation Π_0, …, Π_k of Π, that is, the following sequence with $\Pi_k = \Pi$.*

$$\Pi_0 = \texttt{RestrictGraph(ProcessShape(}P\texttt{))}$$
$$\Pi_{i+1} = \texttt{RestrictGraph(FlowClosureStep(LocalClosureStep(}\Pi_i, \mathcal{B}\texttt{)))}$$

This derivation is also used in the proof of Theorem 12.10.3 and other proofs in Chapter 12. It is not hard to prove the following by the induction on i.

(1) $\mathrm{inamb}_{\Pi_i}(\iota,\varphi)$ *implies* $(\iota,\varphi) \in \mathcal{S}$

(2) $(\chi_0 \xrightarrow{\{\iota \mapsto \iota'\}} \chi_1) \in \Gamma_i$ *implies* $(\iota,\iota') \in \mathcal{N}$

(3) for all ι, ι' *such that* $\mathrm{nodeunder}_{\Pi_i}(\iota, \chi_0)$ *and* $\mathrm{nodeunder}_{\Pi_i}(\iota', \chi_1)$ *and such that there is the flow edge* $\chi_0 \overset{\sigma}{\dashrightarrow} \chi_1$) *in* Π_i *it holds that*

$$\forall \varphi : (\iota, \varphi) \in \mathcal{S} \Rightarrow (\iota', \bar{\sigma}\varphi) \in \mathcal{S}$$

Items (1) and (2) when applied to Π *prove the claim. Item (3) is designed to prove the induction step.* ∎

The following is the left-to-right implication of Theorem 18.4.5. The assumption $\mathrm{ftags}(B) \cup \mathrm{ntags}(B) \subseteq \mathrm{dom}(\mathcal{N}) \cup Z$ is clearly satisfied because $\mathrm{ftags}(B) \cup \mathrm{ntags}(B) \subseteq \mathrm{dom}(\mathcal{N})$. Also $\star \in \mathrm{labels}_{\mathcal{S}}$ by the definition.

Proposition 19.0.2. *Let*

(1) $\mathrm{itags}(B) \subseteq Z$
(2) $\mathrm{ftags}(B) \cup \mathrm{ntags}(B) \subseteq \mathrm{dom}(\mathcal{N}) \cup Z$
(3) $l \in \mathrm{labels}_{\mathcal{S}}$

Then $(\mathcal{S}, \mathcal{N}) \models^l B$ *implies* $\vdash (\!|B|\!) : \langle \Gamma_{(\mathcal{S},\mathcal{N},Z)}, \mathrm{nodeof}_{\mathcal{S}}(l) \rangle$.

Proof. *Let* $\Gamma = \Gamma_{(\mathcal{S},\mathcal{N},Z)}$, *and* $\chi = \mathrm{nodeof}_{\mathcal{S}}(l)$, *and* $\Pi = \langle \Gamma, \chi \rangle$. *Let us prove the claim* $\vdash (\!|B|\!) : \Pi$ *by induction on the structure of* B. *Let*

$B = \mathtt{0}$**:** *Clear.*

$B = B_0 \mid B_1$**:** *From* $(\mathcal{S}, \mathcal{N}) \models^l B$ *it follows that* $(\mathcal{S}, \mathcal{N}) \models^l B_0$ *and* $(\mathcal{S}, \mathcal{N}) \models^l B_1$. *The assumptions of the induction step are clearly satisfied for both* B_0 *and* B_1. *Thus by the induction hypothesis we have that* $\vdash (\!|B_0|\!) : \Pi$ *and* $\vdash (\!|B_1|\!) : \Pi$. *Thus the claim.*

$B = \mathtt{[}B_0\mathtt{]}^{l_0}$*: From* $(\mathcal{S}, \mathcal{N}) \models^l B$ *it follows that* $\mathcal{S}(l, l_0\,\mathtt{[]})$ *and* $(\mathcal{S}, \mathcal{N}) \models^{l_0} B_0$. *Thus we see that the assumptions of the induction step for* B_0 *and* l_0 *are clearly satisfied. Let* $\chi_0 = \mathrm{nodeof}_{\mathcal{S}}(l_0)$. *Thus by the induction hypothesis we have that* $\vdash (\!|B|\!) : \langle \Gamma, \chi_0 \rangle$. *From the construction of* $\Gamma_{(\mathcal{S},\mathcal{N},Z)}$ *it follows that* $(\chi \xrightarrow{l_0\,\mathtt{[]}} \chi_0) \in \Gamma$ *and thus the claim because* $(\!|B|\!) = l_0\,\mathtt{[}(\!|B_0|\!)\mathtt{]}$.

$B = N.B_0$**:** *Suppose, for example,* $N = \mathsf{enter}\ n$. *The proof for other capabilities (communication actions are handled separately) is analogous. Let* $\iota = \overline{n}$. *Furthermore let* $F = \mathsf{enter}\ n$ *and* $\varphi = \mathsf{enter}\ \iota$. *We see that* $(\!|B|\!) = F.(\!|B_0|\!)$ *and* $\vdash F : \varphi$. *Now* $(\mathcal{S}, \mathcal{N}) \models^l B$ *implies* $(\mathcal{S}, \mathcal{N}) \models^l B_0$. *The assumptions of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have* $\vdash (\!|B_0|\!) : \Pi$.

To prove the claim it is enough to prove that $\varphi \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$. *From (2) we know that either* $\iota \in \mathrm{dom}(\mathcal{N})$ *or* $\iota \in Z$. *Suppose*

$\iota \in \mathrm{dom}(\mathcal{N})$**:** *Then we have some* ι' *such that* $\mathcal{N}(\iota, \iota')$*. It follows from* $(\mathcal{S}, \mathcal{N}) \models^l$ $\mathsf{enter}\ n$ *that* $\mathcal{S}(l, \mathsf{enter}\ \iota')$*. It is easy to see that* $\varphi = \overline{\{\iota' \mapsto \iota\}}(\mathsf{enter}\ \iota') \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*. Thus the claim.*

$\iota \in Z \backslash \mathrm{dom}(\mathcal{N})$**:** *Then we have that* $\varphi \in \mathrm{inertcaps}_{(\mathcal{N},Z)}$*. Thus the claim.*

$B = !B_0$**:** *Simply apply the induction hypothesis.*

$B = (\nu n)B_0$**:** *Now* $(\mathcal{S}, \mathcal{N}) \models^l B$ *implies* $(\mathcal{S}, \mathcal{N}) \models^l B_0$*. The assumption of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have* $\vdash (\!|B_0|\!) : \Pi$ *and thus the claim because* $(\!|B|\!) = \nu n.(\!|B_0|\!)$*.*

$B = d\ n?\{m\}.B_0$**:** *Let* $\iota = \overline{n}$ *and* $\iota_0 = \overline{m}$*. Furthermore let* $F = d\ n(m)$ *and* $\varphi = d\ \iota(\iota_0)$*. We see that* $(\!|B|\!) = F.(\!|B_0|\!)$ *and* $\vdash F : \varphi$*. From* $(\mathcal{S}, \mathcal{N}) \models^l B$ *we have that* $(\mathcal{S}, \mathcal{N}) \models^l B_0$*. The assumptions of the induction step for* B_0 *are satisfied (number (3) because* $\iota_0 \in Z$ *follows from* $\iota_0 \in \mathrm{itags}(B)$ *and (1)). Thus by the induction hypothesis we have that* $\vdash B_0 : \Pi$*.*

To prove the claim it is enough to prove that $\varphi \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*. From (2) we know that either* $\iota \in \mathrm{dom}(\mathcal{N})$ *or* $\iota \in Z$*. Suppose*

$\iota \in \mathrm{dom}(\mathcal{N})$**:** *Then we have some* ι' *such that* $\mathcal{N}(\iota, \iota')$*. It follows from* $(\mathcal{S}, \mathcal{N}) \models^l$ $d\ n?\{m\}$ *that* $\mathcal{S}(l, d\ \iota'(\iota_0))$*. We see that* $\varphi = \overline{\{\iota' \mapsto \iota\}}(d\ \iota'(\iota_0)) \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*. Thus the claim.*

$\iota \in Z \backslash \mathrm{dom}(\mathcal{N})$**:** *Then we have that* $\varphi \in \mathrm{inertcaps}_{(\mathcal{N},Z)}$*. Thus the claim.*

$B = d\ n!\{m\}.B_0$**:** *Let* $\iota = \overline{n}$ *and* $\iota_0 = \overline{m}$*. Furthermore let* $F = d\ n\langle m\rangle$ *and* $\varphi = d\ \iota\langle\iota_0\rangle$*. We see that* $(\!|B|\!) = F.(\!|B_0|\!)$ *and* $\vdash F : \varphi$*. From* $(\mathcal{S}, \mathcal{N}) \models^l B$ *we have that* $(\mathcal{S}, \mathcal{N}) \models^l B_0$*. The assumptions of the induction step for* B_0 *are satisfied. Thus by the induction hypothesis we have that* $\vdash B_0 : \Pi$*.*

To prove the claim it is enough to prove that $\varphi \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*. From (2) we know that either* $\iota \in \mathrm{dom}(\mathcal{N})$ *or* $\iota \in Z$ *and the same for* ι_0*. Suppose*

$\iota \in \mathrm{dom}(\mathcal{N})\ \&\ \iota_0 \in \mathrm{dom}(\mathcal{N})$**:** *Then we have* ι' *and* ι'_0 *such that* $\mathcal{N}(\iota, \iota')$ *and* $\mathcal{N}(\iota_0, \iota'_0)$*. It follows from* $(\mathcal{S}, \mathcal{N}) \models^l d\ n?\{m\}$ *that* $\mathcal{S}(l, d\ \iota'\langle\iota'_0\rangle)$*. But now it is easy to see that* $\varphi \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*. Thus the claim.*

$\iota \in Z \backslash \mathrm{dom}(\mathcal{N}) \vee \iota_0 \in Z \backslash \mathrm{dom}(\mathcal{N})$**:** *Then we have that* $\varphi \in \mathrm{inertcaps}_{(\mathcal{N},Z)}$*. Thus the claim.* ∎

The following is the right-to-left implication of Theorem 18.4.5.

Proposition 19.0.3. *Let*

(1) $(\mathcal{S}, \mathcal{N})$ *be closed*

(2) $\mathrm{itags}(B) \subseteq Z$

(3) $\mathrm{ftags}(B) \cup \mathrm{ntags}(B) \subseteq \mathrm{dom}(\mathcal{N}) \cup Z$

(4) $\forall \iota \in \mathrm{ntags}(B) : \mathcal{N}(\iota, \iota)$

(5) $l \in \mathrm{labels}_{\mathcal{S}}$

Then $\vdash (\!|B|\!) : \langle \Gamma_{(\mathcal{S},\mathcal{N},Z)}, \mathrm{nodeof}_{\mathcal{S}}(l) \rangle$ *implies* $(\mathcal{S}, \mathcal{N}) \models^{l} B$.

Proof. *Let* $\Gamma = \Gamma_{(\mathcal{S},\mathcal{N},Z)}$, *and* $\chi = \mathrm{nodeof}_{\mathcal{S}}(l)$, *and* $\Pi = \langle \Gamma, \chi \rangle$. *Let us prove the claim* $(\mathcal{S}, \mathcal{N}) \models^{l} B$ *by induction on the structure of* B. *Let*

$B = \mathtt{0}$**:** *Clear.*

$B = B_0 \mid B_1$**:** *It is easy to see that* $\vdash (\!|B_0|\!) : \Pi$ *and* $\vdash (\!|B_1|\!) : \Pi$. *The assumptions of the induction step are clearly satisfies. Thus the claim follows directly from the induction hypothesis.*

$B = \mathtt{[}B_0\mathtt{]}^{l_0}$*:* *We know that* $(\!|\mathtt{[}B_0\mathtt{]}^{l_0}|\!) = l_0\,\mathtt{[}(\!|B_0|\!)\mathtt{]}$ *and* $\vdash l_0\,\mathtt{[}(\!|B_0|\!)\mathtt{]} : \langle \Gamma, \chi \rangle$. *Thus* $\mathcal{S}(l, l_0\,\mathtt{[]})$ *and* $l_0 \in \mathrm{labels}_{\mathcal{S}}$. *Let* $\chi_0 = \mathrm{nodeof}_{\mathcal{S}}(l_0)$. *Thus we have* $\vdash (\!|B_0|\!) : \langle \Gamma, \chi_0 \rangle$. *The assumptions of the induction step for* B_0 *and* l_0 *are clearly satisfied. Thus by the induction hypothesis we have that* $(\mathcal{S}, \mathcal{N}) \models^{l_0} B_0$ *which together with* $\mathcal{S}(l, l_0\,\mathtt{[]})$ *proves the claim.*

$B = N.B_0$**:** *Suppose, for example,* $N = \mathsf{enter}\ n$. *The proof for other capabilities (communication actions are handled separately) is analogous. Let* $\iota = \overline{n}$. *We know* $\vdash \mathsf{enter}\ n.(\!|B_0|\!) : \Pi$ *and thus there is some* φ *such that* $\vdash \mathsf{enter}\ n : \varphi$. *From the construction of* $\Gamma_{(\mathcal{S},\mathcal{N},Z)}$ *it follows that* $\varphi = \mathsf{enter}\ \iota$, *and* $\varphi \in \mathrm{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$, *and also* $\vdash (\!|B_0|\!) : \Pi$. *The assumptions of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have that* $(\mathcal{S}, \mathcal{N}) \models^{l} B_0$.

To prove the claim it is enough to prove the goal $(\mathcal{S}, \mathcal{N}) \models^{l} \mathsf{enter}\ \iota$, *that is,* $\forall \iota' : \mathcal{N}(\iota, \iota') \Rightarrow \mathcal{S}(l, \mathsf{enter}\ \iota')$. *The goal holds trivially when there is no* ι' *such that* $\mathcal{N}(\iota, \iota')$. *So, let* ι' *be such that* $\mathcal{N}(\iota, \iota')$. *This means that* $\iota \in \mathrm{dom}(\mathcal{N})$ *and thus* $\varphi \notin \mathrm{inertcaps}_{(\mathcal{N},Z)}$. *Hence we know that* $\varphi \in \mathrm{activecaps}_{(\mathcal{S},\mathcal{N})}(l)$. *Thus there are some* φ_0, ι_0, *and* ι_0' *such that* $\mathcal{S}(l, \varphi_0)$, *and* $\mathcal{N}(\iota_0, \iota_0')$, *and* $\iota_0' \in \mathrm{ftags}(\varphi_0)$, *and also* $\{\iota_0' \mapsto \iota_0\}\varphi_0 = \varphi = \mathsf{enter}\ \iota$. *It is clear that* $\varphi_0 = \mathsf{enter}\ \iota_0'$ *and* $\iota_0 = \iota$. *We have* $\mathcal{S}(l, \mathsf{enter}\ \iota_0')$ *and* $\mathcal{N}(\iota, \iota_0')$. *Thus the goal follows from assumption (1) by point (1) of Definition 18.4.3.*

$B = !B_0$**:** *Simply apply the induction hypothesis.*

$B = (\nu n)B_0$**:** *Let* $\iota = \overline{n}$. *We have* $(\!|B|\!) = \nu n.(\!|B_0|\!)$. *Thus it is clear that it holds* $\vdash (\!|B_0|\!) : \Pi$. *The assumptions of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have that* $(\mathcal{S}, \mathcal{N}) \models^{l} B_0$. *To proof the claim it is enough to prove that* $\mathcal{N}(\iota, \iota)$ *which holds by assumption (4).*

$B = d\ n\mathtt{?\{}m\mathtt{\}}.B_0$**:** *Let* $\iota = \overline{n}$ *and* $\iota_0 = \overline{m}$. *We know* $\vdash d\ n\mathtt{(}m\mathtt{)}.(\!|B_0|\!) : \Pi$ *and thus there is some* φ *such that* $\vdash d\ n\mathtt{(}m\mathtt{)} : \varphi$. *From the construction of* $\Gamma_{(\mathcal{S},\mathcal{N},Z)}$

it follows that $\varphi = d\ \iota(\iota_0)$*, and* $\varphi \in \text{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*, and also* $\vdash (\!|B_0|\!) : \Pi$*. The assumptions of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have that* $(\mathcal{S},\mathcal{N}) \models^l B_0$*.*

To prove the claim it is enough to prove the goal $(\mathcal{S},\mathcal{N}) \models^l d\ \iota(\iota_0)$*, that is,* $\forall \iota' : \mathcal{N}(\iota,\iota') \Rightarrow \mathcal{S}(l, d\ \iota'(\iota_0))$*. The goal holds trivially when there is no* ι' *such that* $\mathcal{N}(\iota,\iota')$*. So, let* ι' *be such that* $\mathcal{N}(\iota,\iota')$*. This means that* $\iota \in \text{dom}(\mathcal{N})$ *and thus* $\varphi \notin \text{inertcaps}_{(\mathcal{N},Z)}$*. Hence we know that* $\varphi \in \text{activecaps}_{(\mathcal{S},\mathcal{N})}(l)$*. Thus there are some* φ_0*,* ι_1*, and* ι_1' *such that* $\mathcal{S}(l,\varphi_0)$ *and* $\mathcal{N}(\iota_1,\iota_1')$ *and* $\iota_1' \in \text{ftags}(\varphi_0)$ *and also* $\overline{\{\iota_1' \mapsto \iota_1\}}(\varphi_0) = \varphi = d\ \iota(\iota_0)$*. It is clear that* $\varphi_0 = d\ \iota_1'(\iota_0)$ *and* $\iota_1 = \iota$*. We have* $\mathcal{S}(l, d\ \iota_1'(\iota_0))$ *and* $\mathcal{N}(\iota,\iota_1')$*. Thus the goal follows from assumption (1) by point (7) of Definition 18.4.3.*

$B = d\ n!\{m\}.B_0$**:** *Let* $\iota = \overline{n}$ *and* $\iota_0 = \overline{m}$*. We know* $\vdash d\ n\texttt{<}m\texttt{>}.(\!|B_0|\!) : \Pi$ *and thus there is some* φ *such that* $\vdash d\ n\texttt{<}m\texttt{>} : \varphi$*. From the construction of* $\Gamma_{(\mathcal{S},\mathcal{N},Z)}$ *it follows that* $\varphi = d\ \iota\texttt{<}\iota_0\texttt{>}$*, and* $\varphi \in \text{allowedin}_{(\mathcal{S},\mathcal{N},Z)}(l)$*, and also* $\vdash (\!|B_0|\!) : \Pi$*. The assumptions of the induction step for* B_0 *are clearly satisfied. Thus by the induction hypothesis we have that* $(\mathcal{S},\mathcal{N}) \models^l B_0$*.*

To prove the claim it is enough to prove the goal $(\mathcal{S},\mathcal{N}) \models^l d\ \iota\texttt{<}\iota_0\texttt{>}$*, that is,* $\forall \iota', \iota_0' : \mathcal{N}(\iota,\iota')\ \&\ \mathcal{N}(\iota_0,\iota_0') \Rightarrow \mathcal{S}(l, d\ \iota'\texttt{<}\iota_0'\texttt{>})$*. The goal holds trivially when there is no* ι' *such that* $\mathcal{N}(\iota,\iota')$ *or there is no* ι_0' *such that* $\mathcal{N}(\iota_0,\iota_0')$*. So, let* ι' *and* ι_0' *be such that* $\mathcal{N}(\iota,\iota')$ *and* $\mathcal{N}(\iota_0,\iota_0')$*. This means that* $\iota \in \text{dom}(\mathcal{N})$ *and* $\iota_0 \in \text{dom}(\mathcal{N})$*, and thus* $\varphi \notin \text{inertcaps}_{(\mathcal{N},Z)}$*. Hence we know that* $\varphi \in \text{activecaps}_{(\mathcal{S},\mathcal{N})}(l)$*. Thus there are some* ι'' *and* ι_0'' *such that* $\mathcal{N}(\iota,\iota'')$ *and* $\mathcal{N}(\iota_0,\iota_0'')$ *and* $\mathcal{S}(l, d\ \iota''\texttt{<}\iota_0''\texttt{>})$*. Thus the goal follows from assumption (1) by point (8) of Definition 18.4.3.* ∎

Chapter 20

Conclusions

This chapter concludes the three parts of the thesis. Contributions of the thesis were already discussed in Section 1.8.

In Part I we have presented the Poly✶ system which fixes and extends the previous work of Makholm and Wells [MW05, MW04a]. These fixes and extensions were summarized in Section 9.2. Additional possible extensions and future work topics related to Poly✶ were discussed in Section 9.3.

Part II presents a type inference algorithm and a constructive proof of the existence of principal typings. These results are published for a first time in this thesis. Future work related to the type inference algorithm and the proof of principal typings is closely related to extensions of Poly✶ because the algorithm and the proof have to reflect these extensions. The type inference algorithm presented in Part II is not compositional. To develop a compositional type inference algorithm, which is important because of effectiveness and modularity, is left for the future research.

In Part III, we have demonstrated usage of shape types and we have evaluated their expressiveness. We have presented embeddings of three systems from the literature which were concluded separately in sections 14.5, 16.5, 18.5. As a future work, we would like to (1) relate shape types with other systems which also use graphs to represent types [Yos96, Kön99], and (2) to study the relationship between shape types and session types [Hon93].

Bibliography

[AG99] Martìn Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inform. & Comput.*, 148(1):1–70, January 1999.

[AMW04a] Torben Amtoft, Henning Makholm, and J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. Technical Report HW-MACS-TR-0015, Heriot-Watt Univ., School of Math. & Comput. Sci., February 2004. A shorter successor is [AMW04b].

[AMW04b] Torben Amtoft, Henning Makholm, and J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. In *IFIP TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*, pages 591–604. Kluwer Academic Publishers, 2004. A more detailed predecessor is [AMW04a].

[AW02] Torben Amtoft and J. B. Wells. Mobile processes with dependent communication types and singleton types for names and capabilities. Technical Report 2002-3, Kansas State University, Department of Computing and Information Sciences, December 2002.

[Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theoret. Comput. Sci.*, 335(2-3):131–146, 2005.

[Bek84] Hans Bekič. Towards a mathematical theory of processes. In Cliff B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *LNCS*. Springer-Verlag, 1984.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[Bou97] Gérard Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 228–241, 1997.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[CG99] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pages 79–92, 1999.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, July 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.

[CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000), Tohoku University, Sendai, Japan*, volume 1872 of *LNCS*, pages 333–347. Springer-Verlag, August 2000.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR 1993*, volume 715 of *LNCS*, pages 509–523, 1993.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, pages 128–141, 2001.

[JW09] Jan Jakubův and J. B. Wells. The expressiveness of generic process shape types. Technical Report HW-MACS-TR-0069, Heriot-Watt Univ., July 2009.

[JW10] Jan Jakubův and J. B. Wells. Expressiveness of generic process shape types. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *LNCS*, pages 103–119. Springer Berlin / Heidelberg, February 2010.

[Kön99] Barbara König. Generating type systems for process graphs. In *CONCUR 1999*, volume 1664 of *LNCS*, pages 352–367. Springer-Verlag, 1999.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge Press, 1999.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inform. & Comput.*, 100(1):1–77, September 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i & ii. *Inform. & Comput.*, 100(1):1–77, 1992.

[MW04a] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., School of Math. & Comput. Sci., November 2004. A shorter successor is [MW05].

[MW04b] Henning Makholm and J. B. Wells. Type inference for PolyA. Technical Report HW-MACS-TR-0013, Heriot-Watt Univ., School of Math. & Comput. Sci., January 2004.

[MW05] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *Programming Languages & Systems, 14th European Symp. Programming*, volume 3444 of *LNCS*, pages 389–407. Springer-Verlag, 2005. A more detailed predecessor is [MW04a].

[NNPR07] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, and Debora Rosa. Control flow analysis for bioambients. *ENTCS*, 180(3):65–79, 2007. A preliminary version appeared at Bio-CONCUR 2003.

[Par01] Joachim Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra*, pages 479–543. North-Holland, Amsterdam, 2001.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

[PV05] Catuscia Palamidessi and Frank D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the EATCS*, 87:105–125, 2005.

[RPS+04] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: An abstraction for biological compartments. *Theoret. Comput. Sci.*, 325(1):141–167, September 2004.

[San93] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT*, volume 668 of *LNCS*, pages 151–166. Springer-Verlag, 1993.

[Tur95] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.

[Wel02] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoret. Comput. Sci., 16th Conf.*, volume 1180 of *LNCS*, pages 371–386. Springer-Verlag, 1996.

Index of Metavariables

Index of Rule and Condition Labels

TRep (Fig. 7.3), 51
TSeq (Fig. 7.1), 48
TSet (Fig. 7.1), 48
TStar (Fig. 7.1), 48

U1 (Def. 6.1.3), 37
U2 (Def. 6.1.3), 37

W1 (Def. 3.2.1), 19
W2 (Def. 3.2.1), 19
W3 (Def. 3.2.1), 19
W4 (Def. 3.2.1), 19

Index of Mathematical Objects

General Index