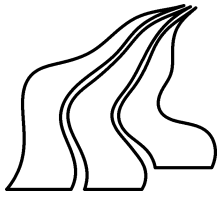
AALBORG UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Fredrik Bajers Vej 7E, 9220 Aalborg Øst, Denmark



Reasoning about Objects using Process Calculus Techniques

PhD thesis

by

Josva Kleist

Supervisors: Hans Hüttel and Kim G. Larsen

January 2000

Pages: 200

Preface

This thesis investigates the applicability of techniques known from the world of process calculi to reason about properties of object-oriented programs.

The investigation is performed upon a small object-oriented language — The ς-calculus of Abadi and Cardelli. The investigation is twofold: First, we investigate translations of ς-calculi into process calculi, with idea that one should be able to show properties of ς-calculus program by showing properties about their translation. Next, we use a labelled transition system adapted to the ς-calculus to investigate the use of process calculi techniques directly on the ς-calculus.

Chapters 1 to 4 contains introduction and background.

In Chapter 5 and 6 we present translations of two ς-calculi into π-calculi. The translation of the untyped Functional ς-calculus in Chapter 5 turns out to be insufficient. Based on our experiences in Chapter 5, we present a translation of a typed Imperative ς-calculus in Chapter 6 that looks promising. We are able to provide simple proofs of the equivalence of different ς-calculus objects using the translation.

In Chapter 7 and 8 we look at direct adaptations of process calculi techniques to the ς-calculus. The work presented in these chapters are of a fairly theoretical nature. In Chapter 7 we investigate the connection between the operational and denotational semantics for a typed Functional ς-calculus. The result is that Abadi and Cardelli's denotational model is sound but not complete with respect to the operational semantics. In Chapter 8 we construct a modal logic for the typed Functional ς-calculus used in the previous chapter. We construct a translation of types to a sub-logic and prove the translation is sound and complete.

Finally, Chapter 9 contains conclusion and directions for further work.

Chapter 5 is based on a paper [HK96] written together with Hans Hüttel, Chapter 6 is based on [KS98] written together with Davide Sangiorgi, Chapter 7 is based on [AHIK97] written together with Luca Aceto, Hans Hüttel and Anna Ingólfsdóttir and Chapter 8 is based on [AHKP97] written together with Dan S. Andersen, Hans Hüttel og Lars H. Pedersen.

Keywords Object-orientation, object calculi, process calculi, ς-calculi, π-calculi, bisimulation, modal logic, semantics through translation, relation between semantics, relation between logic and types.

Dansk Sammenfatning

Ræssonering om objekt orienterede programmer ved hjælp af proces kalkyle teknikker.

Denne afhandling undersøger anvendeligheden af teknikker fra proces kalkyle verdenen til at vise egenskaber ved objekt orienterede programmer.

Undersøgelsen sker på et lille objekt orienteret sprog — Abadi og Cardellis ς-kalkyle. Undersøgelsen er todelt: Først undersøges oversættelser af ς-kalkylen til proces kalkyler med det formål at ræssonere om ς-kalkyle programmer ved hjælp af oversættelsen. Dernæst bruges et mærket transitions system tilpasset ς-kalkylen til at undersøge brugen af proces kalkyle teknikker direkte på ς-kalkylen.

Kapitlerne 1 til 4 indeholder introduktion og baggrundsstof.

I kapitlerne 5 og 6 giver vi en generel oversættelse af programmer skrevet i ς-kalkylen til π-kalkylen. Oversættelsen af den utypede funktionelle ς-kalkyle i kapitel 5 viser sig at være utilstækkelig. På basis af erfaringerne konstrueres en oversættelse af en typet imperativ ς-kalkyle i kapitel 6. Denne oversættelse viser sig at være ret lovende, specielt så giver vi eksempler på hvordan den kan bruges til på relativ simpel vis at vise forskellige ς-kalkyle objekter ækvivalente.

I kapitlerne 7 og 8 ser vi på en direkte tilpasning proces kalkyle teknikkerne til semantikken for ς-kalkylen. Arbejdet i disse kapitler er af ret teoretisk karakter. I kapitel 7 undersøges sammenhængen mellem den operationelle og denationelle semantik for en typet funktionel ς-kalkyle. Resultatet er, at Abadi og Cardelli's denotationelle semantik er sund, men ikke fuldstændig i forhold til den operationelle semantik. I kapitel 8 konstrueres en modal logik til den typede ς-kalkyle fra det forrige kapitel og sammenhængen mellem logikken og type systemet undersøges. Der konstrueres en oversættelse af typer til en del-logik, og det vises at denne oversættelse er sund og fuldstændig.

Endeligt indeholder kapitel 9 konklusioner.

Indholdet i kapitel 5 er skrevet på basis af [HK96] skrevet sammen med Hans Hüttel, kapitel 6 på [KS98] skrevet sammen med Davide Sangiorgi,

kapitel 7 på [AHIK97] skrevet sammen med Luca Aceto, Hans Hüttel og Anna Ingólfsdóttir og kapitel 8 på [AHKP97] skrevet sammen med Dan S. Andersen, Hans Hüttel og Lars H. Pedersen.

Nøgleord og Begreber Objekt orientering, objekt kalkykler, proces kalkyler, ς-kalkylen, π-kalkylen, bisimulering, modal logik, semantik gennem oversættelse, relation mellem semantikker, sammenhæng mellem logik og typer.

Acknowledgments

First, I would l thank my supervisors Hans Hüttel and Kim Guldstrand Larsen, not just for their input on my work, but also for their persistence in trying to drag me into research.

I would also like to thank the co-authors on the papers that formed the basis of this thesis: Luca Aceto, Dan Andersen, Anna Ingólfsdóttir, Lars Pedersen, Davide Sangiorgi, and Hans Hüttel, to whom I owe a lot.

The Aalborg branch of BRICS has been a good place to work, and I want to thank all of the staff and all the visitors for their interest. A special thanks goes to Uwe Nestmann, Luca Aceto and Anna Ingólfsdóttir for their careful proof-reading of this thesis.

Visiting Project Meije at INRIA Sophia Antipolis was a great experience, and I owe special thanks to Illaria Castellani, Silvano Dal-Zilio and Massimo Merro, Davide Sangiorgi and Laurence for making my stay as pleasant as it was.

My family has been a great support to me — without them I would not have made it.

The fincancial support for my studies was provided by BRICS - Basic Research in Computer Science, Centre of the Danish National Research Foundation.

Declaration

This thesis was composed by myself under the supervision of Hans Hüttel and Kim G. Larsen. The work reported herin, unless otherwise stated, is my own.

Aalborg University, January 2000

Josva Kleist

Contents

Chapter 1

Introduction

> *Object: Something material that may be perceived by the senses.*
> (Webster's Dictionary).

In computer science, an object is, roughly speaking, an encapsulation of state and behavior. Objects provide a uniform view on elements in a program; objects can be small, for instance, representing integers, or large, representing a file system. The use of objects has turned out to be a natural way to structure programs and systems.

In recent years, object-oriented techniques have gained wide acceptance in the software industry as a way to structure and develop large and complex systems. This is manifest in the widespread use of C++ [Str91] and the large interest in the Java programming language [GJS96]. The use of objects is not limited to programming languages. There is also a trend towards building distributed systems in an object-oriented manner. For instance, the two most prominent systems for building distributed systems CORBA [OMG] and DCOM [Mic] are both based on communicating objects.

In this thesis, we address the theoretical foundations of object-orientation, especially the use of formal semantics for reasoning about programs. A firm theoretical foundation of a programming language is important because it (hopefully) provides insight into the fundamental concepts of the language in question. A thorough understanding of the semantics of a programming language is vital for writing correct programs in the language. First, this understanding is needed by those implementing the compiler or interpreter for the language. Second, when writing programs, it is important to ensure that they are correct. This not only requires a correct specification of the

application that is being developed, but also requires knowledge about the semantics of the programming constructs being used. Implicit in these two main reasons for focusing on the semantics issues of programming languages are the concepts of program transformation and verification. Program transformation is the process of modifying a program, preferably to a more efficient program, in such a way that the actual result of running the program remains the same. Program transformation can happen behind the scenes performed by the compiler or interpreter. Or, it can be performed by the programmer implementing the application. Verification is the process of ensuring that an implementation meets its specification. This can be done either using informal arguments or using semi-automatic or automatic tools such as theorem provers or model checkers. Semi-automatic or automatic proofs, of course, require that both the specification and the implementation are given in a formal way.

Although the basic concepts in object-orientation appear to be simple, there has been little agreement as to what constitutes the basic parts of an object-oriented language. In fact, it is difficult to distinguish between mere language idioms and real language concepts. This confusion can be seen by regarding the large number of object-oriented languages each having their own sets of features. Also when looking at specific languages there appears to be some sort of confusion. The features included in a language can be quite disparate and might seem to have been added in an ad hoc manner.

These problems are central to the ongoing research in the theoretical foundations of object-oriented programming languages. For procedural and functional languages the theoretical foundations are well-established, supporting the design of, implementation of, and reasoning about programs written in these languages. A similar treatment of object-oriented languages has lacked until recently, but is now emerging. In the following section we give a short overview of some of the theoretical work on object-orientation.

1.1 Theoretical work on object-orientation

We can divide the existing theoretical work on object-oriented languages into three interconnected themes:

i. Semantics of object-oriented languages.

ii. Type systems for object-oriented languages.

iii. Techniques for reasoning about objects.

Semantics The semantics of programming constructs in object-oriented languages is often only described textually (for instance, C++ [Str91], Beta [LMMPN93] and SmallTalk [GR83]). Another technique has been to describe the language through an implementation on some sort of byte-machine (with Java [GJS96] being the most well-known example). More formal methods, such as denotational or operational semantics, also tend to be very low-level, looking more like a description of an implementation.

The aforementioned methods for giving semantics to programming languages have not been entirely successful: The informal textual description often allows several possible contradictory interpretations. The low-level formal descriptions are difficult to use because of their low-level nature, but also because they do not take types (see below) into account. The problem with not considering types in the semantics of object-oriented languages is that a lot of useful properties in object-oriented languages only hold in a typed setting. The type systems of object-oriented languages are also interesting in their own right because of their somewhat peculiar typing rules (at least compared to type systems for functional and imperative languages).

Types In a statically typed language different syntactic entities are given a type, and it is checked that the composition of entities follows certain typing rules. For instance, assuming that addition only works on real numbers, it is checked that + is only used on arguments that are given the type real. A program is *well-typed* if we can check that there are no violations of the typing rules. And a type system is *sound* if no type-error can occur at run-time in a well-typed program.

The study of type systems is usually closely related to that of operational semantics. This is often done in a two-level manner. First, the language is given an untyped operational semantics. Secondly, the type system is defined and proven sound with respect to the operational semantics. For reasoning about the type system this approach is fine, but when it comes to reason about properties of programs that rely on types it quickly becomes cumbersome because the operational semantics is untyped, and it is therefore difficult to do "typed reasoning" on the semantics.

1.1.1 Calculi

Instead of using operational and denotational semantics directly, the current trend in theoretical work on object-orientation is to use some sort of calculus. The semantics of a programming language is then given as a translation of the programming language constructs into the calculus. If the translation is

"well-behaved", one can then use the calculus as a tool for reasoning about the programming language.

A calculus can be thought of as a notation that expresses essential properties of a language or language paradigm. For instance, the (untyped) λ-calculus describes computation as function application through the β-reduction rule:

$$(\lambda(x).M)N \rightarrow M\{^N/_x\}$$

expressing that the (nameless) function $\lambda(x).M$, when given the argument N, will reduce to M with N taking the place of x. The untyped λ-calculus can be equipped with different kinds of type systems that allow us to use the calculus to reason about aspects of typed functional languages.

Various incarnations of the λ-calculus provide a firm theoretical basis for functional programming languages. Most, if not all, aspects of functional programming can be described through extensions of the λ-calculus.

The λ-calculus has also been used to describe object-orientation by among other Compagnoni, Hofmann, Pierce and Turner [CP96, PT93, HP94]. This approach can give a natural understanding of untyped objects as records of functions, but as soon as types are considered, use of the λ-calculus gets extremely complicated.

1.1.2 Object-calculi

The complications of using the λ-calculus as the theoretical foundation of object-oriented languages have led to the development of so-called *object calculi*, that is, calculi that are designed to express the central aspects of object-orientation directly. We can group these calculi according to which features they have. One way to discern the different object-calculi is to see whether they come equipped with a type system or not. Another is to group them according to whether they are sequential or concurrent.

Untyped sequential object-calculi are not seen very often, except as the untyped part of a typed object-calculus. One reason for this is that the untyped λ-calculus can account for most features of untyped objects. (One exception is OPUS [MMS94] that is intended to account for inheritance and encapsulation.)

In a concurrent setting, untyped object-calculi become more interesting since the concurrency aspects themselves pose problems. One such untyped concurrent object-calculus is Nierstrasz' Object Calculus [Nie92], which can be seen as a merge between the λ-calculus and the π-calculus (we will discuss the π-calculus in detail later) plus some additional features for expressing encapsulation.

Of the typed sequential object-calculi, the two most noteworthy are Fisher's Object Calculus [Fis96] and the ς-calculus by Abadi and Cardelli [AC96]. These two calculi are closely related.

The motivations for Fisher's Object Calculus and Abadi and Cardelli's ς-calculus are also the same — to investigate the type systems of object-oriented languages. They both model objects as records of named methods and build method activation the same way. A main difference between them is that Fisher's Object Calculus admits dynamic extension of objects with new methods, whereas the ς-calculus only allows redefinition of already existing methods. This might seem like a minor difference, but it turns out that dynamic extension of objects makes typing much more complicated. Both calculi have already shown their ability to express a large set of features found in object-oriented languages. In Chapter 3 we have a closer look at the ς-calculus, since it will be our model of object-orientation throughout this thesis.

Typed concurrent object-calculi, as the name suggests, aim at describing features of typed concurrent object-oriented languages. The most well-known is probably TyCO [Vas94], for which some theoretical work exist. One particular strand of work on typed concurrent object-calculi is the study of *non-uniform type-systems*. A non-uniform type-system is a type-system that captures the fact, that it might not make sense to call some methods at particular moments during program execution (for instance, it does not make sense to call the pop method on an empty stack object). Non-uniform type-systems are still in their infancy, and we shall in this thesis only consider conventional (uniform) types-systems.

1.1.3 Process-calculi

Concurrent object-calculi have a lot in common with another family of calculi, namely process-calculi and, in particular, the π-calculus.

Process-calculi have been developed to support reasoning about concurrent systems. Common to all process-calculi are therefore syntactic constructs that express parallelism and some kind of synchronization between parallel processes. To support the reasoning about concurrent systems one can use some notion of behavioral relation between processes or logical specification.

A prototypical process-calculus is Milner's Calculus of Communicating Systems (CCS) [Mil89]. We shall give a brief review of CCS, since its simplicity allows an easy introduction to the basic principles behind a process-calculus. In CCS, parallel processes (or systems) are built from simple syntactic constructors including action prefixing ($a.P$ denotes the process P

prefixed with the action a) and parallel composition ($P \mid Q$ is the process P put in parallel with Q). The semantics of CCS is given as a labelled transition system with labels taken from the set of actions (and coactions) plus the special label τ. The basic rules of the operational semantics are the following:

$$\text{(Act)}\ \frac{-}{a.P \xrightarrow{a} P} \qquad \text{(ParL)}\ \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \text{(Com)}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

(Act) expresses that the process $a.P$ can do an action a and become the process P. (ParL) says that if a process P can do an action a and by doing so become P', then it is free to do so, also if it is put in parallel with the process Q (there is of course a similar rule for the component Q of the parallel composition). Finally, (Com) says that if P and Q can perform complementary actions, then they have the possibility of synchronizing (or we could say communicate) resulting in an internal transition, denoted by the label τ.

Behavioral relations Behavioral relations relate processes according to some criteria, usually based on a requirement that related processes must have the same kind of observable behavior. These relations are normally either equivalence relations or preorders. Equivalences allow us to express that two systems are equal from the point of view of the equivalence, and preorders usually express that one system has a more refined behavior than the other.

Equivalences and preorders for process-calculi have been studied extensively and there is a large number of them. The labelled transition semantics allows a very elegant definition of equivalence between processes — called *bisimulation* [Par81, Mil89]. A strong bisimulation is a symmetric binary relation $\mathcal{R}$ over pairs of processes, where it holds that for all pairs $(P, Q) \in \mathcal{R}$, if $P \xrightarrow{a} P'$ then there exists a Q' such that $Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$. In other words two processes P and Q are bisimilar if all actions P can do can be matched by Q and the resulting pairs of processes are again bisimilar (and vice versa).

Not only is bisimulation an elegant way of defining equivalence; bisimulation also provides a nice technique for proving two processes equivalent; that of *co-induction*. To establish the bisimulation equivalence of two processes P and Q, we simply have to come up with a relation $\mathcal{R}$ containing the pair

(P, Q), and show that $\mathcal{R}$ meets the bisimulation criteria (by showing that the relation is closed under transitions).

Another characteristic of process-calculi is that they often come equipped with an algebraic theory. This theory allows us to prove the equivalence of processes by applying algebraic laws instead of having to resort to the definition of equivalence.

Behavioral relations can be used to specify properties of a parallel system by coming up with a specification as a system and then relating the implementation to the specification using some suitable notion of preorder or equivalence. This way of specifying systems is not always very practical since we must come up with *one* specification system covering all aspects of the systems at a chosen level of detail. Instead it is often easier to specify the system by having several specifications, each dealing with specific properties of the systems.

Logical specifications One way to do this is to use a logical specification language. For CCS, Hennesy and Milner introduced a modal logic with the aim of giving an alternative characterization of bisimulation equivalence. The characterization is that two processes are bisimulation equivalent if and only if they satisfy the same set of formulae (have the same properties). Hennesy-Milner-Logic only allows one to state finite properties of a system, but it was later extended by Larsen [Lar90] to allow the specification of infinite properties by adding recursion to the logic.

As a simple example of how modal logic allow us to create a loose specification, assume we want to specify that a system on reception of a coin can return some coffee. Such a property can be specified as: $X \stackrel{\triangle}{=} [coin]\langle\overline{coffee}\rangle X$. A system satisfying the property X should have the behavior that all observations of a *coin* transition can be followed by a $\overline{coffee}$ transition leading to a system satisfying X again. ($[act]F$ is satisfied by a system P if for all *act* transitions the resulting system satisfies F, and $\langle act\rangle F$ is satisfied by P if there exists an *act* transition leading to a system satisfying F.)

The property X is satisfied by the system: $M \stackrel{\triangle}{=} coin.\overline{coffee}.M$, but also by the system $N \stackrel{\triangle}{=} coin.(\overline{coffee}.N + \overline{tea}.N)$. This is because the specification only specifies a *part* of the behavior of a system.

The large number of equivalences and the rich algebraic theory for process-calculi plus the use of logics for specifying properties of systems have attracted the interest of researchers in the field of programming language semantics. An early example was given in [Mil89] where Milner uses CCS to give a semantics for a procedural concurrent language.

CCS is a static calculus in that the communication patterns are fixed. An offspring of CCS, the π-calculus [MPW92] remedies this by allowing the transmission of channels over channels (or *names* as they are called in the π-calculus). This allows a much more dynamic or mobile behavior of a system and makes the π-calculus a lot more useful for modelling object-oriented languages where the behavior also changes as references are passed around. The π-calculus has already been used to give semantics to concurrent object-oriented languages by, among others, Walker [Wal95] and Jones [Jon93]. The problem with these translations is that they do not translate types, and this makes reasoning about the languages in the π-calculus difficult. The typed concurrent object-calculi, as for instance TyCO [Vas94] and the concurrent ς-calculus [GH98], can be seen as an attempt to come up with a way to reason about typed languages using process-calculi. Another solution brought forward by Pierce and Sangiorgi [PS96] is to equip the π-calculus with a type-system. Such a type-system can, if it is advanced enough, make the π-calculus more useful for expressing features of typed object-oriented languages.

Another way to get the advantages of process-calculi into the world of objects is to adopt the techniques directly to the object-calculus. For instance, Gordon and Rees [GR96] have successfully adopted bisimulation to the ς-calculus, resulting in considerably simpler proofs of properties in the ς-calculus.

1.2 This thesis

The goal of this thesis is to deepen the investigation of the applicability of process calculus techniques for reasoning about object-oriented languages and programs. The question we try to answer is:

> *Can one successfully adapt and use process calculus techniques to reason about object-oriented languages?*

Our approach is:

i. To consider the use of the π-calculus to give semantics to variants of the ς-calculus, and examine the usability of this semantics to reason about properties in the ς-calculus.

ii. To work on the adaptation of process calculi to the ς-calculus.

This thesis is divided into three parts. Chapters 2 to 4 form an introductory part, discussing features of object-oriented languages in Chapter 2, introducing the ς-calculus in Chapter 3, and the π-calculus in Chapter 4. Chapters

5 and 6 consider the use of the π-calculus to reason about the ς-calculus. In Chapter 5, we translate the untyped Functional ς-calculus, discovering that the translation we come up with, although correct, does not support reasoning about programs well. In Chapter 6, we turn our attention to the Imperative ς-calculus with a first-order type system, and develop a translation that is more amenable for reasoning about programs. Next, in Chapters 7 and 8, we consider the adaptation of process calculus techniques to the ς-calculus. Chapter 7 considers the relation between Gordon and Rees' labelled transition system for the Functional ς-calculus and Abadi and Cardelli's denotational model. And, in Chapter 8, we build a modal logic on top of Gordon and Rees' labelled transition system and investigate its relation to one of the type-systems for the ς-calculus. Finally, Chapter 9 contains conclusions and directions for further work.

Chapter 2

Object-Orientation in Programming Languages

During the last decade the term object-oriented has been used within most areas of computing. There exist object-oriented languages, databases, operating systems, file systems etc. This not only shows the applicability of object-orientation but also creates confusion as to what it means to be object-oriented. This chapter gives a brief introduction to the basic concepts and terminology of object-orientation as it is used within the programming language community. Readers familiar with object-orientation are invited to skip this chapter and proceed to Chapter 3, where we introduce the model for object-orientation that we shall be working with.

2.1 Advantages of object-orientation

So why use object-orientation and not some other programming paradigm, such as functional or procedural paradigms where there already exists a well developed understanding of the programming concepts? According to Lehrmann Madsen, Møller-Pedersen, and Nygaard [LMMPN93], there are three main benefits of object-orientation. Namely, *real world apprehension*, *stability of design* and *reusability* of both design and implementations.

Real world apprehension. Object-oriented programming is based on the principle that programs, as far as possible, should reflect the part of reality that the programs deal with. This makes it easier to write and understand programs as humans naturally have training in understanding the real world.

This principle applies to most programming paradigms but object-orientation has proven to fit this principle particularly well.

Stability of design. The design of an object-oriented program is often based on a model, called the physical model in [Jac83], of the system that the program is meant to deal with, and it is on top of this model that the functionality of the system is built. As a consequence of the real world apprehension one more easily gets a stable design, because although the functional demands to a system might change several times in the course the program design process, the underlying physical model normally remains stable. And since it is the physical model that forms the scaffolding for the program, major restructuring of the design is not normally needed.

Reusability. In most development organizations there is a desire to reuse existing code in new programs. But often the existing code needs to be modified to fit its new use. This calls for language support for incremental program modification and flexible ways of embedding the reusable parts into their new environment. There seems to be a common agreement that object-oriented languages at least have some features supporting reusability.

2.2 What are objects?

Conceptually, object-orientation is based on an understanding of the world as consisting of objects. A real-world object (such as a car) can consist of other objects (wheels, engine etc.). Objects have properties or uses (for instance a car can be used to drive from one place to another). In analogy with this view of the world, *objects* in object-oriented languages consist of *variables* holding (references to) other objects and *methods* that perform computations on the object. The set of methods and variables in an object are commonly referred to as the attributes of the object. To refer to an attribute within an object the most common notation is the dot-notation, e.g. writing $a.\mathrm{m}(b_1, \ldots, b_n)$ to activate the method named m in the object a with parameters b_1 to b_n. The expression $a.\mathrm{m}(b_1, \ldots b_n)$ is also often referred to as sending to a the message m, indicating that an object is an autonomous entity that we from the outside can request to perform certain operations.

The reason why variables hold references to other objects instead of objects themselves, is that it can be advantageous to share objects. For instance, consider a set of objects representing bank-accounts with each account having an owner. Here it would be rather inexpedient if the variable in an object holding information about the owner denoted an embedded object. For if two accounts had the same owner, the data for the owner would be duplicated and for instance an address change would result in that all accounts belonging

to the person would need to be updated. Therefore, in most object-oriented languages variables denote references to objects and assignment and parameter passing work at the level of references (some languages like C++ also allow objects to be embedded directly into other objects). Ideally, all variables are reference variables, but for efficiency reasons most languages treat basic entities such as integers as embedded objects.

The first two benefits of object-orientation, real world apprehension and stability of design, come from the foundation of object-oriented languages in our perception of the real world. The third benefit, reusability, does not follow directly from having the programming language based on the notion of objects. Although objects may prepare the ground for reuse some extra language support is needed. As we shall see later in this chapter, when we discuss different kinds of object-oriented languages, it appears that some of the differences between object-oriented languages stem from the way incremental program modification is realized.

In object-oriented languages incremental program modification is supported by two mechanisms: *method update* (also called override) and *inheritance*. Existing methods are reused via inheritance, while method override supports modification of code to its new use. Of course, one must ensure that incremental program modification does not lead to run-time errors; the task of ensuring this is usually left to the type system of the language.

A consequence of incremental program modification is that a method cannot necessarily know what siblings it may have at run-time as they may have been modified. This calls for a dynamic and flexible way for a method to refer to its siblings. For this reason, almost all object-oriented languages have a special identifier, called the *self-identifier*, or just self (in C++ and Java called this), that always denotes the object that a method is embedded in[1]. If we, in a method m, activate a sibling method through self, then the associated method-body is *dynamically* looked up in the object and evaluated. This is often called *late binding* or *dynamic dispatch*, in contrast to *static binding* of methods, where the method call would result in the evaluation of the method-body that was available at the time when m was defined.

[1]Note that in some languages, called delegation based languages, discussed in Section 2.5.1, the self of a method can be bound to a different object than the one where the method is embedded

2.3 Type systems

When an object (or set of objects) is to be used, it must fit into a context. This context expects the object to have certain attributes for the program to function. The task of ensuring this is performed by the *type system*.

Let us briefly discuss the basic principles behind typing in object-oriented languages. In general, one can define *type error* in an object-oriented language as the operation of trying to access a non-existing attribute within some object. Object-oriented languages can be either *statically* or *dynamically typed*; a static type system will ensure the no type error will occur at run-time whereas a dynamic type-system will generate a run-time error that the programmer can anticipate and ensure is handled when a type error occurs.

In order to allow code to be reused, it is often too restrictive to require an exact match on the required attributes and the ones that the object have. Instead it is enough to require that the object has *at least* the attributes required by the context (with extra attributes hidden from the context).

To facilitate our discussion, let us introduce some notation. If a is an object and A a type we shall write $a : A$ to denote that a has the type A. If B is another type we write $A <: B$ if all objects of type A can be used in contexts expecting objects of type B and we say that A is a *subtype* of B (and conversely is B a *supertype* of A). The mechanism used to ensure that a can be used in contexts requiring object of type B is called *subsumption* and states that if A is a subtype of B and a has type A then a also has type B. We write the type of methods as $\mathrm{m}(A_1, \ldots, A_n) : B$, denoting that m is a method accepting n arguments of type A_1 to A_n, returning a result of type B.

Types can be composite, for instance the type of a tuple (a, b), where $a{:}A$ and $b{:}B$, would be (A, B). The subtype relation between two composite types is usually derived from the subtype relation between the components of the types. Let $A(X)$ denote a composite type that has X as one of its components. We say that $A(X)$ is *covariant* in the component X, if $B <: C$ implies $A(B) <: A(C)$, *contravariant* if $C <: B$ implies $A(B) <: A(C)$, and *invariant* if $B <: C$ and $C <: B$ implies $A(B) <: A(C)$. For instance, for tuples is $(A, B) <: (A', B')$ if $A <: A'$ and $B <: B'$, that is tuple types are covariant in both components.

As mentioned, one wants a flexible a type system such that objects can be used in as many contexts as possible.

Example 2.1 Consider the following two object types below. What requirements must we make to ensure that $A' <: A$?

```
objectType A{
  var v:B
  method m(C):D

}
```

```
objectType A′{
  var v:B′
  method m:(C′):D′
  method n(E):F
}
```

Since A' should be a subtype of A, objects of type A' should work in contexts expecting objects of type A. This implies that the method m in A' must be able to accept objects that have type C; therefore the type C' must be a supertype of C. What about the result type? Applying the argument as before we can deduce that D' must be a subtype of D. The method n is not used in contexts expecting objects of type A, so we do not need to impose any restrictions on the types E and F.

With respect to the variable v, it can both be read and updated, so this implies that B' must both be a sub- and a supertype of B, meaning B' must denote the same type as B. □

Observe that requirements for methods are only valid if we *cannot* update methods at run-time. To see the problem with dynamic method update, assume $a{:}A'$ and $A' <: A$. By subsumption, a also has type A and can be used in contexts expecting an object of type A. In such a context, we could update the method m with a method returning an argument of type D (not D'). But such an update could result in a type error if we later used $a.\mathrm{m}(c)$ assuming that a had type A' and therefore expecting m to return an object of type D'. Therefore, in the presence of dynamic method update we must in general require invariance in the type of methods.

Although subtyping and subsumption allow a great deal of flexibility in the way we can use objects, in a statically strongly typed object-oriented language it can also cause trouble — in the sense that we can lose type information. To solve this problems, most object-oriented languages includes facilities, called *typecase* (or typecast), to reveal the real type of an object and react according to that. Typecase is normally only used as a last resort, because its use constrains the extensibility of programs by requiring knowledge of which types one can encounter a run-time.

2.4 Class-based languages

The most common object-oriented languages are all class-based; these include languages such as C++ [Str91], Java [GJS96] and SmallTalk [GR83].

Classes serve as skeletons or templates for creating objects, which means that they describe which attributes an object created from a class will have. If an object a is created from a class A we say that a is an *instance* of A.

Example 2.2 Consider the following class:

```
class Account {
  var owner:Person;
  var balance:Real := 0;

  method deposit(amount:Real) {
    self.balance := self.balance+amount;
  }

  method withdraw(amount:Real) {
    self.balance := self.balance−amount;
  }

  method getOwner( ):String {
    return self.owner.name( );
  }
}
```

This class can be used to create objects representing (simple) bank accounts. To create instances of classes one uses the new operator that from a class name will return a reference to a fresh instance of that class. □

Because the code for methods is immutable, the code can be moved from residing within objects into a *method table* shared between all instances of a class, as shown in Figure 2.1. This factorization is often described when discussing class-based languages; but can to some extent be regarded as an implementation technique and, logically, one can just as well consider methods as being embedded directly into the objects (there are other issues, such as variables shared between instances of a class, that might imply that the factorized view is appropriate).

We still have not described how incremental program modification is done in class-based languages. The mechanism used to realize reuse is *inheritance of classes*, where a class can be defined as being an extension of an existing class. If the class A is defined as an extension of B, we say that A is a subclass of B. A subclass will contain all the attributes of its superclass and can add new attributes. Furthermore, it is possible to redefine methods inherited from the superclass, such that the methods can take advantage of the new

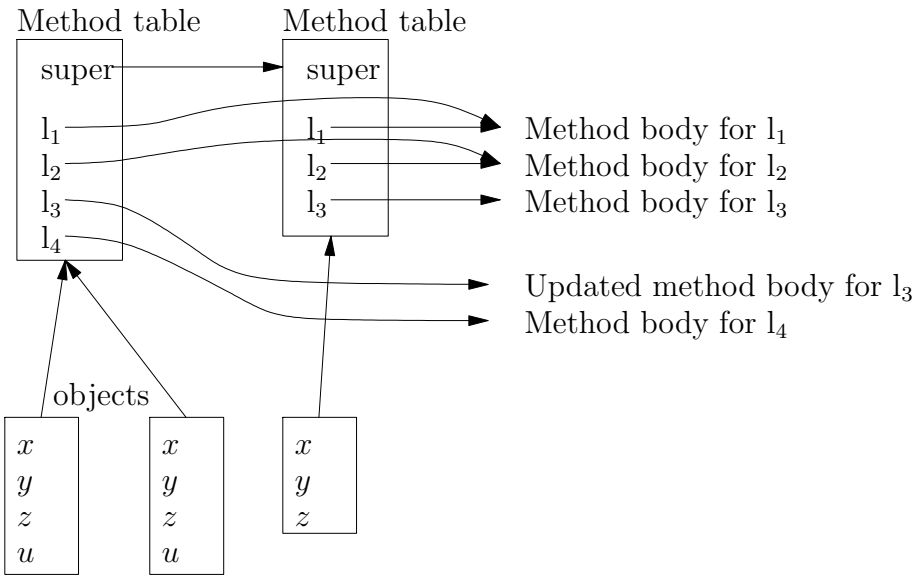


Figure 2.1: Layout model for class-based object-oriented languages

attributes. When redefining a method it can be advantageous to be able to redefine the interface of the method, in most statically typed languages this is disallowed but there exist languages that allow redefinition of signatures.

Example 2.3 Below we create a new class to represent check accounts by extending the account class of Example 2.2.

```
class CheckAccount inherits Account {
  var nbChecks:Integer := 0;

  method cashCheck(amount:Real) {
    self.nbChecks := self.nbChecks+1;
    self.withDraw(amount);
  }
}
```

□

When redefining a method, one might want to refer to the old definition of the method; to this end, most languages have a way to refer to methods in superclasses. There is no common agreement on how do this — for instance in some languages the keyword super is used to refer to the immediate superclass

and in other languages one prepends the method name with the name of the class where the older definition are found. This feature is normally only available within the method definitions, and we can therefore still keep the view that methods are embedded within objects.

Some languages permit inheritance from more than one direct ancestor, called *multiple inheritance*. Multiple inheritance is, in principle, not more difficult than single inheritance except for two minor complications. Firstly, one needs a way to handle name-clashes if two different superclasses use the same name. Secondly, there is the possibility of inheriting a superclass through two different branches in the class-hierarchy, as depicted in Figure 2.2. This gives the problem of whether one gets the variables of the common superclass duplicated or not.

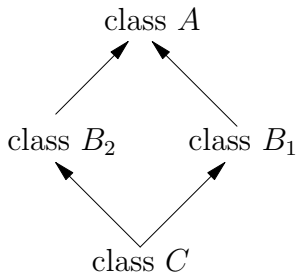


Figure 2.2: Two different branches inheriting the same superclass.

2.4.1 Types in class-based languages

As mentioned earlier in this chapter, one wants a flexible a type system. For instance can instances of the *CheckAccount* class without problems be used in programs designed to work on instances of the *BankAccount* class. Most class based languages permit such cases identifying subclassing and subtyping by having the following rule in their type-system: *If B is a subclass of A, then the type of instances of B is a subtype of the type of instances of A.* That is, the type-system must conform to the class hierarchy. As it is common, we shall in the following identify the type of instances of a given class with its class name.

Identifying subclassing and subtyping naturally implies that the requirements necessary to ensure soundness of subtyping, stated in Section 2.3, now becomes requirements to what modifications we can make to inherited methods.

Flexibility in the way we allow to redefine methods is one way to support reuse of code. Another possibility is to extend the type system to allow existing code to be used more freely.

One such extension has to do with objects that have methods returning self as result. Consider the following two point classes:

```
class Point {
  var x,y:Real;

  method move_x(offset:Real):Point {
    self.x := self.x+offset;
    return self;
  }
}
```

```
class CPoint inherits Point {
    var color:Integer;

    method setColor(newColor:Integer){
      self.color := newColor;
    }

    method move_x(offset:Real):CPoint{
      ...
    }
}
```

In *Point*, we have defined the move_x method to return self. There is one slight problem with the definition of the move_x method — that the type of the result is declared to be *Point*. This implies that one cannot, for instance, call the setColor method after a move_x operation on a *CPoint* (colored point) instance. As we have argued in Section 2.3, it would be sound to redefine the signature of the move_x method in the *CPoint* class to return a *CPoint* instance. But doing so would be inappropriate, since one would have to redefine the move_x method in *every* new class that inherits from *Point*. To remedy this problem some languages, of which the most well-known is Eiffel [Mey92], introduce the notion of *self-types*. If we let Self denote the type of the current self we can define the move_x method as:

```
method move_x(offset:Real):Self{
  self.x := self.x+offset;
  return self;
}
```

and now the move_x method will return a *CPoint* object when invoked on a *CPoint* object. Self-types work fine with the result type of methods, but in general it would be unsound to allow self-types in method arguments and as the type of variables if we require that the subtype relation conforms to the inheritance relation. This has lead to proposals where subtyping does not follow subclassing [CHC90].

One should be aware that the use of self-type requires special care when used in combination with reuse of methods. If, for instance, the move_x method instead of returning self returned a new instance of the *Point* class, as in

```
method moveWrong_x(offset:Real):Self{
  self.x := self.x+offset;
  return new Point;
}
```

Obviously, this is all very fine when the moveWrong$_x$ method resides in the *Point* class. But, when we inherit the moveWrong$_x$ method to the *CPoint* class, the result generated by the method is no longer of type Self. In order to avoid such problems, the type system therefore needs to ensure that if a method has result type Self, then the result returned is either self or a modified self.

2.5 Object-based languages

The main difference between class- and object-based languages is that object-based languages have objects as the only basic concept; there are no classes around to act as schemas for creating objects.

Object-based languages have not been around as long as class-based ones and there is therefore still some disagreement as to what the basic features of an object-based language are. In this section, we shall describe the most common features of object-based languages. For a thorough overview of possible design decisions see [CDM92].

Without classes to facilitate the creation of objects, object-based languages must be able to create objects directly, as shown the following example.

Example 2.4 Below we create an object in an object-based language and bind it to a variable named *account*.

```
var account:Account := object {
  var owner:Person;
  var balance:Real := 0;

  method deposit(amount:Real) {
    self.balance := self.balance+amount;
  }

  method withdraw(amount:Real) { ... }
  method getOwner( ):String { ... }
}
```

□

Most object-based languages have originated in the SmallTalk and AI communities, which have a tradition for favoring flexibility instead of security at run-time, and most object-based languages are therefore dynamically typed, but statically typed object-based languages are emerging. Strongly typed object-based languages have very simple type-systems; normally only supporting object-types, subtyping and subsumption. In our examples, we shall assume the existence of such a type-system and will therefore annotate variables with their types.

It is quite simple to simulate the way classes are used as a schema for creating objects, we can simply wrap the code that creates anonymous objects into a procedure that on invocation returns a new object, this way replacing the new operation with a procedure-call.

If all we could do in an object-based language was to create objects from scratch, possibly wrapping them into a procedure in order to create several objects of a certain shape, an object-based language would be nothing more than a class-based language without inheritance to support incremental program modification. Therefore, most object-based languages offer the ability to *clone* existing objects and to modify the attributes of objects. Cloning creates a shallow copy of an object with sharing of attributes between the clone and the original object. Languages supporting cloning and attribute modification are called prototype-based languages because of a very distinctive manner of writing programs. In prototype-based languages, one has a set of prototypes from where one creates clones and these clones can then be modified.

Cloning and attribute override is one way to support incremental program modification, but its not necessarily flexible enough, because one cannot change the number of attributes in the object one clones from. Several proposals exists to allow more flexibility in the way one can define the behavior of objects, we shall in the following discuss two of these possibilities: *delegation* and *embedding*.

2.5.1 Delegation-based languages

Most object-based languages have some form of *delegation*, including the language Self [US91] which is one of the most well-known object-based languages. Delegation can be thought of as a message send, but without having the self identifier bound to the receiver of the message. This implies that on subsequent activation through self the activation will be performed on the original receiver.

Delegation can happen either implicit or explicit. In a language supporting implicit delegation, an object can appoint an object as its parent object,

and if an attribute is not found in an object, the search continues following the parent link.

Example 2.5 Consider the following two objects ob_1 and ob_2.

```
var ob1:OB1 := object{
  method m1(){
    ...
    self.m2();
    ...
  }
  method m2(){ ... }
};

var ob2:OB2 := object delegates to ob1 {
  method m2(){ ... }
};

ob2.m1();
```

The object ob_2 delegates to ob_1. The final line in the program example requests for the activation of the m_1 method in ob_2. Since the method is not present in ob_2, the request is delegated to ob_1 where the method is found and activated.

As we have sketched, the code of m_1 contains an activation of m_2 on self. Since ob_2 delegates to ob_1 and the original activation of m_1 was on ob_2, self is bound to ob_2. Therefore, the activation of m_2 in m_1 will result in the activation of m_2 in ob_2. □

A possible extension of implicit delegation is the ability to dynamically change which object one delegates to.

If a language has explicit delegation, the programmer can explicitly delegate a message. This gives the programmer more control over when delegation is performed. Explicit delegation can also be used to achieve an effect similar to that of calling a method in a superclass in a class-based language.

Example 2.6 Below we sketch how to create prototype objects that create bank- and check-accounts in a language based on implicit delegation.

```
var accountProto:Account := object {
  var balance:Real := 0;
  var owner:Person;
```

```
  method withdraw(...){ ... }
  ...
}

var checkProto:CheckAccount := object delegates to accountProto {
  var balance:Real := 0;
  var owner:Person;
  var nbChecks:Integer;

  method cashCheck(amount:Real) {
    self.nbChecks := self.nbChecks+1;
    self.withDraw(amount);
  }

}
```

What is interesting about this example is, that in order to make the *checkProto* object work, we need to redeclare the *owner* and *balance* variables, if we did not, then all clones of *checkProto* would end up having the same owner and balance. It is also worth noting, that with this structure we cannot, for instance, change the behavior of the withdraw method for all objects cloned from *accountProto*. This can be solved by having a special prototype object containing only the variables from *accountProto* that then delegates to *accountProto* and clone new objects from this prototype object. We can carry this even further by removing the variables from *accountProto* since they will no longer be used. Eventually, we end up with a definition of *accountProto* that is similar a method table from a class-based language. □

Delegation gives the programmer extreme flexibility, because by changing the behavior of *one* object that other objects delegates to, the programmer will change the behavior of these objects. But its expressiveness does not come without a price: Delegation creates a web of dependencies between objects, making it difficult to anticipate the consequences of program changes and difficult to reason about the behavior of programs.

2.5.2 Embedding-based languages

The previous section showed that delegation is a very powerful mechanism; perhaps too powerful. This leads us to consider a principle for reuse of objects, called *embedding*, that does not create this dependency between the donor object and the one receiving the donation.

One of the advantages of languages based on embedding is that programs written in these languages are easier to implement on a distributed machine architecture, since objects do not need links to their parent or class. Embedding-based languages are relatively new and only a few have emerged, among them Obliq [Car95] which is a distributed object based language.

Just as for delegation-based languages, we can talk about implicit and explicit embedding. Implicit embedding states an object as donor object and creates the new object as an extension of the donor object, possibly redefining some of the donor objects attributes.

Example 2.7 The following program shows how our *checkAccount* object could be created by embedding the *account* object and thereby inheriting all its attributes.

```
var account:Account :=
  object {
    var owner:Person;
    var balance:Real := 0;

    method deposit(amount:Real) {
      self.balance := self.balance+amount;
    }

    method withdraw(amount:Real) { ... }

    method getOwner( ):String { ... }
  }

var checkAcc:CheckAccount :=
  object embeds account {
    var nbChecks:Integer := 0;
    method cashCheck(amount:Real) { ... }
  }
```

□

Explicit embedding gives more flexibility on what is embedded. For instance, if we would like to redefine the withdraw method in the *checkAccount* object, using implicit embedding, we would have to override the withdraw method we got from the *account* object without the ability to refer to it. Instead explicit embedding allows us to refer to embedded attributes in the code of our object, much like we can do in class-based languages when referring to methods defined in superclasses.

Embedding based languages lacks, compared to delegation based languages, the ability to share methods between several objects and make changes to such methods that affect all objects. To remedy this, some languages like Obliq have a feature called *aliasing* or *redirection*. Redirection is almost like delegation, except that self is rebound in the target method. To see the difference between delegation and redirection, assume that we have two objects a and b both having methods m_1 and m_2, and that the method m_1 in b calls m_2. Furthermore, assume that a redirect m_1 to m_1 in b. Let us consider the term $a.m_1$. The difference between redirection and delegation is that, with redirection, when evaluating m_1's body of b, self is bound to b, whereas with delegation self is bound to a. This implies that when a call of m_2 in m_1 in b happens, it is the body found in b that is evaluated. So, with redirection the expression $a.m_1$ behaves as if it was $b.m_1$.

One might think that redirection of a method m_1 in an object a to a method m_2 in an object b, could be done just by letting the body of m_1 in a be a call $b.m_2$. But this is not true in the presence of method update, because if we update m_1 in a then we really change m_1 and not m_2 in b.

2.6 Summary

In the previous discussion, the reader might feel that there are topics missing from the discussion that are also important when considering object-oriented languages. One such topic is encapsulation mechanisms. The reason why we have omitted encapsulation is that, although encapsulation is important for object-oriented languages, encapsulation is related not only to object-oriented languages. Another interesting topic, that we have not discussed, is concurrency. This topic has been omitted for the same reason as encapsulation. Namely, that the concurrency features of object-oriented languages are not that different from those found in other languages supporting concurrent programming[2].

There is one class of object-oriented languages that we have not discussed. It is languages with multiple dispatch such as CLOS [KG89]. The reason for not discussing them is that their model of objects is somewhat different from the view presented in Section 2.2.

All the examples of different object-oriented languages show at least one common pattern: At run-time, we can (or at least would like to) think of objects as being autonomous entities. The class of languages that seems to be farthest away from this pattern is the delegation-based ones. However, it

[2]We deliberately disregard the debate on whether objects are just special kinds of concurrent processes or the other way around.

appears that most delegation-based programs have a structure, where delegation is used as a shared repository for methods, thereby resembling the structure of programs written in class-based languages.

It does therefore seem natural, when we want to assess, whether process calculus techniques can be used to reason about object-oriented programs, to consider a setting where we deal directly with objects. Instead of coming up with our own model, we choose to work with one of the already developed calculi for object-orientation, which is the topic of the following chapter.

Chapter 3

The ς-calculus

3.1 Formalizing object-orientation

As our aim is to study the applicability of process calculus techniques to reason about object-oriented programs, we need to be precise about our representation of object-oriented programs. Generally, we have three possibilities:

- An existing object-oriented language.
- An existing object calculus.
- Building our own formalism.

With the large number of existing languages and calculi, building our own notation seems to be unnecessary and uncalled for. This leaves us with a choice between using a fully-fledged programming language or an object calculus. Working with a programming languages is problematic for several reasons. First of all, most real world object-oriented programming languages contains a lot features that have nothing to do with object-orientation, but which are useful when programming in the language. If we are to consider also these features, this would lead to an unnecessary complication of our work. Secondly, most object-oriented languages lack the formally defined semantics that is needed in order to reason formally about properties of programs written in the languages. Furthermore, in the previous chapter, we have argued that objects are the important entities, when we want to consider techniques for reasoning about object-oriented programs. It does therefore seem natural to consider process-calculi in a pure object-based setting. This could,

of course, be done using some object-based language, but especially for the object-based languages there is no agreement on what the basic constructs are. So, for our study, we choose to work with variations of the ς-calculus [AC96].

Working with an object calculus simplifies things a lot, and one may argue that techniques developed in this setting might not be adaptable to problems in real programming languages. On the other hand, any problem that we encounter in this simple setting, we will most likely also encounter when considering real programming languages.

We have chosen the ς-calculus as our basic object-oriented model primarily because of its clearly object-oriented features and that it has already shown its ability to model features of both object- and class-based languages. A second, more pragmatic reason is that the ς-calculus is actually a family of several calculi built on top of a common untyped core calculus. This allows us to start our study in a simple untyped setting and then, building on top of the knowledge obtained there, continue with typed versions of the ς-calculus.

In the following sections we shall give a brief introduction to the basic parts of the ς-calculus. Our overview is based on the presentation in Abadi and Cardelli's monograph: A Theory of Objects [AC96]. We start out by introducing the untyped ς-calculus, then proceed by adding types and finally discuss the algebraic theory developed by Abadi and Cardelli to reason about ς-calculus expressions.

3.2 The untyped ς-calculus

3.2.1 Syntax and informal semantics

The ς-calculus is very simplistic; the basic ingredients are objects consisting of named methods, method activation and method update. We do not even have variables in objects, but they can be seen as a derived concept, since we can model variables as methods that do not use self. The only extra syntactic construct we need in the ς-calculus is a variable in methods to denote self. Terms in the ς-calculus are built according to the syntax found in Table 3.1

Here $x_i \in \mathbf{SVar}$ ranges over self variables and $\mathrm{l}_i \in \mathbf{MNames}$ ranges over method names. We let $\mathsf{m}(a)$ denote the set of method names and $\mathsf{fv}(a)$ the set of free self variables in a. Objects and self variables are the values of the ς-calculus, and we let v range over these. We shall identify terms up to alpha-conversion of bound variables.

Informally, a method activation $a.\mathrm{l}_j$ is evaluated by first reducing a to some object (value) $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ and then evaluating the method b_j bound

a	::=	$[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$	object (value)
	|	x	self variable
	|	$a.\mathrm{l}$	method activation
	|	$a.\mathrm{l}{\Leftarrow}\varsigma(x)b$	method update

Table 3.1: The syntax of the ς-calculus.

to l_j with the enclosing object bound to the self variable x_j. A method update (also called method override) $a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b$ also first reduces a to an object and then updates the method bound to l_j with the new method $\varsigma(x)b$.

As mentioned earlier, we can regard object variables as a method that do not use self. We call such a method a field, using the notation $\mathrm{l}{=}b$ to denote $\mathrm{l}{=}\varsigma(x)b$ for some $x \notin \mathsf{fv}(b)$. We write field update as $a.\mathrm{l}{:=}b$, denoting $a.\mathrm{l}{\Leftarrow}\varsigma(x)b$ for some $x \notin \mathsf{fv}(b)$.

3.2.2 Semantics

The semantics of the ς-calculus, that we consider, departs from that of Abadi and Cardelli in that we use a small-step reduction semantics, since it will make reasoning about objects easier later on.

We first give the semantics of the basic reductions for objects, shown in Table 3.2.

Let a denote the object $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$, then:

$$
\begin{array}{lcll}
a.\mathrm{l}_j & \rightsquigarrow & b_j\{{}^{a}/_{x_j}\} & j \in I \\
a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b & \rightsquigarrow & [\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}] & j \in I
\end{array}
$$

Table 3.2: The reduction semantics for the Functional ς-calculus

The activation of the method l_j results in the method body being activated with the self variable being replaced with the enclosing object. Method override results in an object with the overridden method replaced with the new method.

For composite expressions we use a leftmost-innermost reduction strategy. This is expressed using reduction contexts [FF86]. A context $C[\cdot]$ denotes a ς-calculus term with a hole, where a term a can be plugged into, written $C[a]$. The syntax of contexts is given by:

$$C[\cdot] ::= C[\cdot].\mathrm{l} \mid C[\cdot].\mathrm{l}\Leftarrow\varsigma(x)b \mid [\cdot]$$

Leftmost-innermost reduction can then be specified using the following rule:

$$\frac{a \rightsquigarrow a'}{C[a] \rightsquigarrow C[a']}$$

Leftmost-innermost reduction implies that in a term $a.\mathrm{l}$ or $a.\mathrm{l}\Leftarrow\varsigma(x)b$, a is always reduced to an object value before activating or overriding a method. If $a \rightsquigarrow^* [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ for some object value, we say that a converge, written $a{\Downarrow}$. If a has an infinite reduction sequence, we say that a diverge, written $a{\Uparrow}$. Finally, if a term that is not a value cannot reduce, such as the term $[].\mathrm{l}$, we say that the term is stuck.

In the basic reductions, we require that method update on a label l is only performed if the object already contains a method bound to l. In an untyped setting it would not lead to problems if we allowed addition of methods to an object. However, as we shall see later, this would lead to problems when we start considering typing issues.

Example 3.1 To give an intuition of how the ς-calculus works, we shall present a few simple examples (taken from [AC96])

$$\begin{array}{llll} \text{let } a & = & [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}] & \text{then } a.\mathrm{l} \rightsquigarrow x.\mathrm{l}\{{}^{a}\!/_{x}\} = a.\mathrm{l} \rightsquigarrow \cdots \\ \text{let } a' & = & [\mathrm{l}{=}\varsigma(x)x] & \text{then } a'.\mathrm{l} \rightsquigarrow x\{{}^{a'}\!/_{x}\} = a' \\ \text{let } a'' & = & [\mathrm{l}{=}\varsigma(y)y.\mathrm{l}\Leftarrow\varsigma(x)x] & \text{then } a''.\mathrm{l} \rightsquigarrow a''.\mathrm{l}\Leftarrow\varsigma(x)x \rightsquigarrow a' \end{array}$$

The object a shows how we can get infinite behavior through the use of self variables. The object a' is a kind of "identity object", where $a'.\mathrm{l}$ evaluates to a'. The object a'' shows how an object can modify itself by performing a method override on a self variable. □

The semantics of objects given in this section is called *stateless* or *functional*, because a method update creates a *new* object instead of modifying the existing. This is in contrast to most object-oriented languages, which are called *statebased* or *imperative*, where update modifies the object directly. We shall later in this thesis (in Chapter 6) consider an imperative semantics for the ς-calculus, but we start out with the functional semantics, as it is simpler to understand.

Although the ς-calculus is simple, it is in fact Turing-powerful as shown by the following translation of the untyped lazy λ-calculus into the ς-calculus.

$$\begin{aligned}
[\![x]\!] &\triangleq x \\
[\![b(a)]\!] &\triangleq [\![b]\!] \cdot [\![a]\!] \text{ with } p \cdot q \triangleq (p.\text{arg}:=q).\text{val} \\
[\![\lambda(x)b]\!] &\triangleq [\text{arg}=\varsigma(x)x.\text{arg},\ \text{val}=\varsigma(x)[\![b]\!]\{x.\text{arg}/x\}]
\end{aligned}$$

The translations builds upon the idea that we represent a function $\lambda(x)b$ as an object having a field arg and a method val. The field val contains the function body and the field arg is used to hold the argument to the function. Upon application, first the field arg is updated to contain the argument to the function and then the method val containing the function body is activated. This translation is sound in the sense, that β-reduction $(\lambda(x)b)(a) \to b\{a/x\}$ is mimicked (up-to extraction of a from the field arg) by the translation (see [AC96, Section 6.3]).

Knowing that we can represent the λ-calculus in the ς-calculus allows us to use λ-notation to give arguments to methods, as in

$$a = [\text{xp}=0,\ \text{mv}_x=\varsigma(s)\lambda(x).(s.\text{xp} := s.\text{xp} + x)]$$

where $a.\text{mv}_x(7) \rightsquigarrow^* [\text{xp}=7,\ \text{mv}_x=\varsigma(s)\lambda(x).(s.\text{xp} := s.\text{xp} + x)]$.

3.3 Type systems

One of the main motivation for the ς-calculus is the study of various type systems for object-oriented programming languages within a unified framework. In this thesis, we consider two basic type systems: a simple first order type system with subtyping, named $\mathbf{Ob}_{1<:}$, and an extension where we add recursive types, named $\mathbf{Ob}_{1<:\mu}$.

3.3.1 The first order type system

The first type system for the ς-calculus is a very simple one. The only types are object types given by the grammar:

$$A ::= [\text{l}_i{:}B_i\ ^{i\in I}]$$

Essentially, an object has type $[\text{l}_i{:}B_i\ ^{i\in I}]$, if it has methods labelled l_i for $i \in I$ and on activation of a method l_j the result is of type B_j.

To simplify reasoning about the type system, we extend the grammar of objects to contain type annotations on the binding of self-variables, such that an object now is written as $[\text{l}_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in I}]$ and method update is written as $a.\text{l}{\Leftarrow}\varsigma(x{:}A)b$.

3.3.2 Assigning types to objects

$\mathbf{Ob}_{1<:}$ has two kinds of judgments: Type judgments and subtyping judgments.

Type judgments, shown in Table 3.3, are of the form $\Gamma \vdash a{:}A$ and state that the object a has type A under the assumptions in Γ, where Γ describes typing assumptions for free self variables. For instance, $\Gamma(x) = A$ states that we assume that the free self variable x has type A. If Γ is empty, we sometimes just write $a{:}A$ instead of $\emptyset \vdash a{:}A$. Whenever the typing assumptions in Γ are extended with the additional assumption $x{:}A$, we write this as $\Gamma[x{:}A]$ (assuming here that no assumption about the type of x occurs in Γ).

An object a has type A under the set of assumptions Γ, if $\Gamma \vdash a{:}A$ can be inferred from the type assignment rules. The most interesting rule of the type system is the rule (Object), which states that in order to give an object $a = [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I}]$ the type $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$, we must be able to give each method within a its corresponding result type in a type environment, where we assume that the self variable of the method already has the type A.

The type system $\mathbf{Ob}_{1<:}$ also incorporates a notion of *subtyping*, which intuitively captures the idea that some types are more general than others. The expression $\vdash A <: B$ denotes that A is a subtype of B and thus that objects of type A may be used in lieu of objects of type B (as stated in rule (Subsump)). The subtype judgment for $\mathbf{Ob}_{1<:}$ is very simple, simply stating using the following rule

$$(\textsc{Sub Obj})\quad \frac{J \subseteq I}{\vdash [\mathrm{l}_i{:}B_i\ ^{i\in I}] <: [\mathrm{l}_i{:}B_i\ ^{i\in J}]}$$

that when $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$ is a subtype of $B = [\mathrm{l}_i{:}B_i\ ^{i\in J}]$ then A is a "longer" type.

The soundness of the type system is proved as a *subject reduction theorem*:

Theorem 3.1 (Subject reduction for $\mathbf{Ob}_{1<:}$) *Let a be a closed term. If $\emptyset \vdash a{:}A$ and $a \rightsquigarrow b$ then $\emptyset \vdash b{:}A$.*

This theorem ([AC96, Th. 8.3-7] adapted to a small-step semantics) states that types are preserved under reduction — if a closed term a has a type A, and a reduces to b, then b will also have the type A.

Subject reduction guarantees that a well-typed ς-calculus term cannot get stuck. Because the only way an expression can get stuck is if we try to call or override a non-existing method. But if an expression a is given a type A, stating that the object has some method labelled l, and $a \rightsquigarrow^* v$, then v also has type A, and from the rule (Object) we see that this can only be the case if v has a method labelled l.

$$\text{(Var)}\quad \frac{\Gamma(x) = A}{\Gamma \vdash x{:}A} \qquad \text{(Select)}\quad \frac{\Gamma \vdash a{:}[\mathrm{l}_i{:}B_i\ ^{i\in I}] \quad j \in I}{\Gamma \vdash a.\mathrm{l}_j{:}B_j}$$

$$\text{(Object)}\quad \frac{\forall i \in I \quad \Gamma[x_i{:}A] \vdash b_i{:}B_i \quad A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]}{\Gamma \vdash [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I}] : A}$$

$$\text{(Update)}\quad \frac{\Gamma \vdash a{:}A \quad \Gamma[x{:}A] \vdash b{:}B_j \quad j \in I \quad A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]}{\Gamma \vdash a.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)b : A}$$

$$\text{(Subsump)}\quad \frac{\Gamma \vdash a{:}A \quad \vdash B <: A}{\Gamma \vdash a : B}$$

Table 3.3: Type assignment rules for $\mathbf{Ob}_{1<:}$

We can now use the type system $\mathbf{Ob}_{1<:}$ to give a more precise explanation of the constraints on typing outlined in Section 2.3. The subtype rule (Sub Obj) requires that for an object type A to be a subtype of B, the types of common methods must be the same, called *invariance*. This requirement is necessary to ensure soundness of the type system.

Example 3.2 To see the importance of the invariance constraint, let us try the following rule:

$$\text{(Sub Obj CoVar)}\quad \frac{\forall i \in J \quad \vdash A_i <: B_i \quad J \subseteq I}{\vdash [\mathrm{l}_i{:}A_i\ ^{i\in I}] <: [\mathrm{l}_i{:}B_i\ ^{i\in J}]}$$

where we allow subtyping of common methods. This rule is *covariant* in the method types. If we assuming (Sub Obj CoVar) to be sound, we can derive a contradiction. Let $B_1 = [\mathrm{l}_1{:}A_1,\ \mathrm{l}_2{:}A_2]$ and $B_2 = [\mathrm{l}_1{:}A_1]$, by (Sub Obj) we have $B_1 <: B_2$. We now use these two types to construct the types $C_1 = [\mathrm{l}{:}B_1,\ \mathrm{l}'{:}A_2]$ and $C_2 = [\mathrm{l}{:}B_2,\ \mathrm{l}'{:}A_2]$. By rule (Sub Obj CoVar) (but *not* by (Sub Obj)) we have $C_1 <: C_2$. Let b_1 have type B_1; then the object $c = [\mathrm{l}{=}\varsigma(x{:}C_1)b_1,\ \mathrm{l}'{=}\varsigma(x{:}C_1)x.\mathrm{l}.\mathrm{l}_2]$ has type C_1. By subsumption c also has type C_2, so the expression $(c.\mathrm{l}{\Leftarrow}\varsigma(x{:}C_2)b_2).\mathrm{l}'$ is well-typed, if b_2 has type B_2.

But this could lead to a run-time error since b_2 might not contain the l_2 method. □

This rule for subtyping is sound in the typed λ-calculus, so the example indicates why the use of typed λ-calculus to give semantics to typed object-oriented languages has turned out to be difficult.

We could also try a rule for subtyping of objects where we allowed the type of methods to be supertypes, called *contravariant* subtyping. But here it is even easier to create an example showing the rule to be unsound.

Example 3.3 Assuming contravariant subtyping, the type C_2 (from the previous example) would be a subtype of C_1. Now, given an object c of type C_2, we use subsumption to give c the type C_1, which leads us to believe that $c.\mathrm{l}$ returns a result of type B_1 on which we can activate the l_2 method. But the l_2 method might not be there since $c.\mathrm{l}$ might return a result of type B_2. □

For covariant subtyping the problem is the interplay between subsumption and method update, and for contravariant subtyping the problem is subsumption and method activation. A solution is to use variance annotations (c.f. [AC96, Section 8.7]) to get covariant subtyping by forbidding override of a method and contravariant typing by forbidding activation of a method.

The encoding of the λ-calculus into the ς-calculus gives rise to a type discipline with subtyping for λ-calculus terms. But unfortunately this is not the same as the type discipline normally used for the λ-calculus with subtyping. In the λ-calculus with subtyping, we have the following rule for subtyping of function types:

$$\frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'}$$

But because of the invariance of the method types, the translation into the ς-calculus with the type system $\mathbf{Ob}_{1<:}$, blocks subtyping of functions. One solution to this would be to take functions as primitives in the ς-calculus adding type rules to deal with them. Abadi and Cardelli do this in the type system $\mathbf{FOb}_{1<:}$ [AC96, Chapter 8]. Another solution is to use variance annotations in the translation to recover the subtype relation between functions [AC96, Section 8.7-2].

The type systems for the ς-calculus only allow us to update existing methods. One might think it would be easy to incorporate addition of methods to objects into the type system $\mathbf{Ob}_{1<:}$, but as shown by Fisher [Fis96] this

is not without complications. If we assumed the following rule for typing addition of methods:

$$\frac{\Gamma \vdash a{:}[\mathrm{l}_i{:}B_i\ ^{i\in I}] \quad \Gamma[x{:}A] \vdash b{:}B \quad k \notin I \quad A = [\mathrm{l}_i{:}B_i, \mathrm{l}_k{:}B\ ^{i\in I}]}{\Gamma \vdash a.\mathrm{l}_k \Leftarrow \varsigma(x{:}A)b : A} \text{ (Method Add)}$$

We would discover that this would lead to an unsound type system. Because we might have used subsumption to give a the type $[\mathrm{l}_i{:}B_i\ ^{i\in I}]$ and one of the methods visible in a might rely on a method with the label l_k hidden by subsumption. So in order to handle method addition soundly, it requires that the use of subsumption is limited.

3.3.3 Adding recursion

The type system $\mathbf{Ob}_{1<:}$ is not very expressive. For instance, it is difficult to find useful types for objects which contain methods that return self or a modified self. To see the weakness of $\mathbf{Ob}_{1<:}$ consider the following example:

Example 3.4 Let a denote the object $[\mathrm{l}{=}\varsigma(x{:}A)x]$ that on activation of the method l returns self. What type can we assign to this object in $\mathbf{Ob}_{1<:}$? Since a contains only one method, the type of a must be on the form $[\mathrm{l}{:}B]$ for some unknown B. To see what B can be, consider the following derivation:

$$\frac{\dfrac{\vdash [\mathrm{l}{:}B] <: B}{x{:}[\mathrm{l}{:}B] \vdash x{:}B}}{\emptyset \vdash [\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}B])x] : [\mathrm{l}{:}B]}$$

From this we can conclude that B must be a subtype of $[\mathrm{l}{:}B]$. This can only be true if B is either the empty object type $[\,]$ or the type $[\mathrm{l}{:}B]$. The second type it cannot be, because this would lead to an infinite type expression. Therefore the type of a must be $[\mathrm{l}{:}[\,]]$, implying that the result of an activation of l on a is the empty object (although the actual result is the object a). □

The type we *would* like to give a is in fact the "infinite type":

$$[\mathrm{l}{:}[\mathrm{l}{:}[\mathrm{l}{:}[\mathrm{l}{:}[\ldots]]]]],$$

or, in other words, a solution to the type equation $A = [\mathrm{l}{:}A]$ To this end, we create an extended type system $\mathbf{Ob}_{1<:\mu}$ by adding recursive types to $\mathbf{Ob}_{1<:}$. [AC96, Chapter 9] describes the type system $\mathbf{Ob}_{1<:\mu}$ for the ς-calculus that we have shown so far. In the Chapters 7 and 8, we work with a slight

extension of the ς-calculus due to Gordon and Rees [GR96]. So instead of introducing $\mathbf{Ob}_{1<:\mu}$ for the version of the ς-calculus we have already seen and then later extend the calculus and the type system, we shall already now give the extended calculus and introduce the recursive type system for that calculus.

The extension made by Gordon and Rees in [GR96] to the ς-calculus is the addition of boolean values and a conditional construct to the ς-calculus. The set of object terms is defined by the abstract syntax in Table 3.4.

a	::=	$[\mathrm{l}_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in I}]$	objects
	$\mid$	x	self variables
	$\mid$	$a.\mathrm{l}$	method activation
	$\mid$	$a.\mathrm{l}{\Leftarrow}\varsigma(x{:}A)b$	method override
	$\mid$	$\mathsf{fold}(A,a) \mid \mathsf{unfold}(a)$	recursive fold/unfold
	$\mid$	$\mathsf{if}(a,b_1,b_2)$	
	$\mid$	$\mathsf{true} \mid \mathsf{false}$	booleans

Table 3.4: The ς-calculus extended with booleans and recursive types.

A value, denoted by v, is either an object ($[\mathrm{l}_i{=}\varsigma(x_i{:}B_i)b_i\ ^{i\in I}]$), a boolean value ($\mathsf{true}, \mathsf{false}$) or a folded value ($\mathsf{fold}(A,v)$). The fold and unfold operations are used to explicitly handle the isomorphism between a recursive type and its unfolding.

The set of reduction rules is extended to include the reductions found in Table 3.5.

$$\mathsf{if}(\mathsf{true},b_1,b_2) \rightsquigarrow b_1 \qquad \mathsf{if}(\mathsf{false},b_1,b_2) \rightsquigarrow b_2$$
$$\mathsf{unfold}(\mathsf{fold}(A,v)) \rightsquigarrow v$$

Table 3.5: Additional reduction rules for the extended ς-calculus.

We also need to extend the set of reduction contexts to:

$$C[\cdot] ::= C[\cdot].\mathrm{l} \mid C[\cdot].\mathrm{l}{\Leftarrow}\varsigma(x)b \mid \mathsf{if}(C[\cdot],b_1,b_2) \mid \mathsf{unfold}(C[\cdot]) \mid \mathsf{fold}(A,C[\cdot]) \mid [\cdot]$$

The set of $\mathbf{Ob}_{1<:\mu}$ type expressions is defined via the following abstract syntax:

$$A ::= \mathsf{Bool} \mid [\mathrm{l}_i{:}A_i\ ^{i\in I}] \mid \mathsf{Top} \mid \mu(X)A \mid X$$

Here Bool denotes the only ground type, namely that of truth values. The type $[\mathrm{l}_i{:}A_i\ ^{i\in I}]$ as usual denotes an object record type, where the method l_i has type A_i. Top denotes the most general or unspecified type, $\mu(X)A$ is a recursive type and X ranges over **TypeVar**, the set of type variables.

In order to ensure uniqueness of recursively defined types, Abadi and Cardelli define a syntactic predicate of *formal contractivity* on type variables. $A \succ Y$ should be read as 'variable Y is formally contractive in type expression A'. Informally, this means that any occurrence of Y occurs within the scope of a method label in the type expression A. The rules defining the predicate are shown in Table 3.6. We let **Types** denote the set of type expressions that meets this criteria and shall, when assigning types to expressions, only use types from **Types**.

$$\frac{X \neq Y}{X \succ Y} \qquad \frac{}{\mathsf{Top} \succ Y} \qquad \frac{}{[\mathrm{l}_i{:}A_i\ ^{i\in I}] \succ Y} \qquad \frac{A \succ Y}{\mu(X)A \succ Y}$$

Table 3.6: Formal contractivity

The rules for typing expressions are the ones for $\mathbf{Ob}_{1<:}$ from Table 3.3 plus the new ones found in Table 3.7. As mentioned earlier, the operations fold and unfold are used to handle the isomorphism between a type and its unfolding in an explicit manner (c.f. [AC96, Chapter 9] for details). The rule (Unfold) states that if an expression a has the recursive type $\mu(X)B$ then $\mathsf{unfold}(a)$ will have the unfolded type $B\{{}^{\mu(X)B}/_{X}\}$.

Subtyping in $\mathbf{Ob}_{1<:\mu}$ is somewhat more complicated than in $\mathbf{Ob}_{1<:}$ since we now have to deal with recursive types. Subtyping now requires an environment Γ that describes constraints on type variables, where $\Gamma(X) = A$ states that we assume $X <: A$. A subtype judgment is consequently written as $\Gamma \vdash A <: B$, and states that the type A is a subtype of B, given the subtyping assumptions in Γ. The subtyping relation is defined by the inference rules of Table 3.8 plus the rule (Sub Obj) on page 32.

Example 3.5 We can now use $\mathbf{Ob}_{1<:\mu}$ to give the expression $[\mathrm{l}{=}\varsigma(x{:}A)x]$ the type $A = \mu(X)[\mathrm{l}{:}X]$ (with an occurrence of an operation fold to deal with the recursive type) using the following derivation:

$$\frac{\dfrac{x{:}A \vdash x{:}A}{\emptyset \vdash [\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}A])x]{:}[\mathrm{l}{:}A]}}{\emptyset \vdash \mathsf{fold}(A, [\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}A])x]){:}A}$$

(Fold)
$$\frac{\Gamma \vdash a{:}B\{{}^{A}\!/_{X}\} \quad A = \mu(X)B}{\Gamma \vdash \mathsf{fold}(A, a) : A}$$

(Unfold)
$$\frac{\Gamma \vdash a{:}A \quad A = \mu(X)B}{\Gamma \vdash \mathsf{unfold}(a) : B\{{}^{A}\!/_{X}\}}$$

(If)
$$\frac{\Gamma \vdash b{:}\mathsf{Bool} \quad \Gamma \vdash a_1, a_2 : A}{\Gamma \vdash \mathsf{if}(b, a_1, a_2) : A}$$

(Bool)
$$\frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b{:}\mathsf{Bool}}$$

(Subsump)
$$\frac{\Gamma \vdash a{:}A_1 \quad \Gamma \vdash A_1 <: A_2}{\Gamma \vdash a : A_2}$$

Table 3.7: Type assignment rules for $\mathbf{Ob}_{1<:\mu}$

(Sub Refl)
$$\frac{-}{\Gamma \vdash A <: A}$$

(Sub Trans)
$$\frac{\Gamma \vdash A_1 <: A_2 \quad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_1 <: A_3}$$

(Sub X)
$$\frac{\Gamma(X) = A}{\Gamma \vdash X <: A}$$

(Sub Top)
$$\frac{-}{\Gamma \vdash A <: \mathsf{Top}}$$

(Sub Rec)
$$\frac{\Gamma[X_2 <: \mathsf{Top}, X_1 <: X_2] \vdash A_1 <: A_2}{\Gamma \vdash \mu(X_1)A_1 <: \mu(X_2)A_2}$$

Table 3.8: The subtyping relation for $\mathbf{Ob}_{1<:\mu}$

□

Another property of the type system $\mathbf{Ob}_{1<:}$ and $\mathbf{Ob}_{1<:\mu}$ is that for any type A, there is a divergent object expression Ω_A definable as:

$$\Omega_A \triangleq [\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}A])x.\mathrm{l}].\mathrm{l}.$$

In some sense, this property relies on the fact that a divergent object never evaluates to a value. If an object never reduces to a value, we never get in a position, where we actually try to activate or update a method.

Although more expressive then $\mathbf{Ob}_{1<:}$, the type system $\mathbf{Ob}_{1<:\mu}$ is not as expressive as we would like it to be, as illustrated by considering an adaptation of the *Point*/*CPoint* problem from Section 2.4 to the ς-calculus (this example is taken from [AC96, Section 9.5]).

Example 3.6 Assuming the addition of functions to the ς-calculus we can build a one-dimensional *Point* object (modulo some fold/unfold operations) as:

$$[\mathrm{xp}{=}0,\ \mathrm{move}_x{=}\varsigma(s{:}Pt)\lambda x.(s.\mathrm{xp} := x)]$$

having type

$$Pt = \mu(X_1)[\mathrm{xp}{:}Real,\ \mathrm{move}_x{:}Real{\to}X_1]$$

In a similar manner we could create a one-dimensional *CPoint* having type

$$CPt = \mu(X_2)[\mathrm{xp}{:}Real,\ \mathrm{color}{:}Integer,\ \mathrm{move}_x{:}Real{\to}X_2]$$

We would like the relation $CPt <: Pt$ to hold. But if we try to derive this in $\mathbf{Ob}_{1<:\mu}$ we get stuck, when trying to show subtyping between the object types, as shown be the partial inference tree:

$$\frac{X_1 <: \mathsf{Top}, X_2 <: X_1 \vdash [\mathrm{xp}{:}Real,\ \mathrm{color}{:}Integer,\ \mathrm{move}_x{:}Real{\to}X_2] <: [\mathrm{xp}{:}Real,\ \mathrm{move}_x{:}Real{\to}X_1]}{\emptyset \vdash \mu(X_2)[\mathrm{xp}{:}Real,\ \mathrm{color}{:}Integer,\ \mathrm{move}_x{:}Real{\to}X_2] <: \mu(X_1)[\mathrm{xp}{:}Real,\ \mathrm{move}_x{:}Real{\to}X_1]}$$

Where the type $Real{\to}X_2$ should be equal to the type $Real{\to}X_1$ in order for the inference to complete.

By an example similar to the one from Example 3.2, we can see that *CPt* in fact cannot be a subtype of *Pt* if $\mathbf{Ob}_{1<:\mu}$ is to sound. □

The problem outlined in the above example can be solved by a more advanced type system as shown in [AC96]. Since we in this thesis only use the type systems presented so far, we refer the interested reader to [AC96] for a detailed description of more advanced type systems for the ς-calculus.

3.4 Equational theories

Abadi and Cardelli do not only study type systems for the ς-calculus; they also give typed equational theories for reasoning about ς-calculus expressions in a purely syntactic way. In this section, we present the equational theory for $\mathbf{Ob}_{1<:\mu}$.

All judgments are of the form $\Gamma \vdash a \leftrightarrow b : A$, where Γ is a type environment mapping self variables to types, a and b are objects, and A is a type. The intended interpretation of this judgment is that the expressions a and b are considered equal as objects of type A, given the assumptions in Γ about the free variables in a and b. That the equational theory is a typed theory (we equate objects under an assumption about their type), also stresses the importance of types when one wants to reason about the behavior of object-oriented programs.

The rules in Table 3.9 establish symmetry and transitivity, plus a limited form of reflexivity on variables; a general rule for reflexivity is not needed, as it follows as a derived rule. Table 3.10 collects congruence rules for

(Eq Symm)	(Eq Trans)	(Eq x)
$\dfrac{\Gamma \vdash a \leftrightarrow b : A}{\Gamma \vdash b \leftrightarrow a : A}$	$\dfrac{\Gamma \vdash a \leftrightarrow b : A, \quad b \leftrightarrow c : A}{\Gamma \vdash a \leftrightarrow c : A}$	$\dfrac{\Gamma(x) = A}{\Gamma \vdash x \leftrightarrow x : A}$

Table 3.9: Equivalence-inducing equational rules

objects and rules corresponding to the clauses of the reduction semantics. Finally, we have in Table 3.11 the rules for subtyping. The most interesting rule is (Eq Sub Object), defined in Table 3.11 which allows one to prove equalities between objects with different collections of methods. To prove two objects of different length equal we need to prove their common methods equal assuming the "shorter" object type; for the rest of the methods we are allowed to use the "longer" object type.

The equational theory allows us to prove many interesting equalities between objects. Consider the following example (from [AC96]), to see what we can prove in the equational theory, its limitations and why it is important to consider types when reasoning about objects.

Example 3.7 Consider the following two objects.

$$a = [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2{=}\mathsf{true}] \qquad b = [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2{=}\varsigma(x{:}[\mathrm{l}_1{:}\mathsf{Bool},\ \mathrm{l}_2{:}\mathsf{Bool}])x.\mathrm{l}_1]$$

(Eq Object) where $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$

$$\frac{\forall i \in I \quad \Gamma, x_i{:}A \vdash b_i \leftrightarrow b'_i : B_i}{\Gamma \vdash [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I}] \leftrightarrow [\mathrm{l}_i{=}\varsigma(x_i{:}A)b'_i\ ^{i\in I}] : A}$$

(Eq Select)

$$\frac{\Gamma \vdash a \leftrightarrow b : [\mathrm{l}_i{:}B_i\ ^{i\in I}] \quad j \in I}{\Gamma \vdash a.\mathrm{l}_j \leftrightarrow b.\mathrm{l}_j : B_j}$$

(Eq Override) where $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$

$$\frac{\Gamma \vdash a \leftrightarrow a' : A \quad \Gamma, x{:}A \vdash b \leftrightarrow b' : B_j \quad j \in I}{\Gamma \vdash a.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)b \leftrightarrow a'.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)b' : A}$$

(Eq If)

$$\frac{\Gamma \vdash b \leftrightarrow b' : \mathsf{Bool} \quad \Gamma \vdash a_1 \leftrightarrow a'_1 : A,\ a_2 \leftrightarrow a'_2 : A}{\Gamma \vdash \mathsf{if}(b, a_1, a_2) \leftrightarrow \mathsf{if}(b', a'_1, a'_2) : A}$$

(Eq Fold) where $A = \mu(X)B$

$$\frac{\Gamma \vdash a \leftrightarrow b : B\{{}^{A}\!/_{X}\}}{\Gamma \vdash \mathsf{fold}(A, a) \leftrightarrow \mathsf{fold}(A, b) : A}$$

(Eq Unfold) where $A = \mu(X)B$

$$\frac{\Gamma \vdash a \leftrightarrow b : A}{\Gamma \vdash \mathsf{unfold}(a) \leftrightarrow \mathsf{unfold}(b) : B\{{}^{A}\!/_{X}\}}$$

(Eval Select)

$$\frac{\Gamma \vdash a{:}A \quad j \in I}{\Gamma \vdash a.\mathrm{l}_j \leftrightarrow b_j\{{}^{a}\!/_{x_j}\} : B_j} \quad \text{where} \quad \begin{array}{l} A = [\mathrm{l}_i{:}B_i\ ^{i\in I}] \\ a = [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I\cup J}] \end{array}$$

(Eval Override) where $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$, $a = [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I\cup J}]$

$$\frac{\Gamma \vdash a{:}A \quad \Gamma, x{:}A \vdash b{:}B_j \quad j \in I}{\Gamma \vdash a.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)b \leftrightarrow [\mathrm{l}_i{=}\varsigma(x_i{:}A')b_i,\ \mathrm{l}_j{=}\varsigma(x{:}A)b\ ^{i\in I\cup J\setminus\{j\}}] : A}$$

(Eval Fold) where $A = \mu(X)B$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{fold}(A, \mathsf{unfold}(a)) \leftrightarrow a : A}$$

(Eval Unfold) where $A = \mu(X)B$

$$\frac{\Gamma \vdash a : B\{{}^{A}\!/_{X}\}}{\Gamma \vdash \mathsf{unfold}(\mathsf{fold}(A, a)) \leftrightarrow a : B\{{}^{A}\!/_{X}\}}$$

(Eval If$_1$)

$$\frac{\Gamma \vdash b_1, b_2 : B}{\Gamma \vdash \mathsf{if}(\mathsf{true}, b_1, b_2) \leftrightarrow b_1 : B}$$

(Eval If$_2$)

$$\frac{\Gamma \vdash b_1, b_2 : B}{\Gamma \vdash \mathsf{if}(\mathsf{false}, b_1, b_2) \leftrightarrow b_2 : B}$$

Table 3.10: Equational rules specific to the calculus

$$\frac{\Gamma \vdash a \leftrightarrow b : A \quad \Gamma \vdash A \lessdot B}{\Gamma \vdash a \leftrightarrow b : B} \text{ (Eq Subsump)} \qquad \frac{\Gamma \vdash a{:}A,\ b{:}B}{\Gamma \vdash a \leftrightarrow b : \mathsf{Top}} \text{ (Eq Top)}$$

$$\frac{\forall i \in I \quad \Gamma, x_i{:}A \vdash b_i{:}B_i \quad \forall j \in J \setminus I \quad \Gamma, x_j{:}A' \vdash b_j{:}B_j}{\Gamma \vdash [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in I}] \leftrightarrow [\mathrm{l}_i{=}\varsigma(x_i{:}A')b_i\ ^{i\in J}] : A}$$

(Eq Sub Object) where $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$, $A' = [\mathrm{l}_i{:}B_i\ ^{i\in J}]$, $I \subseteq J$

Table 3.11: Equational rules for subtyping

If we consider a and b to be objects of type $A = [\mathrm{l}_1{:}\mathsf{Bool},\ \mathrm{l}_2{:}\mathsf{Bool}]$, we do not want them to be equal because we can override the l_1 method with for instance false, and obtain two different results from the activation of l_2.

Now consider the following two types $B_1 = [\mathrm{l}_1{:}\mathsf{Bool}]$ and $B_2 = [\mathrm{l}_2{:}\mathsf{Bool}]$; both are supertypes of A.

We can use the equational theory to prove the equality $\emptyset \vdash a \leftrightarrow b : B_1$ (using (Eq Sub Object)). This equality also seems reasonable, since hiding the l_2 method prevents us from detecting any difference between a and b.

Now, what about the type B_2? Intuitively we want a to be equal to b at the type B_2, for on activation of the only available method l_2 we get the same result from a and b. And if we override the l_2 method with the same method on a and b, a and b become identical objects. But this equality is not derivable in the equational theory! However, it can be shown by using bisimulation defined on top of a labelled transition system for the ς-calculus [GR96]. In Chapter 7 we shall consider the labelled transition system of Gordon and Rees and compare it with the denotational foundation on which Abadi and Cardelli build their equational theory. □

Chapter 4

The π-calculus

In the Introduction, we mentioned the use of process calculi when giving semantics to programming languages. Most of the languages that have been described have either been λ-calculi and/or have incorporated some notion of concurrency. One of the main motivations for using process calculi to give semantics to concurrent languages has been the built-in notion of concurrency. In [Mil89], Milner shows how to use CCS to give semantics to a concurrent imperative programming language with shared variables, but CCS' problems with expressing dynamic change of structure makes it quite cumbersome to express semantics of programming languages where references are an integral part (such as most object-oriented languages). The π-calculus extends CCS with the ability to pass channels (or names, as they are also called in the π-calculus) over channels. Calculi based on this kind of communication are called mobile calculi. Mobile calculi allow a natural way of expressing references, and for this reason they have become prevalent when giving process calculus semantics to programming languages.

In recent years, the theory of the π-calculus has been studied extensively. Two topics are particularly interesting when using the π-calculus to give semantics to programming languages, namely the study of subcalculi, such as the asynchronous π-calculus [ACS98], and of type disciplines for the π-calculus [Mil93, PS96, VH93].

Subcalculi The full π-calculus is a very expressive calculus. One way to study its expressiveness is by examining what impact constraints of the π-calculus have on its expressiveness. This study takes several forms. One way is to give encodings of a calculus into a subcalculus or to show the non-existence

of such an encoding (e.g. [Pal97]). Another way is to study equalities between processes. A subcalculus usually allows one to equate processes that were not equal in the supercalculus, because the set of contexts available in a subcalculus to discriminate between processes is smaller than the similar set in the supercalculus (e.g. [MS98]).

From the point of view of using the π-calculus to give semantics to programming languages, it is interesting to note that we rarely need the full π-calculus to give semantics to programming languages. Using a subcalculus makes reasoning about a translation of a programming languages into the π-calculus easier, because a subcalculus has less discriminating power than the full calculus.

Type systems In the polyadic π-calculus, where it is possible to pass several names at a time on a channel, the usage of names gives rise to a simple typing discipline, originally called *sorting*, ensuring that receivers and transmitters agree on the number of names that are passed (c.f. [Mil93]). This type system has later been extended in several ways with for instance subtyping [PS96] and polymorphism [Tur96].

Types in the π-calculus have proven quite useful when studying translations. The most well-known example is Milner's optimized encoding of the untyped call-by-value λ-calculus, that turned out to be wrong in the untyped π-calculus but sound in a typed version of the π-calculus [PS96] (because the type system defines a subcalculus).

In this chapter, we introduce the basic polyadic π-calculus and its theory. Readers already familiar with the π-calculus can skip this chapter and proceed to Chapter 5 where we start our study of the ς-calculus by using the π-calculus to give semantics to the untyped ς-calculus.

4.1 The polyadic π-calculus

We have chosen to start out by presenting the full polyadic π-calculus and then introduce interesting subcalculi later, as this makes the restrictions easier to understand.

4.1.1 Syntax

Process terms in the polyadic π-calculus is given by the following syntax:

$$
\begin{array}{lcll}
P, Q, \ldots & ::= & \sum_{i \in I} \alpha_i.P_i & \text{Guarded sum} \\
& | & P|Q & \text{Parallel composition} \\
& | & (\nu \tilde{a})P & \text{Restriction} \\
& | & !P & \text{Replication} \\
& | & [a{=}b]P & \text{Matching} \\
& | & [a{\neq}b]P & \text{Mismatching} \\
 & & & \\
\alpha & ::= & \overline{a}\langle \tilde{b} \rangle & \text{Output prefix} \\
& | & a(\tilde{y}) & \text{Input prefix} \\
& | & \tau & \text{Silent prefix}
\end{array}
$$

Here we let $a, b, \ldots, x, y, \ldots \in \mathbf{Names}$ range over an countable infinite set of names, $\tilde{b}$ denotes a tuple $b_1, b_2 \ldots b_n$ of names. $P \in \mathbf{Proc}$ will range over processes or agents.

A *guarded choice*, $\sum_{i \in I} \alpha_i.P_i$, where I is a finite index set, is a process that can perform *one* of its possible actions, preempting the others. A prefix can be an output, input or silent prefix. *Output prefixing*, $\overline{a}\langle \tilde{b} \rangle$, expresses a process' wish to transmit the names $\tilde{b}$ over the name a. We write output of the empty tuple $\langle\rangle$ on the name a, $\overline{a}\langle\rangle$, as $\overline{a}$ and similarly a for $a()$. *Input prefixing*, $a(\tilde{y})$, gives the possibility of receiving over the channel a a tuple of names, with $\tilde{y}$ denoting the formal parameters. Finally, the *silent prefix*, τ, can be derived from the other two using communication, but we have chosen to include it for simplicity. A τ prefix simply denotes an internal action that can be performed without participation of other processes. In case $I = \emptyset$, we write $\sum_{\emptyset} \alpha_i.P_i$ as $\mathbf{0}$, denoting the inactive process. We usually omit the trailing $\mathbf{0}$ from processes, writing $\overline{a}\langle b \rangle.\mathbf{0}$ as $\overline{a}\langle b \rangle$. We shall also feel free to use $+$ for binary choice.

Parallel composition, $P|Q$, denotes two processes that run in parallel. P and Q run independently, but can also communicate over shared names.

Restriction, $(\nu \tilde{a})P$, is one of the most difficult operators to understand. Restriction is responsible for *scoping* in the π-calculus. For instance, in the expression $\overline{a}\langle b \rangle \mid (\nu a)(\overline{a}\langle c \rangle \mid a(y).P)$ the a outside the restriction is different from the a's inside the restriction; we say that the restriction makes the name a *private*. Therefore we can also safely rename a inside the restriction to some other name not occurring within the restriction.

Replication, $!P$, can be thought of as denoting an unbounded number of copies of the process P in parallel.

The last two operators, *match* and *mismatch*, allow us to test for equality of names. Matching allows us to continue if the names are equal and mismatching if they are unequal. In most presentations of the π-calculus, match

and mismatch are left out, but as we shall see later some sort of matching makes it easier to encode the ς-calculus.

There are two binding constructs in the π-calculus, namely restriction $(\nu\tilde{a})P$ and input prefixing $a(\tilde{b}).Q$. The former binds the names $\tilde{a}$ in P, the latter binds the names $\tilde{b}$ in Q. In what follows, we let $\mathsf{bn}(P)$, $\mathsf{fn}(P)$, and $\mathsf{n}(P)$, respectively, denote the set of bound names, free names, and names of the agent P. We identify terms up to alpha conversion of bound names.

4.1.2 Operational semantics

The semantics of the π-calculus is given as a labelled transition system $(\mathbf{Proc}, \rightarrow)$ where $\rightarrow \subseteq (\mathbf{Proc} \times \mathbf{Label} \times \mathbf{Proc})$ denotes the transition relation with **Label** denoting the set of labels. We write $P \xrightarrow{\mu} Q$ if $(P, \mu, Q) \in \rightarrow$, saying that the process P can do the action μ and become the process Q.

Labels are given by the following grammar:

$$\mu ::= (\nu\tilde{z})\overline{x}\langle\tilde{y}\rangle \mid x\tilde{y} \mid \tau$$

The label $(\nu\tilde{z})\overline{x}\langle\tilde{y}\rangle$ denotes the output of the names $\tilde{y}$ on the name x. The restriction $(\nu\tilde{z})$, where $\tilde{z}$ is a subset of the names in $\tilde{y}$, indicate that the names $\tilde{z}$ are bound (restricted) names; if there are no names bound we omit the restriction on the label. The label $x\tilde{y}$ denotes input of the names $\tilde{y}$ over the name x. Finally, τ denotes an internal action. The transition relation $\rightarrow$ is the smallest relation closed under the inference rules in Table 4.1 (with the symmetric versions of (Sync-L) and (Comp-L) omitted).

The most difficult rules to comprehend are probably (Open), (Comp-L) and (Sync-L) with their side conditions for handling scoping of restricted names. (Open) states that if we transmit a restricted name over its restriction, then we must record this in the label so that we can later (when using (Sync-L)) put the appropriate restriction back in place. (Comp-L) must ensure that the restricted names we carry over a parallel composition do not by accident capture free names when we use the (Sync-L) rule. For instance, consider the expression

$$a(y).P \mid (Q \mid (\nu b)(\overline{a}\langle b\rangle.R))$$

Here we want to transmit the private name b to the process $a(y).P$. In order to do this we need to ensure that we do not by accident capture free occurrences of b in the processes P and Q. So if b occurs free in P or Q, we need to rename the bound name b to some name that does not occur free in P or Q in order for us to infer the transmission.

(Inp)
$$\frac{|\tilde{y}| = |\tilde{z}|}{a(\tilde{y}).P \xrightarrow{a\tilde{z}} P\{\tilde{z}/\tilde{y}\}}$$

(Act)
$$\frac{\alpha \in \{\overline{a}\langle\tilde{x}\rangle,\ \tau\}}{\alpha.P \xrightarrow{\alpha} P}$$

(Sum)
$$\frac{\alpha_j.P_j \xrightarrow{\mu} P'\ j \in I}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{\mu} P'}$$

(Res) where $\tilde{a} \cap \mathsf{n}(\mu) = \emptyset$
$$\frac{P \xrightarrow{\mu} P'}{(\nu\tilde{a})P \xrightarrow{\mu} (\nu\tilde{a})P'}$$

(Open) where $a \notin \tilde{x},\ \tilde{x} \cap \tilde{z} = \emptyset$
$\tilde{x}' = \tilde{x} \setminus \mathsf{n}(\tilde{b}),\ \tilde{z}' = \tilde{z}(\tilde{x} \cap \mathsf{n}(\tilde{b}))$
$$\frac{P \xrightarrow{(\nu\tilde{z})\overline{a}\langle\tilde{b}\rangle} P'}{(\nu\tilde{x})P \xrightarrow{(\nu\tilde{z}')\overline{a}\langle\tilde{b}\rangle} (\nu\tilde{x}')P'}$$

(Comp-L) where $\mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset$
$$\frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q}$$

(Sync-L) where $\tilde{z} \cap \mathsf{fn}(Q) = \emptyset$
$$\frac{P \xrightarrow{(\nu\tilde{z})\overline{a}\langle\tilde{b}\rangle} P'\ \ Q \xrightarrow{a\tilde{b}} Q'}{P|Q \xrightarrow{\tau} (\nu\tilde{z})(P'|Q')}$$

(Rep)
$$\frac{P|!P \xrightarrow{\mu} Q}{!P \xrightarrow{\mu} Q|!P}$$

(Match)
$$\frac{P \xrightarrow{\mu} P'}{[a{=}a]P \xrightarrow{\mu} P'}$$

(MisM) where $a \neq b$
$$\frac{P \xrightarrow{\mu} P'}{[a{\neq}b]P \xrightarrow{\mu} P'}$$

Table 4.1: The inference rules for the π-calculus.

The semantics given here is usually referred to as an *early semantics* because we instantiate the bound variables at the moment we infer an input action (in the rule (Inp)). Another possibility is to use a *late semantics* where we instantiate bound variables when we infer a communication [MPW92]. To do this, we would use the following two inference rules replacing (Inp) and (Sync-L) from Table 4.1:

(Inp_L)
$$\frac{-}{a(\tilde{y}).P \xrightarrow{a(\tilde{y})} P}$$

(Sync_L-L) where $\tilde{z} \cap \mathsf{fn}(Q) = \emptyset,\ |\tilde{b}| = |\tilde{y}|$
$$\frac{P \xrightarrow{(\nu\tilde{z})\overline{a}\langle\tilde{b}\rangle} P'\ \ Q \xrightarrow{a(\tilde{y})} Q'}{P|Q \xrightarrow{\tau} (\nu\tilde{z})(P'|Q'\{\tilde{b}/\tilde{y}\})}$$

The (bound) input label $a(\tilde{y})$ on a transition $P \xrightarrow{a(\tilde{y})} P'$ symbolizes that the process P is willing to receive, on the name a, a tuple of names that is bound to the names $\tilde{y}$ in P'.

4.1.3 Some notational conventions

We often need to consider transitions on the form $P \stackrel{\mu}{\Longrightarrow} P'$, where the τ-transitions have been abstracted away. We therefore define $\Rightarrow$ as $\xrightarrow{\tau}{}^*$ and the weak transition relation $\stackrel{\mu}{\Longrightarrow}$ for any label μ as $\Rightarrow \xrightarrow{\mu} \Rightarrow$, as usual. Furthermore, we define $\stackrel{\widehat{\mu}}{\Longrightarrow}$ as $\Rightarrow$, if $\mu = \tau$, and $\stackrel{\mu}{\Longrightarrow}$ otherwise.

If $P \xrightarrow{\mu} P'$ is the only transition P can perform (up-to alpha-conversion), we often write $P \xrightarrow{\mu}_d P'$. We write $P{\downarrow_a}$ if P can perform an input or output action on the name a ($P \xrightarrow{(\nu\tilde{z})\overline{a}\langle\tilde{b}\rangle}$ or $P \xrightarrow{a\tilde{b}}$) and $P{\Downarrow_a}$ if $P \Rightarrow P'{\downarrow_a}$. We use the notation $P{\Downarrow}$ ('P may converge') to indicate that P, after a sequence of τ-moves, can perform an observable action and $P{\Uparrow}$ ('P diverges') to indicate that P has an infinite sequence of τ-moves and $P \not\Downarrow$. The similarity to the notation defined for the ς-calculus in the previous chapter is entirely deliberate.

For reasons of readability we drop the restriction of names that no longer appear in a process. For instance, $(\nu o)(\overline{x}\langle y\rangle)$ will be written as $\overline{x}\langle y\rangle$ — this is valid because the two expressions are equated by all the equivalences we use.

As it can be seen from the syntax, the present version of the π-calculus does not contain recursive definitions. This is because they can be seen as a derived concept. The idea is to encode a recursive definition $A(\tilde{x}) \stackrel{\triangle}{=} P$ as a replicated process located on some trigger name a, and then replace all calls to the recursive definition with a message on a spawning of a new copy of the replicated process (c.f. [Mil92] for details).

Example 4.1 As an example of how to encode recursive definitions consider the simple definition $A(x) \stackrel{\triangle}{=} x(y).A\langle y\rangle$ which we use in some process P. The definition is translated to $!a(x).x(y).\overline{a}\langle y\rangle$ for some $a \notin \mathsf{n}(P)$ and a modified version of P, denoted $\widehat{P}$, is built by exchanging all occurrences of $A\langle z\rangle$ in P with $\overline{a}\langle z\rangle$. Finally we put the translation of A in parallel with $\widehat{P}$ and restrict a away, getting

$$(\nu a)(\widehat{P} \mid !a(x).x(y).\overline{a}\langle y\rangle)$$

□

4.2 Equivalences for the π-calculus

In this section, we give an overview of the two most well-known definitions of equivalence for the π-calculus, namely *bisimulation* and *barbed bisimulation*. We shall also briefly consider techniques for proving processes equivalent.

4.2.1 Bisimulation equivalence

The basic intuition behind equivalences defined as bisimulations is that of button-pushing experiments. In order for two processes P and Q to be bisimulation equivalent they must have the same set of observable transitions (buttons). If we look at the state P' resulting from choosing one of P's transitions (pushing a button on P), then Q must have a transition with the same label leading to a state Q' that is again bisimilar to P' and vice versa.

Definition 4.1 (Strong early bisimulation equivalence) *A symmetric relation $\mathcal{R}$ is a* strong bisimulation *if for all $P\ \mathcal{R}\ Q$:*

- *If $P \xrightarrow{\mu} P'$, then there exists a Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\ \mathcal{R}\ Q'$.*

If $P\ \mathcal{R}\ Q$ for some strong bisimulation $\mathcal{R}$, we say that P and Q are (strongly) bisimilar (written $P \dot{\sim} Q$).

As the name implies bisimulation equivalence is an equivalence relation, but unfortunately not a congruence relation for the operators of π-calculus. The problem is that $\dot{\sim}$ is not preserved by input prefix, as shown by the following example:

$$\overline{x}.y + y.\overline{x} \dot{\sim} \overline{x} \mid y \text{ but } z(y)(\overline{x}.y + y.\overline{x}) \not\dot{\sim} z(y).(\overline{x} \mid y)$$

On input of the name x, the latter process can perform a τ-transition, which the former cannot. In order to get the induced congruence, we need to close $\dot{\sim}$ under substitutions (that is, replacements of names for names).

Definition 4.2 (Strong congruence) *P and Q are* strongly congruent, *written $P \sim Q$, if $P\sigma \dot{\sim} Q\sigma$ for all substitutions σ.*

Strong bisimulation and congruence are sensitive to the number of internal moves by processes as two equivalent processes also must match on τ-transitions. Weak bisimulation ($\approx$) is defined by replacing the occurrences of $\xrightarrow{\mu}$ in the right-hand side of the above definition with the corresponding weak transition $\stackrel{\widehat{\mu}}{\Longrightarrow}$, thus allowing a process to match a μ-transition by a series of τ-transitions followed by the μ-transition and then again a series of τ-transitions (unless μ is a τ, in which case a process can also match the transition by doing no transitions at all).

Definition 4.3 (Weak bisimulation equivalence) *A symmetric relation $\mathcal{R}$ is a* weak bisimulation *if for all $P\ \mathcal{R}\ Q$:*

- *If $P \xrightarrow{\mu} P'$, then there exists a Q' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$.*

If $P\ \mathcal{R}\ Q$ for some weak bisimulation $\mathcal{R}$, we say that P and Q are weakly bisimilar (written $P \dot{\approx} Q$).

Weak bisimulation suffers from the same problem as strong bisimulation — it is not a congruence. And just as for the strong case we get the induced congruence, denoted $\approx$, by closing under all substitutions. Readers familiar with CCS may remember that weak bisimulation in CCS fails to be a congruence w.r.t. summation; this is not the case in the version of the π-calculus presented here, as we only allow guarded summation.

The definition of bisimulation that we have shown here might seem like the natural choice of bisimulation equivalence for the π-calculus. But, just as we have both an early and a late semantics, we can also give another formulation of bisimulation called *late bisimulation* that is more discriminating than the (early) definition of bisimulation we have given here (c.f. [Mil93]). Although strictly finer than early bisimulation, late bisimulation is not a congruence either, and we still need to close under substitutions to get the congruence. This leads to another formulation of bisimulation, namely *open bisimulation* of Sangiorgi [San96] where substitutions are an integral part of the definition of bisimulation.

4.2.2 Barbed equivalence

Another natural notion of equivalence is that of *barbed bisimulation* [MS92], defined as bisimulation on τ-transitions plus the extra condition that two processes must have the same observable actions.

Definition 4.4 (Barbed bisimulation) *A symmetric relation $\mathcal{R}$ is a* barbed bisimulation *if for all $P\ \mathcal{R}\ Q$, whenever:*

- *If* $P \xrightarrow{\tau} P'$, *then there exists a* Q' *s.t.* $Q \xrightarrow{\tau} Q'$.
- *If* $P{\downarrow_a}$, *then* $Q{\downarrow_a}$.

Two processes P *and* Q *are* barbed bisimilar, *written* $P \dot{\sim}_b Q$, *if there exists a barbed bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$.

The nice feature of barbed bisimulation is that it does not demand a large machinery in its definition — only the notion of reduction (τ-transitions in our case) and an observability predicate are needed. This makes barbed bisimulation easily adaptable to different kinds of calculi.

Looking at Definition 4.4 we would not expect barbed bisimulation to be a congruence, which is also not the case — barbed bisimulation is not even preserved by parallel composition. So again we need a definition of the induced congruence.

Definition 4.5 ($\mathcal{L}$-Barbed congruence) *Let* $\mathcal{L}$ *be a set of* π*-calculus contexts. We say that* P *and* Q *are* $\mathcal{L}$*-congruent, written* $P \sim_b^{\mathcal{L}} Q$, *if* $C[P] \dot{\sim}_b C[Q]$ *for all contexts* $C \in \mathcal{L}$.

The advantage of this definition of barbed congruence is that it is easy to put constraints on the set of contexts that we require two processes to be equivalent under. This we shall later use (in Section 6.2) to define what it means for two typed processes to be typed equivalent.

If we consider $\sim_b^{\mathcal{C}(\mathbf{Proc})}$, where $\mathcal{C}(\mathbf{Proc})$ is the set of all π-calculus contexts, then it has been shown by Sangiorgi [San93] to coincide with strong (early) bisimulation congruence for image-finite processes[1]. One way to look at strong bisimulation in light of this result is as proof technique for showing processes barbed congruent.

We can define weak barbed bisimulation and congruence, denoted $\dot{\approx}_b$ and $\approx_b^{\mathcal{L}}$ respectively, in a similar way as we did for bisimulation.

4.2.3 Proof techniques

Of course, one can use the definition of bisimulation directly to argue that two processes are equivalent by establishing a bisimulation relation containing the two processes and then argue that the relation is preserved under substitutions. But bisimulation relations can quickly become very large and difficult to deal with. To this end it is important to have some auxiliary techniques to establish the equivalence between processes. The two main techniques we use in this thesis are those of algebraic laws and up-to techniques.

[1] The set of image-finite processes is the largest set of processes $\mathbf{Proc}_f$ in $\mathbf{Proc}$, which is closed under transitions and for all μ and $P \in \mathbf{Proc}_f$, the set $\{P' : P \stackrel{\mu}{\Longrightarrow} P'\}$ is finite (up-to alpha-conversion).

Algebraic laws allow us to prove processes equivalent in a way similar to the way we prove algebraic equations in arithmetic — by replacing equals for equals in the processes and thereby modifying them so that they end up as syntactically equal processes. We shall not give a full account of the laws that we shall be using. Most of them are fairly straightforward and intuitively true. (In fact, these rules are so evident that, they in many presentations of the π-calculus, such as [Mil93], they are an integral part of the semantics.) For instance, we have laws $P|Q \sim Q|P$ and $P|(Q|R) \sim (P|Q)|R$ stating commutativity and associativity of parallel composition.

Some of the more interesting laws govern restriction and replication. The basic law for restriction, $(\nu x)(P|Q) \sim (\nu x)P|Q$, if $x \notin \mathsf{fn}(Q)$ allows us to move restrictions. This can be used together with laws like $(\nu x)(x(\tilde{y}).P) \sim \mathbf{0}$ to "garbage collect" unreachable components of a system.

For replicated processes we can safely create copies or merge identical replications (depending on how we read the law) by the laws $!P \sim !P|!P$.

Up-to techniques make it possible to show two processes P and Q equivalent by establishing a relation that contains the pair (P, Q) and does not need to be a bisimulation relation. The most well-known technique is that of (weak) bisimulation up-to bisimulation defined as:

Definition 4.6 (Bisimulation up-to bisimulation) *A symmetric relation $\mathcal{R}$ is a weak bisimulation up to bisimulation if for all $P\ \mathcal{R}\ Q$:*

- *If $P \xrightarrow{\mu} P'$ then there exist processes P'', Q' and Q'' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \dot{\sim} P''\ \mathcal{R}\ Q'' \dot{\sim} Q'$*

Since weak bisimulation up to bisimulation implies weak bisimulation [MS92], we can use this to show that two processes are weakly bisimilar. This technique can be extended in several ways (see [SM92]) to give some very powerful proof methods.

Another technique that shall later turn out to be quite useful is based on the following preorder.

Definition 4.7 (Expansion) *A relation $\mathcal{R}$ is an* expansion *if $P\ \mathcal{R}\ Q$ implies:*

i. If $P \xrightarrow{\mu} P'$, then there exists a Q' s.t. $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$.

ii. If $Q \xrightarrow{\mu} Q'$, then there exists a P' s.t. $P \xrightarrow{\widehat{\mu}} P'$ and $P'\ \mathcal{R}\ Q'$.

We say Q expands P*, written* $P \dot{\leq} Q$*, if* $P \mathcal{R} Q$ *for some expansion* $\mathcal{R}$*.*

As usual we need to close under substitutions to get the induced congruence which we denote $\leq$*.*

It is easy to see from the definition of expansion congruence implies weak congruence. The nice feature of expansion is that if $Q \leq P$, and if we have an upper bound on the number of τ-transitions that the term P can perform, then we know that Q will not have more τ-transitions. This property will turn out to be quite useful, when we want to do induction on the number of τ-transitions that a term can perform.

It is also possible to define a barbed version of expansion, which we shall denote $\leq_b^{\mathcal{L}}$.

Definition 4.8 (Barbed expansion) *A relation* $\mathcal{R}$ *is a* barbed expansion *if for all* $P \mathcal{R} Q$*, whenever:*

- *If* $P \xrightarrow{\tau} P'$*, then there exists a* Q' *s.t.* $Q \stackrel{\tau}{\Longrightarrow} Q'$*.*
- *If* $Q \xrightarrow{\tau} Q'$*, then there exists a* P' *s.t.* $P \xrightarrow{\widehat{\tau}} P'$*.*
- *If* $P{\downarrow_a}$*, then* $Q{\Downarrow_a}$*.*
- *If* $Q{\downarrow_a}$*, then* $P{\downarrow_a}$*.*

Let $\mathcal{L}$ *be a set of* π*-calculus contexts. We say that* Q *is a* $\mathcal{L}$*-barbed expansion of* P*, written* $P \leq_b^{\mathcal{L}} Q$*, if for all contexts* $C \in \mathcal{L}$*, there exist a barbed expansion* $\mathcal{R}$ *such that* $C[P] \mathcal{R} C[Q]$*.*

4.3 Typing in the π-calculus

With the ability to pass tuples of names around, we start encountering run-time errors, where one process tries to send a tuple of a different arity than expected by a receiver. This is of course quite unfortunate and something one should try to avoid. A natural way to do this is by using some kind of type system.

The most basic type system is the *sort discipline* due to Milner [Mil93], where one partitions the set of names into sets, each set is named and is associated with a string of set names, called the sort.

Example 4.2 As a simple example, we could have the recursive sorting

$$TwoPair = (TwoPair, Nil) \text{ and } Nil = ( )$$

stating that names from *TwoPair* always are used to carry a pair of names of which the first name is again from *TwoPair* and the second is a name that can only be used to synchronize. □

Type checking sorts is fairly straightforward: We must simply check that the usage of names conforms to the sorting. A process judgment $\Gamma \vdash P : ok$ states that, with respect to the assumptions on names in Γ, the process P is well-sorted. The most interesting rule is the following one for input:

$$\frac{\Gamma \vdash a{:}(T_1 \dots T_n) \quad \Gamma, x_1{:}T_1 \dots x_n{:}T_n \vdash P : ok}{\Gamma \vdash a(x_1 \dots x_n).P : ok}$$

The rule states that the arity of the input prefix $a(x_1 \dots x_n)$ must match the arity given by the assumption on a from Γ and that we must be able to prove that P is well-typed assuming that the use of names x_1 to x_n in P have sort T_1 to T_n.

This fairly simple type system can be extended in several ways. One immediate extension is that of polarity types, based on the observation that in most processes a name is used either for input or for output. Pierce and Sangiorgi have constructed a type system where one can distinguish between the ability to use a name for output, input or both [PS96]. This gives rise to a simple subtype relation, where the capability to do both read and write operations is a subtype of the read capability and of the write capability (see Section 6.2 for a more detailed account).

4.4 Some subcalculi

It is interesting to notice that for most applications of the π-calculus, one does not need the expressive power of the full π-calculus. Using subsets of the π-calculus usually makes reasoning easier, as more equalities hold between processes in subcalculi because of the lesser discriminating power by observers. Another reason why we might be interested in subcalculi is that (seen as a programming language) the π-calculus is much too difficult to implement (especially in a distributed setting). This is witnessed by recent proposals for programming languages based on the π-calculus such as Pict [PT99] and Join [FG96].

The asynchronous π-calculus The crucial difference between the synchronous and asynchronous π-calculus (π_a) lies in the use of the output construct. In the asynchronous π-calculus, output is non-blocking; this is

seen in the syntax as the absence of output prefixing ($\overline{a}\langle\tilde{b}\rangle.P$), instead we only have output atoms without continuations ($\overline{a}\langle\tilde{b}\rangle$), called messages. The asynchronous π-calculus was introduced by Honda and Tokoro [HT91] and has later been studied extensively. The presentation given here follows that of [ACS98].

The syntax for π_a is simply a subset of the syntax of the full π-calculus:

$$\begin{array}{rcll} P, Q, \ldots & ::= & \sum_{i\in I} \alpha_i.P_i & \text{Guarded choice} \\ & | & \overline{a}\langle\tilde{b}\rangle & \text{Output} \\ & | & P|Q & \text{Parallel composition} \\ & | & (\nu\tilde{a})P & \text{Restriction} \\ & | & !a(\tilde{y}).P & \text{Guarded replication} \\ & | & [a{=}b]P & \text{Matching} \\ & | & [a{\neq}b]P & \text{Mismatching} \\ \\ \alpha & ::= & a(\tilde{y}) \mid \tau & \end{array}$$

As it can be seen, we only allow choice on input and internal moves, which seems natural, as we want output to be non-blocking. If fact, we would also omit choice, as it can be faithfully encoded in the asynchronous π-calculus without choice, as shown by Nestmann and Pierce [NP96].

If we interpret $\overline{a}\langle\tilde{b}\rangle$ as $\overline{a}\langle\tilde{b}\rangle.\mathbf{0}$, we can simply use the semantics from Table 4.1 for π_a — but the syntax imposes additional constraints on possible transitions.

Just as in the synchronous π-calculus, several definitions of bisimulation equivalence exists (see [ACS98] for an overview). Barbed bisimulation can be used as a guideline for what the appropriate notion of bisimulation for π_a should be. It seems natural that the observability predicate should only take output actions into account since an asynchronous observer cannot tell if a message has been consumed. So we can define *asynchronous barbed bisimulation* as in Definition 4.4 but using the predicate $\downarrow_{\bar{a}}$ only checking for output actions. Similarly to the synchronous case, we can define *asynchronous barbed congruence* as asynchronous barbed bisimulation closed under all π_a contexts.

The definition of bisimulation from Definition 4.1 is too strong w.r.t. asynchronous barbed congruence, since it allows us to observe input transitions. The definition of bisimulation that seems to have the right power is the following definition due to Amadio, Castallani and Sangiorgi:

Definition 4.9 (Asynchronous bisimulation) *A symmetric relation $\mathcal{R}$ is an asynchronous bisimulation if for all $(P, Q) \in \mathcal{R}$, whenever:*

i. $P \xrightarrow{\mu} P'$ *and* μ *is not an input, then* $Q \xrightarrow{\mu} Q'$ *and* $P' \mathcal{R} Q'$.

ii. $P \xrightarrow{a\tilde{b}} P'$ *then either* $Q \xrightarrow{a\tilde{b}} Q'$ *and* $P' \mathcal{R} Q'$ *or* $Q \xrightarrow{\tau} Q'$ *and* $P' \mathcal{R} (Q' \mid \overline{a}\langle\tilde{b}\rangle)$.

We write $P \dot{\sim}_a Q$ *if* $P \mathcal{R} Q$ *for some asynchronous bisimulation* $\mathcal{R}$.

The definition of asynchronous bisimulation is thus as the standard definition for bisimulation except for the case of input actions. This part expresses that if a process absorbs what it has just emitted, this can be "absorbed" in an internal action. In an asynchronous π-calculus without matching, $\dot{\sim}_a$ is a congruence and coincides with asynchronous barbed congruence. With matching $\dot{\sim}_a$ is not a congruence, and (as usual) we need to close under substitutions to get the induced congruence which coincides with with asynchronous barbed congruence (c.f. [ACS98])

Example 4.3 An example (from [ACS98]) of the significance of not being able to observe input action is that we have the following algebraic law:

$$a(b).(\overline{a}\langle b\rangle \mid P) + \tau.P \dot{\sim}_a \tau.P \text{ with } b \notin \mathsf{fn}(P)$$

□

It is easy to see that if two asynchronous π-calculus terms are strongly bisimilar under the ordinary definition of strong bisimulation (Definition 4.1), then they are also asynchronous bisimilar. This will allow us in some cases to establish that $P \dot{\sim}_a Q$ by using the simpler definition of $\dot{\sim}$.

π_a has several important advantages compared to the full π-calculus. The various notions of bisimulation equivalence which differ for the synchronous π-calculus, coincide in the asynchronous calculus [HHK95, ACS98]. Further, bisimulation is a congruence, a result that does not hold in the synchronous case (or if we introduce a matching operator). Finally, the equational theory becomes simpler — in the synchronous case, the matching operator is needed to give an equational theory, whereas this is not needed in the asynchronous π-calculus.

The local π-calculus In the local π-calculus of Merro and Sangiorgi [MS98], denoted Lπ, we restrict π_a to

i. only allow a recipient of a name to use that name in output actions,

ii. disallow a match or mismatch construct.

This might seem like a very degenerate version of the π-calculus but, again, most uses of the π-calculus obey the above mentioned requirements.

The definitions of barbed bisimulation and congruence are the same as for π_a. But now, the alternative characterization using bisimulation becomes a little more involved and we refer the reader to [MS98] for its definition.

Some of the properties of Lπ processes also hold for processes of the full π-calculus. For instance assume that the name x in the following obeys the requirements of Lπ and is only used for output in P_1 and P_2, then the following law (from [Mil93]) holds:

$$(\nu x)(P_1 \mid P_2 \mid !x(\tilde{y}).Q) \sim (\nu x)(P_1 \mid !x(\tilde{y}).Q) \mid (\nu x)(P_2 \mid !x(\tilde{y}).Q)$$

4.5 The lazy λ-calculus in the π-calculus

Just as for the ς-calculus, there exists a simple translation of the lazy λ-calculus into the π-calculus, as first shown by Milner [Mil92]. We shall discuss the translation in some detail since it has many similarities with the translations we shall later show for the ς-calculus.

The following translation from [San95] maps λ-calculus terms into the π-calculus (in fact, into Lπ):

$$\begin{aligned}
[\![x]\!]_p &\triangleq \overline{x}\langle p\rangle \\
[\![\lambda(x)b]\!]_p &\triangleq p(x,q).[\![b]\!]_q \\
[\![b(a)]\!]_p &\triangleq (\nu r t)([\![b]\!]_r \mid \overline{r}\langle t,p\rangle \mid !t(q).[\![a]\!]_q) \quad t \notin \mathsf{fn}([\![b]\!]_r \mid [\![a]\!]_q)
\end{aligned}$$

In the translation, we assume that the names p, q, r and t are different from variables in "λ-terms".

The core of translation is the translation of application. In the translation of the application $b(a)$ we first evaluate $[\![b]\!]_r$ where r is a private name (observe that in $!t(q).[\![a]\!]_q$ the input prefix blocks evaluation of $[\![a]\!]_q$). The private name r is used to free the evaluation of $[\![b]\!]_r$, and to supply $[\![b]\!]_r$ with a trigger-name (t), that can be used to start the translation of a. What happens is that $[\![b]\!]_r$ when evaluating to something like the translation of a λ-abstraction, it will read the $\overline{r}\langle t,p\rangle$ message, gaining access to the argument located at t.

Example 4.4 To see how this works, consider the λ-term $(\lambda y.y(b))(a)$ with y not free in b. This term can do a β-reduction and become $a(b)$. We now show how the translation into the π-calculus can mimic the reduction via a series of reductions:

$$
\begin{array}{rl}
 & [\![(\lambda y.y(b))(a)]\!]_p \\
= & (\nu rt)([\![\lambda y.y(b)]\!]_r \mid \overline{r}\langle t,p\rangle \mid\, !t(q).[\![a]\!]_q) \\
= & (\nu rt)(r(y,q).[\![y(b)]\!]_q \mid \overline{r}\langle t,p\rangle \mid\, !t(q).[\![a]\!]_q) \\
\xrightarrow{\tau}_d\sim & (\nu t)([\![y(b)]\!]_p\{{}^t\!/_y\} \mid\, !t(q).[\![a]\!]_q) \\
= & (\nu t)((\nu rt')(\overline{t}\langle r\rangle \mid \overline{r}\langle t',p\rangle \mid\, !t'(q).[\![b]\!]_q) \mid\, !t(q).[\![a]\!]_q) \\
\xrightarrow{\tau}_d\sim & (\nu t)((\nu rt')([\![a]\!]_r \mid \overline{r}\langle t',p\rangle \mid\, !t'(q).[\![b]\!]_q) \mid\, !t(q).[\![a]\!]_q)
\end{array}
$$

In the final expression note that t does not occur free in the expression $(\nu rt')([\![a]\!]_r \mid \overline{r}\langle t',p\rangle \mid\, !t'(q).[\![b]\!]_q)$. This implies that the expression $!t(q).[\![a]\!]_q$ is superfluous and can be removed, resulting in the term $(\nu rt')([\![a]\!]_r \mid \overline{r}\langle t',p\rangle \mid\, !t'(q).[\![b]\!]_q)$ (formally the two terms are strongly congruent), which is the translation of $a(b)$. □

So, what does it mean that the translation is correct? Considering the translation as a kind of implementation of the lazy λ-calculus, we would expect the translation to be able to match evaluations in the λ-calculus.

A first suggestion would be that if $a \to \lambda x.b$, then $[\![a]\!]_p \Rightarrow [\![\lambda x.b]\!]_p$. This is not entirely true as we can see from the above example. What instead happens is that $[\![a]\!]_p$ reduces to some process P that is weakly bisimilar to $[\![\lambda x.b]\!]_p$. Furthermore if $[\![a]\!]_p \Rightarrow P$ and $P{\downarrow_q}$, then $q = p$ and $a \to \lambda x.b$ and $P \approx [\![\lambda x.b]\!]_p$. The first statement states that the translation can mimic the reductions of the λ-calculus, and the second that the translation does not exhibit some extra behavior. These two results ensure what is called *operational correspondence*.

Another correctness criteria for translations is that of *full abstraction* w.r.t. some equivalence in the source calculus and target calculus. Let $\simeq$ be some notion of equivalence defined on terms from the source calculus and $\simeq'$ an equivalence on terms of the target calculus, then a translation is fully abstract, if $a \simeq b$ implies $[\![a]\!]_p \simeq' [\![b]\!]_p$ and $[\![a]\!]_p \simeq' [\![b]\!]_p$ implies $a \simeq b$. Full abstraction allows one to use equivalences defined in the target calculus to prove equivalence between agents in the source calculus. But because of the expressiveness of the π-calculus, full abstraction is normally not achieved when translating sequential calculi. Instead what one gets is that equivalence between translated terms implies equivalence between the original terms. However, if two translated terms are inequivalent in the target calculus, nothing conclusive can be said about their equivalence in the source calculus.

For a detailed account of the relation between the λ-calculus and its translations into the π-calculus, we refer the reader to [San94a, San94b, San95].

Chapter 5

Translating the untyped Functional ς-calculus

We start our study of the applicability of process calculus techniques to the ς-calculus by giving a translation of the untyped Functional ς-calculus into the π-calculus.

Such an encoding can provide us with several insights into the fundamentals of implementing object-oriented languages. Most object-oriented languages are implemented using a notion of references and in particular for the notion of self, whereas the Functional ς-calculus uses substitution (c.f. Table 3.2). In the π-calculus references are part and parcel of the syntax in the guise of names. Thus, an encoding of the ς-calculus into the π-calculus directly shows how references can be used to represent substitution and the notion of self. Furthermore, the concurrent nature of the π-calculus provides us with an idea of how to implement an object-oriented language in a distributed setting.

There are additional benefits from the point of view of verification of program properties, provided that the encoding is sound in the sense that it will 'internalize' the object calculus by reflecting behavioral properties of object terms. For when the encoding is sound in this sense, one can reason about the behavior of objects at the π-calculus level, as shown for the Imperative ς-calculus in Chapter 6.

Because subsets of the π-calculus usually allows more terms to be equated than the full calculus, it seems natural to try to use as small a subset of the π-calculus as possible. Therefore, we start our investigation by trying to use the local π-calculus (Lπ) introduced in Section 4.4.

Before we start considering how to encode the ς-calculus, let us briefly review the basic requirements of a "good" translation inspired by Palamidessi

in [Pal97]:

i. Compositionality.

ii. Preservation of some intended semantics.

Compositionality means that the translation of a term $[\![a \otimes b]\!]$, where $\otimes$ is some operator in the source language, is given in terms of the translations $[\![a]\!]$ and $[\![b]\!]$. Compositionality naturally makes the use of the translation to reason about the source language easier. Something even better, but more difficult to obtain, is *uniformity* — that certain operators in the source language translate directly to corresponding operators in the target languages. So, if our hypothetical translation of $a \otimes b$ is $[\![a \otimes b]\!] \stackrel{\triangle}{=} [\![a]\!] \oplus [\![b]\!]$, then the translation is said to be uniform w.r.t. $\otimes$. [1].

Requirement ii, preservation of some intended semantics, is the very basic requirement. A translation would not be worth much if it did not respect the intended semantics of the source language. In the case of our translation of the ς-calculus into the π-calculus, our basic requirement is that the translation is *operationally correct*, meaning that the π-calculus translation can mimic reductions in the ς-calculus. In this chapter, we shall show that the encoding is operationally correct in the sense that it respects reductions. We also show that our encoding is *not* fully abstract with respect to weak barbed congruence in the π-calculus and consider the restrictions which we will need to impose on terms of the π-calculus in order to achieve full abstraction.

The structure of the rest of the chapter is as follows: In Section 5.1 we define our encoding of the ς-calculus into the asynchronous π-calculus and discuss alternative encodings. In Section 5.2 we give a few examples showing how the final translation works. Section 5.3 state and prove the main results concerning the operational correspondence of the encoding. Section 5.4 examines the relation between equivalences of ς-calculus terms and their encodings. Finally, in Section 5.5 we discuss the results of this chapter.

5.1 The encoding

In what follows, we shall assume that the sets of method names and self variables in the encoding coincide with the set of method names and self

[1] The difference in our definition compared to Palamidessi is firstly, instead of clause ii, she requires a reasonable semantics, that is a semantics that respects divergence. Secondly, we use a more general notion of uniformity. In her terminology, uniformity means that parallel composition and restriction in the source language are translated as parallel composition and restriction in the target language.

variables in the ς-calculus.

Let us start by considering how we can translate a degenerate version of the untyped ς-calculus which does not have method override. The central idea of the encoding is that, as soon as an object term reduces to a value (i.e. a variable or an object record), an object reference is output on a special, designated value channel.

An object reference is used to activate methods. In order to activate a method, its surrounding object is passed the name of the method, and a reply (value) channel where the location of the result of the method is returned. We shall represent an object value as a replicated process located at some name s, that on reception of a method name l and a reply channel r will activate the method bound to l. When activating a method, we must ensure that the self variable of the method is correctly bound to its enclosing object.

A first attempt at an encoding of object values might resemble something like:

$$[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]]\!]_p \ \triangleq\ (\nu s)\Big(\overline{p}\langle s\rangle \mid\ !s(m,r).\overline{m}\langle s,r\rangle \mid \prod_{i\in I} !\mathrm{l}_i(x_i,q).[\![b_i]\!]_q\Big)$$

In this translation p denotes the value channel on which we publish a reference s, where the object can be reached. The replicated part $!s(m,r).\overline{m}\langle s,r\rangle$ can repeatedly receive a pair consisting of a method name (bound to m) and a reply channel (bound to r). The received pair is used to select the correct method by outputting on m the self s of the object and the reply channel r. Since m should denote a method name, one of the methods located at some l_i should be able to synchronize and start computing.

There is just one minor problem with this translation. A method can be activated if an attempt is made to activate a method with the same name in another object. This is possible, because methods are always ready to be activated by a signal on their method name. This can not be prevented by making the method names private to the object, for then we would not be able to activate the methods. To remedy this we need to ensure that a method is only ready to receive on its method name when its surrounding object is active. To do this we move the methods beneath the input prefix on s, and replace the replicated parallel composition with a input-guarded choice. This ensures i) that the methods can only be activated when the object has received a request to do so, and ii) that the methods not selected are preempted when activating the right one. Table 5.1 presents the full translation for the Functional ς-calculus without method override:

In the case of method activation $a.\mathrm{l}$, we see how we first create a private reply channel q, then evaluate $[\![a]\!]_q$ while we wait for a reply to be returned along q. The reply we receive along q is a pointer to the object resulting from

$$
\begin{array}{rcl}
[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]]\!]_p & \triangleq & (\nu s)\Big(\overline{p}\langle s\rangle \mid !s(m,r).\Big(\overline{m}\langle s\rangle \mid \sum_{i\in I}\mathrm{l}_i(x_i).[\![b_i]\!]_r\Big)\Big) \\
[\![a.\mathrm{l}]\!]_p & \triangleq & (\nu q)([\![a]\!]_q \mid q(x).\overline{x}\langle \mathrm{l}, p\rangle) \\
[\![x]\!]_p & \triangleq & \overline{p}\langle x\rangle
\end{array}
$$

Table 5.1: A translation of ς-calculus without method override.

evaluating a. Now all we need to do is to request activation of the method l with the reply channel p. In the translation of object values we have made a minor optimization compared to our previous, flawed translation — instead of passing the reply channel around when selecting the right method, we now use the fact that the method bodies are located beneath the input prefix where we receive the reply channel.

Now all we need to do is to figure out how to deal with method override. Let us recall the semantics of method override given in Section 3.2, the reduction rule was:

$$a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b \ \rightsquigarrow\ [\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}] \quad j\in I$$

That is, a method override on an object value creates a *new* object with the only difference from the old one being that we have replaced the method bound to l_j with a new body. Or, to put it differently, the only difference between the old and the new object is the method bound to l_j. So if we can somehow modify the encoding such that we can express that an object is just like another except for some new method, then we would be done. In order to do this we need a 'relay construct' that must pass on requests for a method call to the new body of l if l is invoked. In all other cases, the 'relay process' will pass the request to the old method bodies. Of course, we need to ensure that all occurrences of self in the old method bodies are correctly bound to the new object.

This means that the location s of the translation of an object value no longer needs to denote the self of its methods, since an override operation of one of its methods leads to the creation of a new self. Therefore, we must on activation of a method provide the location of the *current self* such that method lookup can begin at the right place. This idea of the encoding is illustrated in Figure 5.1.

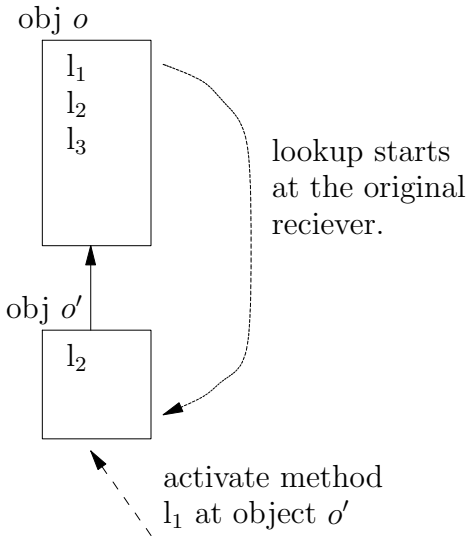


Figure 5.1: Method override and lookup.

5.1.1 An encoding into Lπ

Now that we have the basic principles of the translation, we can proceed to give a translation of the Functional ς-calculus into Lπ extended with input-guarded choice. We start out by giving the sorting that we want the translation to obey:

Method names	$l, m:$	Method	$\rightarrow$	(ObjRef)
Reply channels	$p, q, r:$	RepCh	$\rightarrow$	(ObjRef)
Object references	$s, t:$	ObjRef	$\rightarrow$	$(\mathit{Method}, \mathit{ObjRef}, \mathit{RepCh})$
Self variables	$x, y, z, u:$	ObjRef		

In the sorting given above the clause $p, q, r : \mathit{RepCh} \rightarrow (\mathit{ObjRef})$ expresses that p, q and r range over the set RepCh of reply channels and that names from the set ObjRef of object reference names are transmitted over reply channels. For reasons of simplicity we shall assume that m (with a possible index or mark) does not belong to the set **MNames** of ς-calculus method names (but it can be instantiated via an input action to become a method name).

The encoding function $[\![\cdot]\!]_p$ is defined in Table 5.2.

Compared to the previous translation, which did not consider method override, we see that we now on method activation pass the object reference as a parameter on the object reference itself. In the case where there is no

$$
\begin{array}{rcl}
[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]]\!]_p & \triangleq & (\nu s)\Big(\overline{p}\langle s\rangle \mid !s(m,y,r).\Big(\overline{m}\langle y\rangle \mid \sum_{i\in I} \mathrm{l}_i(x_i).[\![b_i]\!]_r\Big)\Big) \\
 & & \text{where } y \notin \cup_{i\in I}\mathsf{fv}(b_i) \\
[\![a.\mathrm{l}]\!]_p & \triangleq & (\nu q)([\![a]\!]_q \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle) \\
[\![x]\!]_p & \triangleq & \overline{p}\langle x\rangle \\
[\![a.\mathrm{l}{\Leftarrow}\varsigma(x)b]\!]_p & \triangleq & (\nu q)\Big([\![a]\!]_q \mid q(z).(\nu t)(\overline{p}\langle t\rangle \mid !t(m,y,r). \\
 & & \Big(\overline{m}\langle y\rangle \mid \mathrm{l}(x).[\![b]\!]_r + \sum_{\mathrm{l}'\in M\setminus\{\mathrm{l}\}} \mathrm{l}'(x).\overline{z}\langle m,y,r\rangle\Big)\Big)\Big) \\
 & & \text{where } y, z \notin \mathsf{fv}(b)
\end{array}
$$

Table 5.2: The translation from the untyped Functional ς-calculus to Lπ.

method override present, this changes nothing compared with the previous encoding, since all we have done is to move the installation of the self of a method from the translation of an object value to the translation of a method activation. But this change allows us to control where method lookup should start, which is exactly what we need in order to deal with method override.

The interesting case is the one dealing with method override $a.\mathrm{l}{\Leftarrow}\varsigma(x)b$. First, we evaluate the translation of a at a private name q where we receive an object reference that is bound to the name z. Then, the translation method override returns a new object reference t, and start a replicated process located at t that receives requests for methods activations. But the only method we have available is the overriding method bound to l, so for all other requests we need to resend them to z. Observe that, when resending, we do not change the object reference y that denotes the "true" self of the object. This allows a request for method activation on self to start at the outermost series of overrides.

The last detail of the translation is the set M in the relay

$$\sum_{\mathrm{l}'\in M\setminus\{\mathrm{l}\}} \mathrm{l}'(x).\overline{z}\langle m,y,r\rangle$$

for methods different from l. Ideally, M should be (or at least contain) the set of method labels that the object a has. So how can we determine this

when translating a method override? The following proposals come to mind (unfortunately most of them do not work):

i. Let M be the set of all names having sort *Method*. This is not allowed, since the sort *Method* must contain an infinite number of names, and we only allow finite summation in the π-calculus.

ii. Take the set of method labels occurring in a. This might seem like a good suggestion; however, it does not work. This is due to the fact that we can override *self* as in $y.\mathrm{l}\Leftarrow\varsigma(x)b$ where the expression is embedded in some method binding y to an object.

iii. Use type information. We could consider using a typed version of the ς-calculus and then try to use type information to figure out which methods to relay.

The typing rule for override states that if we can deduce that a has type $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$ and assuming the self of the method to have type A we can deduce b to have type B_j, then $a.\mathrm{l}_j\Leftarrow\varsigma(x{:}A)b$ has type A.

This might lead us to think that we simply need to relay the methods l_i for $i \in I \setminus \{j\}$. But this also fails because of the subsumption rule, that allows us to "hide" methods in an object.

iv. Forget compositionality and let M be the set of *all* method names in the expression that we are translating.

The only working solution is iv, but sacrificing compositionality will complicate the use of the translation for reasoning about objects. Another aspect that makes the translation less useful is that method selection happens by communication on *free names*; implying that for the translation of two terms a and b to be, for instance, weakly bisimilar, they need to show exactly the same sequences of method activations. This problem could be prevented by restricting all method names from a and b at the outermost level of the translations. But the restrictions prevents us from us from activating any methods from the outside, resulting in an almost useless translation.

Given all these problems and disadvantages of the translation in Table 5.2 we might consider using a more powerful version of the π-calculus. From the above discussion, we see that the problem with compositionality is caused by the way we relay method calls in the translation of method override. Recall what the intended meaning of the translation of method override is: Namely, that we look at the method name received and, if the name is identical to the method we have overridden, then activate the new method body, *otherwise*

pass the request on to the old object. This description of override tells us that the extra power we need is an *if-then-else* construct for testing equality of names, which is exactly what the match and mismatch constructs can be used for.

Another good reason for abandoning the use of communication for describing method selection, is that such a translation will not work in a concurrent setting where several method activations might happen simultaneously. This might not seem as a problem with the present version of the ς-calculus. However, in the proof of the correctness of the translation the problem might surface, simply because of the concurrent nature of the π-calculus.

5.1.2 A translation using π_a

It is straightforward to modify the translation from Table 5.2 to use match and mismatch on method names instead of communication to select the correct method. The only minor complication in the new translation, presented in Table 5.3, is that we need to insert an extra communication on a private name after having selected the right method in order to install the self of the method (in the previous translation we used the communication on the method name to do that)[2].

To make the translation more readable we write $s{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]]\!]$ for the term $!s(m,y,r).(\nu q)(\overline{q}\langle y\rangle \mid \prod_{i\in I}[m{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r)$ denoting the translation of an object value *located* at the name s. For the translation of method override we shall use a similar abbreviation, letting $t{:=}s.[\![\mathrm{l}{\Leftarrow}\varsigma(x)b]\!]$ denote the term $!t(m,y,r).(\nu q)(\overline{q}\langle y\rangle \mid [m{=}\mathrm{l}]q(x).[\![b]\!]_r \mid [m{\neq}\mathrm{l}]q(x).\overline{s}\langle m,y,r\rangle)$, that is an override of l located at t that on activations different from l will relay to the object located at s. We shall also, when they are obvious from the context, omit index sets in expressions.

The internal workings of the new translation are the same as in the previous one, except for one minor detail. In the translation of method override, in the case where the mismatch applies, we read the message located at the private name q' but do not use the input name for anything. This might seem like a superfluous input that could be removed. But again, as we shall see later, the removal of the message at q' makes reasoning about the translation easier.

[2]We could avoid the communication by ensuring that all methods in an object uses the same self variable (by $[\![b_i]\!]_r\{y/x_i\}$, exchanging the self variable x_i in each method b_i with the variable y common to all methods in the surrounding object), however the extra communication makes reasoning easier (see Example 5.2).

$$
\begin{array}{rcl}
[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]]\!]_p & \triangleq & (\nu s)\Big(\overline{p}\langle s\rangle \mid !s(m,y,r).(\nu q)\Big(\overline{q}\langle y\rangle \mid \\
& & \prod_{i\in I}[m{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r\Big)\Big) \\
& & \text{where } y \notin \cup_{i\in I}\mathsf{fv}(b_i) \\
[\![a.\mathrm{l}]\!]_p & \triangleq & (\nu q)([\![a]\!]_q \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle) \\
[\![x]\!]_p & \triangleq & \overline{p}\langle x\rangle \\
[\![a.\mathrm{l}{\Leftarrow}\varsigma(x)b]\!]_p & \triangleq & (\nu q)\Big([\![a]\!]_q \mid q(z).(\nu t)\Big(\overline{p}\langle t\rangle \mid !t(m,y,r).(\nu q')(\overline{q}'\langle y\rangle \mid \\
& & [m{=}\mathrm{l}]q'(x).[\![b]\!]_r \mid [m{\neq}\mathrm{l}]q'(x).\overline{z}\langle m,y,r\rangle)\Big)\Big) \\
& & \text{where } y, z \notin \mathsf{fv}(b)
\end{array}
$$

Table 5.3: The translation from the untyped Functional ς-calculus to π_a with match and mismatch.

5.2 Examples

Before considering how to prove that the translation is operationally correct, we shall present a few examples showing how the translation can mimic the reductions of ς-calculus terms

Example 5.1 Our first example is the encoding of the following simple object:

$$a = [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]$$

It contains only one method l, that when activated will activate itself, leading to the infinite reduction sequence $a.\mathrm{l} \rightsquigarrow a.\mathrm{l} \rightsquigarrow \cdots$. In our encoding this gives rise to the following behavior:

$$
\begin{array}{rl}
 & [\![a.\mathrm{l}]\!]_p \\
= & (\nu q)\Big((\nu s)(\overline{q}\langle s\rangle \mid s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!]) \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle\Big) \\
\xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!] \mid \overline{s}\langle \mathrm{l}, s, p\rangle\Big) \\
\xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!] \mid (\nu q)(\overline{q}\langle s\rangle \mid [\mathrm{l}{=}\mathrm{l}]q(x).[\![x.\mathrm{l}]\!]_p)\Big) \\
\sim & (\nu s)\Big(s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!] \mid (\nu q)(\overline{q}\langle s\rangle \mid q(x).(\nu q')([\![x]\!]_{q'} \mid q'(y).\overline{y}\langle \mathrm{l}, y, p\rangle))\Big) \\
\xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!] \mid (\nu q')(\overline{q}'\langle s\rangle \mid q'(y).\overline{y}\langle \mathrm{l}, y, p\rangle)\Big) \\
\sim & (\nu q)\Big((\nu s)(\overline{q}\langle s\rangle \mid s{:=}[\![[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}]]\!]) \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle\Big) \\
= & [\![a.\mathrm{l}]\!]_p
\end{array}
$$

Here we see how one ς-calculus reduction is mimicked by three deterministic internal communications by the translation. The first use of strong bisimulation congruence is an application of the law $[a{=}a]P \sim P$ allowing us to remove a successful match. The second instance is an application of the laws that allow us to move restrictions around as long as we do not capture any free names or release any bound. □

In our next example we see how method override works in the translation.

Example 5.2 Consider the object

$$a = [\mathrm{l}_1{=}\varsigma(x)x,\ \mathrm{l}_2{=}\varsigma(x)b]$$

If we override l_1 or l_2, we obtain a new object with the old method replaced by the overriding one. As mentioned earlier, the encoding describes this by means of a 'relay', a process that handles activations of the overriding method itself and *resends* all other method activations to the original object.

Let us consider the expression $(a.\mathrm{l}_2{\Leftarrow}\varsigma(x)x.\mathrm{l}_1).\mathrm{l}_2$. The semantics of the ς-calculus gives the following sequence of reductions:

$$
\begin{array}{rcl}
(a.\mathrm{l}_2{\Leftarrow}\varsigma(x)x.\mathrm{l}_1).\mathrm{l}_2 & \rightsquigarrow & [\mathrm{l}_1{=}\varsigma(x)x,\ \mathrm{l}_2{=}\varsigma(x)x.\mathrm{l}_1].\mathrm{l}_2 \\
 & \rightsquigarrow & [\mathrm{l}_1{=}\varsigma(x)x,\ \mathrm{l}_2{=}\varsigma(x)x.\mathrm{l}_1].\mathrm{l}_1 \\
 & \rightsquigarrow & [\mathrm{l}_1{=}\varsigma(x)x,\ \mathrm{l}_2{=}\varsigma(x)x.\mathrm{l}_1]
\end{array}
$$

Because the encoding of override behaves like a relay construct we cannot expect the translation to evaluate to something that is exactly (modulo garbage

collection and placement of restrictions) the translation of $[\mathrm{l}_1{=}\varsigma(x)x,\ \mathrm{l}_2{=}\varsigma(x)x.\mathrm{l}_1]$. The following computation shows what we get instead:

$$
\begin{array}{rl}
 & [\![(a.\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1).\mathrm{l}_2]\!]_p \\
= & (\nu q)\Big((\nu q')\Big((\nu s)(\overline{q}'\langle s\rangle \mid s{:=}[\![a]\!]) \mid q'(x).(\nu t)(\overline{q}\langle t\rangle \mid \\
 & \quad t{:=}x.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!])\Big) \mid q(x).\overline{x}\langle \mathrm{l}_2, x, p\rangle\Big) \\
\xrightarrow{\tau}{}^{2}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid \overline{t}\langle \mathrm{l}_2, t, p\rangle\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!] \mid \\
 & \quad (\nu q)(\overline{q}\langle t\rangle \mid [\mathrm{l}_2{=}\mathrm{l}_2]q(x).[\![x.\mathrm{l}_1]\!]_p \mid [\mathrm{l}_2{\neq}\mathrm{l}_2]q(x).\overline{s}\langle \mathrm{l}_2, t, p\rangle)\Big)\Big) \\
\sim & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid (\nu q)(\overline{q}\langle t\rangle \mid \\
 & \quad q(x).[\![x.\mathrm{l}_1]\!]_p)\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid (\nu q)(\overline{q}\langle t\rangle \mid \\
 & \quad q(x).\overline{x}\langle \mathrm{l}_1, x, p\rangle)\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid \overline{t}\langle \mathrm{l}_1, t, p\rangle\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!] \mid \\
 & \quad (\nu q)(\overline{q}\langle t\rangle \mid [\mathrm{l}_1{=}\mathrm{l}_2]q(x).[\![x.\mathrm{l}_1]\!]_p \mid [\mathrm{l}_1{\neq}\mathrm{l}_2]q(x).\overline{s}\langle \mathrm{l}_1, t, p\rangle)\Big)\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!] \mid \\
 & \quad (\nu q)([\mathrm{l}_1{=}\mathrm{l}_2]q(x).[\![x.\mathrm{l}_1]\!]_p) \mid \overline{s}\langle \mathrm{l}_1, t, p\rangle\Big)\Big) \\
\sim & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!] \mid \overline{s}\langle \mathrm{l}_1, t, p\rangle\Big)\Big) \\
\xrightarrow{\tau}_{d}\sim & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid \\
 & \quad (\nu q)(\overline{q}\langle t\rangle \mid [\mathrm{l}_1{=}\mathrm{l}_1]q(x).\overline{p}\langle x\rangle \mid [\mathrm{l}_1{=}\mathrm{l}_2]q(x).[\![b]\!]_p)\Big) \\
\xrightarrow{\tau}_{d} & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid \\
 & \quad \overline{p}\langle t\rangle \mid (\nu q)([\mathrm{l}_1{=}\mathrm{l}_2]q(x).[\![b]\!]_p)\Big) \\
\sim & (\nu t)\Big((\nu s)\Big(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_2 \Leftarrow \varsigma(x)x.\mathrm{l}_1]\!]\Big) \mid \overline{p}\langle t\rangle\Big)
\end{array}
$$

Observe how the object reference of the *original* receiver is passed on when the receiver does not have the requested method itself. This ensures that when the correct method is found, we know where to start looking for other methods. Looking at the translation, we see that the term $(\nu t)((\nu s)(s{:=}[\![a]\!] \mid$

$t{:=}s.[\![\mathrm{l}_2{\Leftarrow}\varsigma(x)x.\mathrm{l}_1]\!]$) represents the object $[\mathrm{l}_1{=}\varsigma(x)x$, $\mathrm{l}_2{=}\varsigma(x)x.\mathrm{l}_1]$ located at t. In the following section we shall show that this is true in a precise sense.

In the above example, we apply a law saying $[a{\neq}a]P \sim \mathbf{0}$ to garbage collect dead parts of the translation. One might expect a similar law to hold for matching, as $[a{=}b]P \sim \mathbf{0}$, but this is not true in all contexts. For instance the context $c(b)[\cdot]$ is able to distinguish between the two terms, as an input action can lead to the identification of a and b (of course, the translation never creates such contexts). Therefore the omission of the term $(\nu q)([\mathrm{l}_1{=}\mathrm{l}_2]q(x).[\![x.\mathrm{l}_1]\!]_p)$ in the second use of $\sim$ relies on the fact that the input prefix $q(x)$ is on a name private to that term, implying that the term is inactive. (This explains why it is advantageous to remove the message located at q' in the mismatch case of method override). □

5.3 Operational correspondence

In this section we shall show the operational correctness of the encoding.

First note that our translation is more liberal than the semantics for the untyped ς-calculus given in Table 3.2 because the translation not only works for updates of existing methods but also for addition of new methods to an object. For the untyped ς-calculus there is nothing wrong in allowing this behavior, so we shall assume the following more liberal rule for method override

$$a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b \;\rightsquigarrow\; [\mathrm{l}_i{=}\varsigma(x_i)b_i,\, \mathrm{l}_j{=}\varsigma(x)b^{\;i\in I\setminus\{j\}}]$$

Notice that we have dropped the requirement $j \in J$.

The examples from the previous section hint at the kind of results we can expect. The last example makes it clear that we cannot expect to prove a very tight operational correspondence between reductions in the ς-calculus and its translation into the π-calculus. For instance, the ideal result would be something like

$$a \rightsquigarrow b \text{ iff } [\![a]\!]_p \xrightarrow{\tau}{}^{*}_{d}\sim [\![b]\!]_p$$

where $\sim$ is used to garbage collect unreachable parts and rearrange terms. But this cannot hold, simply because the relay construct used to handle method override, as in the translation of $[\mathrm{l}_i{=}\varsigma(x_i)b_i{}^{\;i\in I}].\mathrm{l}_j{\Leftarrow}\varsigma(x)b$, requires some extra internal communication in order to mimic that of $[\mathrm{l}_i{=}\varsigma(x_i)b_i,\, \mathrm{l}_j{=}\varsigma(x)b^{\;i\in I\setminus\{j\}}]$. This observation implies that we need something weaker than $\sim$ to relate reductions in the ς-calculus with the behavior of the translation. So instead of strong bisimulation congruence we shall use a weak congruence relation, for

example weak bisimulation congruence ($\approx$). Using a congruence relation instead of just an equivalence relation will make reasoning easier. For instance, if we know that some π-calculus term P is congruent to the translation of the object a located at some name p, this will allow us to exchange P with $[\![a]\!]_p$ in every π-calculus context $C[\cdot]$.

Since weak congruence disregards internal communication we can remove the sequence of deterministic τ-transitions from the above correctness criteria and instead state that operational correctness in our case is something like:

$$a \rightsquigarrow^* b \text{ iff } [\![a]\!]_p \approx [\![b]\!]_p$$

We need to consider (possibly empty) sequences of reductions because $[\![a]\!]_p \approx [\![a]\!]_p$.

By definition, two processes are weakly congruent if they are weakly bisimilar under all substitutions obeying the sorting of the translation. Unfortunately, this does not hold for the translation if we use the sorting from Section 5.1. To see why, consider the following simple object:

$$a = [\mathrm{l}_1{=}\varsigma(x)b, \mathrm{l}_2{=}\varsigma(x)c]$$

We would like the translation to have the property that $[\![a.\mathrm{l}_1]\!]_p \approx [\![b\{^a/_x\}]\!]_p$. But if we use the substitution $\sigma = \{^{\mathrm{l}_1}/_{\mathrm{l}_2}\}$ mapping the method name l_2 to l_1 then the translation $[\![a.\mathrm{l}_1]\!]_p\sigma$ will have a nondeterministic choice between running the translation of b or c, whereas the translation $[\![b_1\{^a/_x\}]\!]_p\sigma$ already has resolved this choice. This implies that the two terms cannot be weakly congruent under the sorting given in Section 5.1.

For π-calculus terms resulting from the translation this is not a problem, as we *never* bind a method name (we always use the name m that by convention is not a method name). This requirement must be extended to all π-calculus terms by a more refined sorting stating exactly that method labels cannot be bound by an input. This solves the problem, since we then cannot end up identifying two method labels. At first glance this requirement might seem somewhat ad hoc, but it simply amounts to stating that the names from **MName** are *constants* and that one cannot go around substituting constants with other constants. This restriction translates into a constraint on substitutions, that a substitution σ must not identify method names. It is easy to see from the translation that in the case of closed objects we do not need to consider substitutions at all, when we want to establish congruence properties, since there is only one free name that is not a method label (the reply channel). A final observation worth making is that, except for names from *Method* (that are matched against each other), all other names conform

to the requirements of Lπ. This allows us to use laws valid only for Lπ on all names.

In the following, let $\sim_b^\varsigma$ ($\approx_b^\varsigma$) denote (weak) asynchronous barbed congruence on contexts obeying the above-motioned requirements and similarly $\leq^\varsigma$ and $\leq_b^\varsigma$ denotes expansion congruence and barbed expansion congruence respectively defined over the same set of processes.

The two following lemmas provide the basis for the proof of soundness of method activation and method override. Our first lemma states that locating an object at an object reference and binding the object reference to a self variable, is equivalent to substituting the object for the self variable:

Lemma 5.1 (Substitution) *Let a denote an object value $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ and let b be an arbitrary object, then the following property holds:*

$$(\nu s)(s{:=}[\![a]\!] \mid [\![b]\!]_p\{^s/_x\}) \sim_b^\varsigma [\![b\{^a/_x\}]\!]_p$$

Proof. We prove the property using induction on the structure of b, applying algebraic laws to bring the left hand side into the shape of the right hand side.

$b = y$: We have two cases; either $y = x$ and we have:

$$\begin{array}{rcl}(\nu s)(s{:=}[\![a]\!] \mid [\![x]\!]_p\{^s/_x\}) & = & (\nu s)(s{:=}[\![a]\!] \mid \overline{p}\langle s\rangle)\\ & = & [\![a]\!]_p\\ & = & [\![x\{^a/_x\}]\!]_p\end{array}$$

or $y \neq x$ and:

$$\begin{array}{rcl}(\nu s)(s{:=}[\![a]\!] \mid [\![y]\!]_p\{^s/_x\}) & = & (\nu s)(s{:=}[\![a]\!] \mid \overline{p}\langle y\rangle)\\ & \sim & (\nu s)(s{:=}[\![a]\!]) \mid \overline{p}\langle y\rangle\\ & \sim & \overline{p}\langle y\rangle\\ & = & [\![y]\!]_p\\ & = & [\![y\{^a/_x\}]\!]_p\end{array}$$

$b = b'.\mathrm{l}$: Assume w.l.o.g. that $q \notin \mathsf{fn}(s{:=}[\![a]\!])$ (this will allow us to move $s{:=}[\![a]\!]$ beneath a restriction on q without capturing free occurrences of q in $s{:=}[\![a]\!]$). This is used in the following calculation:

$$\begin{array}{rcl} & & (\nu s)(s{:=}[\![a]\!] \mid [\![b'.\mathrm{l}]\!]_p\{^s/_x\})\\ & = & (\nu s)(s{:=}[\![a]\!] \mid (\nu q)([\![b']\!]_q\{^s/_x\} \mid q(y).\overline{y}\langle \mathrm{l}, y, p\rangle))\\ & \sim & (\nu q)((\nu s)(s{:=}[\![a]\!] \mid [\![b']\!]_q\{^s/_x\}) \mid q(y).\overline{y}\langle \mathrm{l}, y, p\rangle)\end{array}$$

Now we can use the induction hypothesis to replace the subexpression $(\nu s)(s{:=}[\![a]\!] \mid [\![b']\!]_q\{{}^s\!/_x\})$ with $[\![b'\{{}^a\!/_x\}]\!]_q$, and all there is left for us to do is to observe what the resulting term is the translation of:

$$\begin{aligned}(\nu q)([\![b'\{{}^a\!/_x\}]\!]_q \mid q(y).\overline{y}\langle \mathrm{l}, y, p\rangle) &= [\![b'\{{}^a\!/_x\}.\mathrm{l}]\!]_p \\ &= [\![b.\mathrm{l}\{{}^a\!/_x\}]\!]_p\end{aligned}$$

$b = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ {}^{i\in I}]$: Assume w.l.o.g. that for all $i \in I$ $x_i \neq x$. Then expanding the translation $(\nu s)(s{:=}[\![a]\!] \mid [\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ {}^{i\in I}]]\!]_p\{{}^s\!/_x\})$ we get

$$\begin{aligned}(\nu s)\Big(s{:=}[\![a]\!] \mid (\nu t)\Big(\overline{p}\langle t\rangle \mid\ !t(m,y,r).(\nu q)(\overline{q}\langle y\rangle \mid \\ \textstyle\prod_{i\in I}[m{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r\{{}^s\!/_x\})\Big)\Big)\end{aligned}$$

Now remember that we in Section 4.4 stated that if a name, for instance s, is used only for output in the terms $!s(\tilde{y}).Q$, P_1 and P_2 then we can distribute a restricted occurrence of $!s(\tilde{y}).Q$ over $P_1 \mid P_2$. This result can be strengthened to allow us also to move the replicated term beneath matching and input prefix if s does not occur in these operators. This can be used to get the following expression, that is strongly barbed congruent (using $\sim_b^{\varsigma}$) to the one above:

$$\begin{aligned}(\nu t)\Big(\overline{p}\langle t\rangle \mid\ !t(m,y,r).(\nu q)\Big(\overline{q}\langle y\rangle \mid \textstyle\prod_{i\in I}[m{=}\mathrm{l}_i]q(x_i).(\nu s)(s{:=}[\![a]\!] \mid \\ [\![b_i]\!]_r\{{}^s\!/_x\})\Big)\Big)\end{aligned}$$

We now have an expression where we can apply the induction hypothesis on all subexpressions of the form $(\nu s)(s{:=}[\![a]\!] \mid [\![b_i]\!]_r\{{}^s\!/_x\})$ getting:

$$\begin{aligned}&(\nu t)(\overline{p}\langle t\rangle \mid\ !t(m,y,r).(\nu q)(\overline{q}\langle y\rangle \mid \prod_{i\in I}[m{=}\mathrm{l}_i]q(x_i).[\![b_i\{{}^a\!/_x\}]\!]_r)) \\ = \ &[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\{{}^a\!/_x\}\ {}^{i\in I}]]\!]_p \\ = \ &[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i\ {}^{i\in I}]\{{}^a\!/_x\}]\!]_p\end{aligned}$$

$b = b'.\mathrm{l}{\Leftarrow}\varsigma(y)c$: As before, assume w.l.o.g. that $x \notin \{y,z,u\}$. The translation of $(\nu s)(s{:=}[\![a]\!] \mid [\![b'.\mathrm{l}{\Leftarrow}\varsigma(y)c]\!]_p\{{}^s\!/_x\})$ is:

$$\begin{aligned}(\nu s)\Big(s{:=}[\![a]\!] \mid (\nu q)\Big([\![b']\!]_q\{{}^s\!/_x\} \mid q(u).(\nu t)(\overline{p}\langle t\rangle \mid\ !t(m,z,r).(\nu q')(\\ \overline{q}'\langle z\rangle \mid [m{=}\mathrm{l}]q'(y).[\![c]\!]_r\{{}^s\!/_x\} \mid [m{\neq}\mathrm{l}]q'(y).\overline{u}\langle m,z,r\rangle))\Big)\Big)\end{aligned}$$

As before we can distribute $s{:=}[\![a]\!]$ resulting in the term:

$$(\nu q)\Big((\nu s)(s{:=}[\![a]\!] \mid [\![b']\!]_q\{{}^s\!/_x\}) \mid q(u).(\nu t)\Big(\overline{p}\langle t\rangle \mid !t(m,z,r).(\nu q')(\\ \overline{q}'\langle z\rangle \mid [m{=}\mathrm{l}]q'(y).(\nu s)(s{:=}[\![a]\!] \mid [\![c]\!]_r\{{}^s\!/_x\}) \mid [m{\neq}\mathrm{l}]q'(y).\overline{u}\langle m,z,r\rangle)\Big)\Big)$$

Which puts us in a position to apply the induction hypothesis and then we are done:

$$\begin{array}{rl} & (\nu q)\Big([\![b'\{{}^a\!/_x\}]\!]_q \mid q(u).(\nu t)\Big(\overline{p}\langle t\rangle \mid !t(m,z,r).(\nu q')(\\ & \quad \overline{q}'\langle z\rangle \mid [m{=}\mathrm{l}]q'(y).[\![c\{{}^a\!/_x\}]\!]_r) \mid [m{\neq}\mathrm{l}]q'(y).\overline{u}\langle m,z,r\rangle\Big)\Big) \\ = & [\![b'\{{}^a\!/_x\}.\mathrm{l}{\Leftarrow}\varsigma(y)c\{{}^a\!/_x\}]\!]_p \\ = & [\![(b'.\mathrm{l}{\Leftarrow}\varsigma(y)c)\{{}^a\!/_x\}]\!]_p \end{array}$$

□

In the above proof we use $\sim_b^\varsigma$ to distribute $s{:=}[\![a]\!]$ over parallel composition. Readers familiar with [Mil93] may recall that there is a similar law for the full π-calculus stating that

$$(\nu x)(!x(\tilde{y}).Q|P_1|P_2) \sim (\nu x)(!x(\tilde{y}).Q|P_1) \mid (\nu x)(!x(\tilde{y}).Q|P_2)$$

if free occurrences of x in P_1, P_2 and Q are *only* used in output position. The requirement on x is needed to ensure that x cannot escape its restriction because a context receiving x may be able to "impersonate" Q by receiving on x. For names obeying Lπ this is not a problem as we are not allowed to receive on received names. By inspection of the translation we see that we could not have used the law from [Mil93] because we can reveal s to the outside (for instance if a method in a returns self as result).

The next lemma deals with the relay construct that is the core of the translation of method override.

Lemma 5.2 (Relay) *Let a denote an object value $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ and let b be an arbitrary object, then the following property holds:*

$$(\nu t)(t{:=}[\![a]\!] \mid s{:=}t.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!]) \geq^\varsigma s{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}]]\!]$$

Proof. Before we start giving the proof, observe that we do *not* require j to be contained in I and therefore show that the relay construct also works in an extended version of the ς-calculus where we allow addition of methods (like in the object calculus of Fisher [Fis96]).

We show this property by exhibiting an expansion relation (up-to $\sim$) containing the pair:

$$\Big((\nu t)(t{:=}[\![a]\!] \mid s{:=}t.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!]),\ s{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b^{\ i\in I\setminus\{j\}}]]\!]\Big)$$

In principle, we have to check the property for any substitution σ but since the relation that we come up with remains invariant under substitutions this poses no problem.

It is easy to see that the relation must contain more pairs than the one mentioned. Before presenting the full relation, let us try to argue what pairs we also need to put into the relation. First of all, both processes are able to receive requests for method invocation $s\langle \mathrm{l}, s', p\rangle$ starting the processes responsible for resolving which method is to be started. For the relay construction the decision of which method should be activated takes place as a three-stage process: First we check if the name l is equal to l_j; if it is then we are done. If not, the second stage relays the message on the name t. And the third stage determines which of the methods l_i matches l. These three steps are matched in one step by the object with the new method embedded. After having resolved which method to start, say b_k, it will be started, meaning that we also need to take methods that are started into account. Finally, since we work in a concurrent setting new requests for method activation can arrive while we are still in the process of resolving others, implying that our relation should contain a scenario where several requests for method activation are being handled. This leads us to propose the following relation as our candidate:

$$\begin{aligned}
\mathcal{R} \;=\; \Big\{\Big(\;\; & (\nu\tilde{x})\Big((\nu t)(t{:=}[\![a]\!] \mid s{:=}t.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!] \mid \\
& \prod_{m\in M}(\nu q)(\overline{q}\langle s_m\rangle \mid [\mathrm{l}_m{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_m} \mid [\mathrm{l}_m{\neq}\mathrm{l}_j]\overline{t}\langle \mathrm{l}_m, s_m, r_m\rangle) \mid \\
& \prod_{n\in N}\overline{t}\langle \mathrm{l}_n, s_n, r_n\rangle) \mid \\
& \prod_{o\in O}(\nu q)(\overline{q}\langle s_o\rangle \mid \prod_{i\in I}[\mathrm{l}_o{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_o}) \mid P\Big), \\
& (\nu\tilde{x})\Big(s{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b^{\ i\in I\setminus\{j\}}]]\!] \mid \\
& \prod_{m\in M}(\nu q)(\overline{q}\langle s_m\rangle \mid \prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}_m{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_m} \mid [\mathrm{l}_m{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_m}) \mid \\
& \prod_{n\in N}(\nu q)(\overline{q}\langle s_n\rangle \mid \prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}_n{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_n} \mid [\mathrm{l}_n{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_n})
\end{aligned}$$

$$\prod_{o\in O}(\nu q)(\overline{q}\langle s_o\rangle \mid \prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}_o{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_o} \mid [\mathrm{l}_o{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_o}) \mid P\Big)\ \Big)$$
$$\Big|\quad P\in\mathbf{Proc},\ \mathrm{l}_m\in\mathbf{MName},\ \mathrm{l}_n,\mathrm{l}_o\in\mathbf{MName}\setminus\{\mathrm{l}_j\}\Big\}$$

In $\mathcal{R}$ the processes ranged over by M represent the first part of the sequence of actions performed by the relay construct to decide which method to activate, N represent the second and O the third. The process P denotes already activated methods and the restriction $(\nu\tilde{x})$ needs to be there because activated methods may activate other methods recursively using a private name as reply channel.

Now we "only" need to go through the tedious process of showing that $\mathcal{R}$ is in fact an expansion relation up to $\sim$. To make the discussion a little bit easier we shall use the following abbreviations:

$$\begin{aligned}
S_1 &\triangleq t{:=}[\![a]\!] \mid s{:=}t.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!]\\
S_2^M &\triangleq \prod_{m\in M}(\nu q)(\overline{q}\langle s_m\rangle \mid [\mathrm{l}_m{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_m} \mid [\mathrm{l}_m{\neq}\mathrm{l}_j]q(x).\bar{t}\langle \mathrm{l}_m, s_m, r_m\rangle))\\
S_3^N &\triangleq \prod_{m\in N}\bar{t}\langle \mathrm{l}_n, s_n, r_n\rangle\\
S_4^O &\triangleq \prod_{o\in O}(\nu q)(\overline{q}\langle s_o\rangle \mid \prod_{i\in I}[\mathrm{l}_o{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_o})\\
T_1 &\triangleq s{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}]]\!]\\
T_2^W &\triangleq \prod_{w\in W}(\nu q)(\overline{q}\langle s_w\rangle \mid \prod_{i\in I\setminus\{j\}}[\mathrm{l}_w{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_{r_w} \mid [\mathrm{l}_w{=}\mathrm{l}_j]q(x).[\![b]\!]_{r_w})\\
&\qquad \text{for } W\in\{M,N,O\}
\end{aligned}$$

With these abbreviations we can write $\mathcal{R}$ as

$$\Big\{\Big((\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^M \mid S_3^N) \mid S_4^O \mid P\Big), (\nu\tilde{x})\Big(T_1 \mid T_2^M \mid T_2^N \mid T_2^O \mid P\Big)\Big)\Big\}$$

We now proceed with a case analysis of the possible transitions

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^M \mid S_3^N) \mid S_4^O \mid P\Big) \xrightarrow{\alpha} S$$

by the left element of the pair. We must be able to find a matching transition

$$(\nu\tilde{x})\Big(T_1 \mid T_2^M \mid T_3^N \mid T_4^O \mid P\Big) \xrightarrow{\widehat{\alpha}} T$$

where $S \sim\mathcal{R}\sim T$. The α-transition may have originated from:

An observable action by P: That is $P \xrightarrow{\alpha} P'$. This transition can be performed by both elements of the pair and the resulting pair is also contained in $\mathcal{R}$.

An observable action by S_1: This can only be an input action $s\langle \mathrm{l}_k, s_k, p_k\rangle$ resulting in the addition of an element to S_2. Therefore the resulting system is of the form:

$$S = (\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\cup\{k\}} \mid S_3^N) \mid S_4^O \mid P\Big)$$

This transition is easily matched by the right hand system by doing exactly the same transition resulting in

$$T = (\nu\tilde{x})\Big(T_1 \mid T_2^{M\cup\{k\}} \mid T_2^N \mid T_2^O \mid P\Big)$$

And the resulting pair is related by $\mathcal{R}$.

An internal communication between P and S_1: For this to happen P must have performed the transition $P \xrightarrow{\alpha} P'$ with $\alpha = (\nu\tilde{y})\overline{s}\langle \mathrm{l}', s', p'\rangle$ resulting in the system

$$(\nu\tilde{x}\tilde{y})((\nu t)(S_1 \mid S_2^{M\cup\{m'\}} \mid S_3^N) \mid S_4^O \mid P')$$

where $\mathrm{l}_{m'} = \mathrm{l}'$, $s_{m'} = s'$ and $r_{m'} = p'$. Such an internal action is easily matched by the right hand system resulting in the system

$$(\nu\tilde{x}\tilde{y})(T_1 \mid T_2^{M\cup\{m'\}} \mid T_2^N \mid T_3^O \mid P')$$

and it is easy to see that the pair consisting of these two systems is contained in $\mathcal{R}$.

An internal action by S_2^M: This can only happen by an internal action by one of S_2^M's components, say m'. The transition (omitting the index) must be of the form $(\nu q)(\overline{q}\langle s\rangle \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_p \mid [\mathrm{l}{\neq}\mathrm{l}_j]q(x).\bar{t}\langle \mathrm{l}, s, p\rangle) \xrightarrow{\tau} Q$. There are two possible internal transitions depending on whether l is equal to l_j or not.

If l is equal to l_j then $Q = (\nu q)([\mathrm{l}{\neq}\mathrm{l}_j]q(x).\bar{t}\langle \mathrm{l}_j, s, p\rangle) \mid [\![b]\!]_p\{s/x\}$. The part $(\nu q)([\mathrm{l}{\neq}\mathrm{l}_j]q(x).\bar{t}\langle \mathrm{l}_j, s, p\rangle)$ cannot do any actions, therefore S must be strongly congruent to

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^N) \mid S_4^O \mid P \mid [\![b]\!]_p\{s/x\}\Big)$$

By construction, there must be a component of T_2^M of the form $(\nu q)(\overline{q}\langle s\rangle \mid \prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_p \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_p)$. Again, since l is equal to l_j this process can do an internal transition to a process Q' of the form:

$$(\nu q)(\prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_p) \mid [\![b]\!]_p\{s/x\}$$

And again the part $(\nu q)(\prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_p)$ is dead so T must be strongly congruent to

$$(\nu\tilde{x})\Big(T_1 \mid T_2^{M\setminus\{m'\}} \mid T_2^N \mid T_2^O \mid P \mid [\![b]\!]_p\{s/x\}\Big)$$

So we have $S \sim\mathcal{R}\sim T$

If l is not equal to l_j then $Q = \overline{t}\langle \mathrm{l}, s, p\rangle$. This means that Q now fits the shape of S_3 so we have

$$S = (\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^{N\cup\{m'\}}) \mid S_4^O \mid P\Big)$$

This transition can be matched by the corresponding process in T_2^M by doing nothing (but now we know that l is not l_j so we can now move the process to T_3^N getting

$$T = (\nu\tilde{x})\Big(T_1 \mid T_2^{M\setminus\{m'\}} \mid T_3^{N\cup\{m'\}} \mid T_4^O \mid P\Big)$$

And now it is easy to see that $S\ \mathcal{R}\ T$.

An internal communication between S_1 and S_3^N: This is caused by the consumption by the replicated process located at t of a message $\overline{t}\langle \mathrm{l}_{n'}, s_{n'}, r_{n'}\rangle$ for $n' \in N$. Such a communication will spawn off a process of the shape that fits S_4, resulting in the total system:

$$S = (\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^M \mid S_3^{N\setminus\{n'\}}) \mid S_4^{O\cup\{n'\}} \mid P\Big)$$

This can again be matched by the right hand system resulting in two related systems by doing nothing, all we need to do is to rearrange the index sets to:

$$T = (\nu\tilde{x})\Big(T_1 \mid T_2^M \mid T_2^{N\setminus\{n'\}} \mid T_2^{O\cup\{n'\}} \mid P\Big)$$

An internal action by S_4^O: This case follows the same line of reasoning as for S_2^M. The action must be caused by a one of S_4's components, say o', doing the transition (again we omit the index o):

$$(\nu q)(\overline{q}\langle s\rangle \mid \prod_{i\in I} q(x_i).[\![b_i]\!]_r) \xrightarrow{\tau} (\nu q)(\prod_{i\in I\setminus\{k\}} q(x_i).[\![b_i]\!]_r) \mid [\![b_k]\!]_r\{s/x_k\}$$

This again means that the total system S is strongly congruent to:

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^M \mid S_3^N) \mid S_4^{O\setminus\{o'\}} \mid P \mid [\![b_k]\!]_r\{s/x\}\Big)$$

By assumption we know that k cannot be j. This implies that the similar component in T_2^O can do a matching transition:

$$\begin{array}{l}(\nu q)(\overline{q}\langle s\rangle \mid \textstyle\prod_{i\in I\setminus\{\mathrm{l}_j\}}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_r) \xrightarrow{\tau} \\ (\nu q)(\textstyle\prod_{i\in I\setminus\{j,k\}}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_r) \mid [\![b_k]\!]_r\{s/x_k\}\end{array}$$

Which gives us a total system T that is strongly congruent to

$$(\nu\tilde{x})\Big(T_1 \mid T_2^{M\setminus\{m'\}} \mid T_3^N \mid T_4^O \mid P \mid [\![b]\!]_r\{s/x\}\Big)$$

So we have $S \sim\mathcal{R}\sim T$.

This ends the first part of the analysis. We now look at how the left hand system can match a transition

$$(\nu\tilde{x})\Big(T_1 \mid T_2^M \mid T_3^N \mid T_4^O \mid P\Big) \xrightarrow{\alpha} T$$

performed by the right hand system by doing a weak transition $\stackrel{\alpha}{\Longrightarrow}$ resulting in a system S. Again we must proceed by a case analysis of the origin of the transition:

An observable action by P: Same argument as in the other direction.

An observable action by T_1: Same argument as before will apply.

An internal transition because P synchronize with T_1: Same argument as before.

An internal action by T_2^W: This is the most interesting case. We have three cases depending on whether W is M, N or O. The most involved case is if $W = M$ so let us restrict ourselves to that case only.

If $T_2^M \xrightarrow{\tau} T'$ then it is because one of the components, labelled m' in T_2^M has performed the transition. That is, the action is performed by some process of the form (as usual we omit the index):

$$(\nu q)\Big(\overline{q}\langle s\rangle \mid \prod_{i\in I\setminus\{j\}} [\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_r\Big)$$

The action can only have occurred by a communication on the private name q implying that l is equal to some l_i or l_j. We need to divide the analysis of the transition into two cases depending on which of the two possibilities applies.

If l is equal to l_j, the result of the transition is:

$$(\nu q)\Big(\prod_{i\in I\setminus\{j\}} [\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r\Big) \mid [\![b]\!]_r\{s/x\}$$

which makes the total system T congruent to

$$(\nu\tilde{x})\Big(T_1 \mid T_2^{M\setminus\{m'\}} \mid T_3^N \mid T_4^O \mid P \mid [\![b]\!]_r\{s/x\}\Big)$$

The transition by T_2^M can be matched by the component S_2^M by a similar internal action giving a total system S congruent to:

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^N) \mid S_4^O \mid P \mid [\![b]\!]_r\{s/x\}\Big)$$

And we get $S \sim\mathcal{R}\sim T$.

If l is not equal to l_j, then l must be equal to some l_k for $k \in I \setminus \{j\}$. This means that the result of the transition is:

$$(\nu q)\Big(\prod_{i\in I\setminus\{j,k\}} [\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r \mid [\mathrm{l}{=}\mathrm{l}_j]q(x).[\![b]\!]_p\Big) \mid [\![b_k]\!]_r\{s/x_k\}$$

which makes the total system T congruent to

$$(\nu\tilde{x})\Big(T_1 \mid T_2^{M\setminus\{m'\}} \mid T_3^N \mid T_4^O \mid P \mid [\![b_k]\!]_r\{s/x_k\}\Big)$$

In order to match this the left hand system need to do three internal actions. The first internal action is performed by the element in S_2^M that has the

index m' resulting in the the removal of that part from S_2 and the addition of it to S_3. Giving us a total system in the form:

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^{N\cup\{m'\}}) \mid S_4^O \mid P\Big)$$

The new component in S_3 is able to communicate with S_1 spawning of a new element in S_4 so that we get:

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^{N}) \mid S_4^{O\cup\{m'\}} \mid P\Big)$$

The new element in S_4 looks like:

$$(\nu q)\Big(\overline{q}\langle s\rangle \mid \prod_{i\in I}[\mathrm{l}{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_r\Big)$$

We know that l is equal to l_k so an internal transition can again happen giving us:

$$(\nu q)\Big(\prod_{i\in I\setminus\{k\}}[\mathrm{l}{=}\mathrm{l}_i].q(x_i).[\![b_i]\!]_r\Big) \mid [\![b_k]\!]_r\{s/x\}$$

This means that we have a system S congruent to:

$$(\nu\tilde{x})\Big((\nu t)(S_1 \mid S_2^{M\setminus\{m'\}} \mid S_3^{N}) \mid S_4^O \mid P \mid [\![b_k]\!]_r\{s/x\}\Big)$$

And we get $S \sim\mathcal{R}\sim T$.

The reasoning for the two other cases follows a similar pattern except that we need fewer transitions by the left hand system in order to match the transition by the right hand system.

This concludes the proof. We have been very thorough this time because the translation of method override is the most complicated part of the translation, as has been witnessed by the problems we have had with getting to the final formulation. In subsequent proofs where we create bisimulation relations, we shall not be as thorough in the above case analysis, but instead just discuss some of the interesting cases. □

Using the two lemmas one immediately get two easy corollaries, stating that method activation and method override work correctly.

Corollary 5.3 *Let* $a = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$. *Then for all* $j \in I$

$$[\![a.\mathrm{l}_j]\!]_p \xrightarrow{\tau} \geq_b^{\varsigma} [\![b_j\{a/x_j\}]\!]_p$$

PROOF. We simply need to observe that $[\![a.\mathrm{l}_j]\!]_p$ can do a series of deterministic τ-steps resulting from communication on private names leading to a term where we can apply Lemma 5.1.

We have that

$$\begin{array}{rcl}
[\![a.\mathrm{l}_j]\!]_p & = & (\nu q)((\nu s)(\overline{q}\langle s\rangle \mid s{:=}[\![a]\!]) \mid q(x).\overline{x}\langle \mathrm{l}_j, x, p\rangle) \\
& \xrightarrow{\tau}_d & (\nu s)(s{:=}[\![a]\!] \mid \overline{s}\langle \mathrm{l}_j, s, p\rangle) \\
& \xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![a]\!] \mid (\nu q)(\overline{q}\langle s\rangle \mid \prod_{i\in I}[\mathrm{l}_j{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_p)\Big) \\
& \xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![a]\!] \mid (\nu q)(\prod_{i\in I\setminus\{j\}}[\mathrm{l}_j{=}\mathrm{l}_i]q(x_i).[\![b_i]\!]_p) \mid [\![b_j]\!]_p\{{}^{s}/_{x_j}\}\Big) \\
& \sim & (\nu s)\Big(s{:=}[\![a]\!] \mid [\![b_j]\!]_p\{{}^{s}/_{x_j}\}\Big) \\
& \sim_b^{\varsigma} & ([\![b_j\{{}^{a}/_{x_j}\}]\!]_p)
\end{array}$$

□

Corollary 5.4 *Let* $a = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$. *Then*

$$[\![a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!]_p \xrightarrow{\tau} \geq^{\varsigma} [\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}]]\!]_p$$

Again observe that this property in fact holds for a calculus where we allow the addition of methods.

PROOF. As in the previous case all we need to do is to observe that the left hand process can do a series of internal communications to become a process where Lemma 5.2 applies:

$$\begin{array}{rcl}
[\![a.\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!]_p & = & (\nu q)\Big((\nu s)(\overline{q}\langle s\rangle \mid s{:=}[\![a]\!]) \mid q(y). \\
& & \qquad (\nu t)(\overline{p}\langle t\rangle \mid t{:=}y.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!])\Big) \\
& \xrightarrow{\tau}_d & (\nu s)\Big(s{:=}[\![a]\!] \mid (\nu t)(\overline{p}\langle t\rangle \mid t{:=}s.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!])\Big) \\
& \sim & (\nu t)\Big(\overline{p}\langle t\rangle \mid (\nu s)(s{:=}[\![a]\!] \mid t{:=}s.[\![\mathrm{l}_j{\Leftarrow}\varsigma(x)b]\!])\Big) \\
& \geq^{\varsigma} & (\nu t)(\overline{p}\langle t\rangle \mid t{:=}[\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}]]\!]) \\
& = & [\![[\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)b\ ^{i\in I\setminus\{j\}}]]\!]_p
\end{array}$$

□

With these two corollaries we can now prove the following result, which states that whenever an object a can reduce to the object b, then the translation of b is weakly congruent to a.

Theorem 5.5 *If* $a \rightsquigarrow b$ *then* $[\![a]\!]_p \xrightarrow{\tau} \approx_b^\varsigma [\![b]\!]_p$

Proof. We prove this property by induction on the structure of the inference of the reduction $a \rightsquigarrow b$.

We have two base cases. Let $c = [\mathrm{l}_i{=}\varsigma(x_i)b_i \ ^{i\in I}]$. Either $a = c.\mathrm{l}_j$ and $b = b_j\{c/x_j\}$ and the result follows from Corollary 5.3, or $a = c.\mathrm{l}_j{\Leftarrow}\varsigma(x)c'$ and $b = [\mathrm{l}_i{=}\varsigma(x_i)b_i,\ \mathrm{l}_j{=}\varsigma(x)c' \ ^{i\in I\setminus\{j\}}]$ and the result follows from Corollary 5.4.

For the inductive case we have $a = C[a']$ and $b = C[b']$ and use the inference rule

$$\frac{a' \rightsquigarrow b'}{C[a'] \rightsquigarrow C[b']}$$

Since the translation is compositional there exists a π-calculus context $\mathcal{C}[\cdot]$ such that $[\![C[a']]\!]_p = \mathcal{C}[[\![a']\!]_q]$ and $[\![C[b']]\!]_p = \mathcal{C}[[\![b']\!]_q]$. (In fact we also know what $\mathcal{C}[\cdot]$ looks like, it is simply the translation of the ς-calculus context $C[\cdot]$.)

By induction we have that if $a' \rightsquigarrow b'$ then $[\![a']\!]_q \approx_b^\varsigma [\![b']\!]_q$. And since $\approx_b^\varsigma$ is a congruence we get

$$a = [\![C[a']]\!] = \mathcal{C}[[\![a']\!]_q] \approx_b^\varsigma \mathcal{C}[[\![b']\!]_q] = [\![C[b']]\!] = b$$

□

The relation from transitions in the π-calculus encoding to reductions in the ς-calculus is somewhat more difficult to express. This is because the π-calculus encoding may need to perform some internal computation before being ready to simulate an object.

The desired property we initially stated was that $[\![a]\!]_p \approx [\![b]\!]_p$ then $a \rightsquigarrow^* b$. But there are several reasons why this property does not hold. First of all, $[\![a]\!]_p \approx [\![b]\!]_p$ says nothing about $[\![a]\!]_p$ "evaluating" to $[\![b]\!]_p$ — the reverse might in fact be the case. Furthermore, we can come up with ς-calculus terms that have bisimilar translations but where one cannot reduce one to the other (for instance terms that are stuck, as the term [].l).

One might then consider requiring that the translation of a should perform some reductions in order to become a process weakly congruent to the translation of b as in: If $[\![a]\!]_p \xrightarrow{\tau}{}^* P$ then $a \rightsquigarrow b$ for some b where $P \approx [\![b]\!]_p$. (We do not use $\rightsquigarrow^*$ because we would like to express that a *can* reduce.) But such a property does not hold because the translation of a ς-calculus term might need to do some internal transitions before getting stuck, as for instance in the translation of $[\,].l$. So we need some means of telling whether the reductions (internal transitions) performed by the translation are "real" reductions caused by similar reductions in the ς-calculus term. The most

natural way to achieve this is to use the observable actions that the translation of a term performs to signal that it has evaluated to a value. This leads to the formulation of the following theorem:

Theorem 5.6 *If* $[\![a]\!]_p \Downarrow_p$ *then* $a \rightsquigarrow^* v$ *where* $v = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ *or* $v = x$ *and* $[\![a]\!]_p \approx_b^\varsigma [\![v]\!]_p$

Proof. Before commencing on the proof, a remark as to why we require an observable action on p and not just any observable action. This is because free occurrences of self variables in ς-calculus terms also can result in observable actions, as in the translation of the term $x.l$. By restricting ourselves to observable actions on p we avoid mistaking such actions as signalling that a term has reduced to a value. (Another possibility would have been only to consider closed terms.)

The proof is by induction on the number of internal transitions performed by $[\![a]\!]_p$ before an action can be observed at the name p. That is we do induction on n in $[\![a]\!]_p \xrightarrow{\tau}{}^n P{\downarrow_p}$ using expansion instead of weak congruence. The reason why we use expansion (and also why the previous lemmas and corollaries were stated using expansion) it that if $P \leq_b^\varsigma Q$ then we know that P will have fewer internal transitions than Q. This will allow us to use the induction hypothesis on a part of a system and still have an upper bound on the number of internal transitions that the total system can perform.

Base case $n = 0$**:** By inspection of the translation we can easily see that there are only two terms whose translations are immediately capable of doing an observable action on p, namely if a is either an object value $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ or a self variable x. In both cases, since a is a value, we have $a \rightsquigarrow^* v = a$ and since $\leq_b^\varsigma$ is reflexive we have $[\![a]\!]_p \leq_b^\varsigma [\![a]\!]_p$.

Inductive case $n > 0$**:** Assume the theorem holds for all τ-sequences of length less than n. We now consider a reduction sequence leading to an observable action on p $[\![a]\!]_p \xrightarrow{\tau}{}^n P$ of length n. We now proceed with a case analysis on the structure of a:

$a = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$**:** This term can immediately perform an observable action and since we assumed $n > 0$, this term does not fit this case.

$a = x$**:** Same argument as above.

$a = b.\mathrm{l}$**:** The translation of a is $(\nu q)([\![b]\!]_q \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle)$ where $p \notin \mathsf{fn}([\![b]\!]_q)$. Since p does not occur free in $[\![b]\!]_q$ the only way we can get an observable action on p is if $[\![b]\!]_q$ performs a number of reductions followed by an

output of a name s on q, an interaction on the name s and then again a series of reductions leading to the observable actions on p. That is, first we have

$$(\nu q)(\llbracket b \rrbracket_q \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle) \xrightarrow{\tau}{}^{m} (\nu q)(P \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle)$$

where $P{\downarrow}q$. Obviously we must have $m < n$, for otherwise $\llbracket b.\mathrm{l} \rrbracket$ would not be able to perform an observable action on p after n reductions. Therefore we can use the induction hypothesis on $\llbracket b \rrbracket_q$ and get that for some value v $b \rightsquigarrow^* v$ and $\llbracket b \rrbracket_q \geq_b^{\varsigma} \llbracket v \rrbracket_q$.

Since $(\nu q)(P \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle){\Downarrow_p}$, we have $(\nu q)(\llbracket v \rrbracket_q \mid q(x).\overline{x}\langle \mathrm{l}, x, p\rangle){\Downarrow_p}$. The only way the last term can perform an observable action on p is if v is an object value $[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in I}]$ and l is equal to some l_j for $j \in I$. The term $(\nu q)(\llbracket v \rrbracket_q \mid q(x).\overline{x}\langle \mathrm{l}_j, x, p\rangle)$ will do two deterministic internal transitions (see the proof of Corollary 5.3) to become a term strongly congruent to $\llbracket b_j\{{}^{v}/_{x_j}\} \rrbracket_p$. This means that the number of internal transitions that $\llbracket b_j\{{}^{v}/_{x_j}\} \rrbracket_p$ can do before an observable action on p is definitely less than n. So we can apply the induction hypothesis again to infer that $\llbracket b_j\{{}^{v}/_{x_j}\} \rrbracket_p{\Downarrow_p}$, $b_j\{{}^{v}/_{x_j}\} \rightsquigarrow v'$ and $\llbracket b_j\{{}^{v}/_{x_j}\} \rrbracket_p \geq_b^{\varsigma} v'$ for some value v'. By transitivity of $\geq_b^{\varsigma}$ we then get that $\llbracket b.\mathrm{l} \rrbracket_p \geq_b^{\varsigma} \llbracket v' \rrbracket_p$.

$a = b.\mathrm{l}{\Leftarrow}\varsigma(x)c$**:** This case is handled with the same argument as in the previous case (although simpler as we only need to apply the induction hypothesis once).

□

Finally from Theorem 5.5 and Theorem 5.6 we get the following result for closed terms stating that a computation of a term a terminates if and only if the corresponding computation for the translation of a does so.

Theorem 5.7 (Computational adequacy) *Let a be a closed ς-calculus term. Then $a{\Downarrow}$ iff $\llbracket a \rrbracket_p{\Downarrow_p}$.*

Since for closed terms, that is terms a where $\mathsf{fn}(\llbracket a \rrbracket_p) \subseteq \mathbf{MName} \cup \{p\}$, it is in fact enough to detect any observable action by the translation of a term a in order to infer that $a{\Downarrow}$ (because the only action that can happen is on p).

We have now finished showing that our translation is operationally correct. In the course of doing this we worked in a restricted version of the π-calculus and adapted a more liberal semantics for the untyped ς-calculus.

The use of a restricted π-calculus in the proofs is mostly for convenience; the main result in Theorem 5.7 does not really depend on the use of a restricted π-calculus simply because the translation from the ς-calculus to the

π-calculus obeys these constraints. On the other hand, had we not worked in this restricted version of the π-calculus the proofs would have been much more complicated primarily because Lemma 5.1, the substitution lemma, does not hold in the full π-calculus.

The use of a more liberal ς-calculus semantics, where we allow addition of methods to objects, is more annoying, since one can claim that the translation in *not sound* w.r.t. the real ς-calculus semantics. But on the other hand, the main reason why Abadi and Cardelli chose not to allow addition of methods was to simplify the type system (c.f. [AC96, Section 6.1.4]). So in an untyped setting we feel that using the more liberal semantics is not unreasonable. Furthermore, if we only consider well-typed ς-calculus terms, the translation is in fact faithful to the ς-calculus semantics from Table 3.2 since a well-typed program never attempts to add a method to an object.

5.4 Reasoning about the ς-calculus

The topic of the previous section was the operational correctness of the translation. Operational correctness can be thought of as saying that "the translation compiles ς-calculus programs correctly into π-calculus machine code". So these results tell us little about how we might consider using the translation to reason about ς-calculus terms. In this section we shall work on this by establishing the basic properties needed to use the translation for reasoning about ς-calculus terms.

So what are the basic properties we want from the translation? One possibility would be to consider the untyped equational laws of the ς-calculus ([AC96, page 63]) and check their validity w.r.t. some π-calculus equivalence. But as most of these laws are quite trivial and stated mostly to introduce the more interesting typed ones, we feel that it is more interesting to consider how a notion of behavioral equivalence for ς-calculus terms relates to π-calculus equivalences.

Notions of program equivalence are, as mentioned earlier, central to the theory and practice of programming languages. They form the basis for program optimization and can be used to justify correctness preserving transformations performed by program manipulation systems. Program equivalences are typically defined according to the following paradigm:

i. A collection of terms that are considered to be directly executable and observable are designated as programs, and their behavior is defined.

ii. Two arbitrary terms are defined to be equivalent iff they have the same behavior in every program context.

The resulting notion of program equivalence is usually referred to as *observational congruence* [Mor68]. Observational congruence for the first-order object calculus with subtyping $\mathbf{Ob}_{1<:\mu}$ has been defined by Gordon and Rees in [GR96] thus: Two programs are observationally congruent iff they have the same termination behavior in all contexts of boolean type. We shall have a closer look at Gordon and Rees' definition of typed observational congruence in Chapter 7. Since we are currently working in an untyped setting we cannot restrict the set of contexts but shall instead consider two terms equivalent iff they have the same termination behavior in all contexts. This leads to the following definition:

Definition 5.8 (Untyped observational congruence) *Let a and b be two closed ς-calculus terms. We say that a and b are observationally congruent, written $a \simeq b$, if for all closed contexts $C[\cdot]$:*

- *$C[a]\Downarrow$ iff $C[b]\Downarrow$.*

Following Gordon and Rees, we have only defined observational congruence on closed terms. The definition can easily be extended to apply also to closed terms simply by considering contexts that close the terms a and b.

Being in an untyped setting implies that contexts are very powerful since we have no way of restricting the set of contexts that we test the two terms in. One must also be aware that we examine objects in contexts that might result in computations leading to a stuck state.

Also our use of the more liberal semantics adds to the power of contexts. For instance the two terms

$$a = [\mathrm{l}{=}\varsigma(x)x] \qquad b = [\mathrm{l}{=}\varsigma(x)a]$$

are observationally congruent if we do not allow addition of methods to objects (b is just the one level unfolding of a). But with addition of methods we can easily distinguish the two terms using the context $C[\cdot] = ([\cdot].\mathrm{l}' \Leftarrow \varsigma(x)\Omega).\mathrm{l}.\mathrm{l}'$, where Ω denotes the divergent term $[\mathrm{l}{=}\varsigma(x)x.\mathrm{l}].\mathrm{l}$.

Now that we have a definition of equivalence on ς-calculus terms, we proceed to study how this definition relates to definitions of equivalence in the π-calculus. The main result is the following:

Theorem 5.9 (Adequacy) *Assume that a and b are closed, then $[\![a]\!]_p \approx_b^\varsigma [\![b]\!]_p$ implies $a \simeq b$.*

Proof. We must show that if $[\![a]\!]_p \approx_b^\varsigma [\![b]\!]_p$ then for all ς-calculus contexts $C[\cdot]$ we have $C[a]\Downarrow$ iff $C[b]\Downarrow$.

Assume that $C[a]\Downarrow$, we must now argue that $C[b]\Downarrow$. By Theorem 5.7 if $C[a]\Downarrow$ then $[\![C[a]]\!]_q\Downarrow_q$. Because $\approx_b^\varsigma$ is a congruence, we have $\mathcal{C}[[\![a]\!]_p] \approx_b^\varsigma \mathcal{C}[[\![b]\!]_p]$ for all π-calculus contexts $\mathcal{C}[\cdot]$, and in particular for the contexts that are encodings of ς-calculus contexts. Furthermore by the definition of barbed bisimulation, if $[\![C[a]]\!]_q \approx_b^\varsigma [\![C[b]]\!]_q$ and $[\![C[a]]\!]_q\Downarrow_q$ then $[\![C[b]]\!]_q\Downarrow_q$. Finally we can now use Theorem 5.7 again to infer that $C[b]\Downarrow$.

A similar argument is used to infer that if $C[b]\Downarrow$ then $C[a]\Downarrow$. □

This result allows us to use the translation into the π-calculus to infer that two ς-calculus terms are equivalent. We have shown the result for $\approx_b^\varsigma$, but to establish that two ς-calculus terms are observationally congruent any π-calculus congruence will do, as long as it is contained in $\approx_b^\varsigma$.

The ultimate goal would be to use the translation to establish all equalities between ς-calculus terms. Unfortunately, this is not the case as shown by the following two objects:

$$a = [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}] \qquad b = [\mathrm{l}{=}\varsigma(x)a.\mathrm{l}]$$

Obviously no ς-calculus context can tell them apart — on activation of the method l both objects diverge, on override of the method l both object evaluate to the same term, and if we add a method the only thing this method can do to experiment with l is either to activate or override it. But we can come up with a π-calculus context that is capable of telling a and b apart. The trick is to use a modified relay construct that will allow us to see which method activations happen internally in a and b, as in the following π-calculus context.

$$\begin{aligned}\mathcal{C}[\cdot] \;=\; (\nu q)\Big(&(\nu p)([\cdot] \mid p(x).(\nu t)(\overline{q}\langle t\rangle \mid t(m,y,r).(\overline{x}\langle m,y,r\rangle \mid \\ &t(m,y,r).\overline{r}\langle t\rangle))) \mid q(x).\overline{x}\langle \mathrm{l},x,r\rangle\Big)\end{aligned}$$

The context $\mathcal{C}[\cdot]$ is built with a special relay construct inside a request for the activation of the method named l. The relay construct is designed such that it will only forward *one* request for method activation to the object placed in the hole; if a second request arrives it will immediately signal this on the name r. This means that the term $\mathcal{C}[[\![a]\!]_p]\Downarrow_r$ whereas $\mathcal{C}[[\![b]\!]_p] \not\Downarrow_r$.

The example above shows that we are able to detect which methods an object can activate internally, therefore even $\approx_b^\varsigma$ is quite discriminating on ς-calculus terms. Unfortunately there is not much to do about this since the ability to detect which methods an objects tries to activate internally is needed in order to describe method override in our translation.

This result, that $\approx_b^{\varsigma}$ is not complete w.r.t. $\simeq$, does not come as a surprise. In fact it would have been very strange if observational congruence between ς-calculus terms implied congruence between the translated terms for some natural π-calculus congruence. The above example does in fact show that it is not even enough to restrict the set of π-calculus contexts to deterministic contexts.

What we can obtain, is a not very interesting result, namely that if we restrict the set of π-calculus contexts to contexts which are translations of ς-calculus contexts, then we do in fact get the reverse result of Theorem 5.9.

Theorem 5.10 *Let $\approx_b^{[\![]\!]}$ denote barbed congruence over the set of translations of ς-calculus contexts and assume that a and b are closed. Then $a \simeq b$ iff $[\![a]\!]_p \approx_b^{[\![]\!]} [\![b]\!]_p$.*

Proof. The proof is along the lines of the proof of Theorem 5.9. □

Of course such a result is not particularly useful, since we have no alternative characterization of $\approx_b^{[\![]\!]}$ but it seems quite difficult to come up with such a characterization.

5.5 Final remarks

In this chapter, we have examined a translation of the *untyped* Functional ς-calculus into the π-calculus. This chapter is a major rewrite of an unpublished manuscript and a BRICS technical report [HK96]. The unpublished manuscript that contained the translation using matching was submitted to FOOL 3 (workshop on the Foundations of Object-Oriented Languages) which was held in conjunction with LICS'96, but was not accepted because it had a large overlap with an invited talk by Davide Sangiorgi on the same subject (we shall discuss his work in the following chapter). The technical report examines the translation that does not use matching; unfortunately, that paper contains some errors, which were pointed out to us by the anonymous referees at CJTCS. The problems are the ones discussed on page 65 in the list of possible solutions to what method labels one needs to relay.

Although perhaps interesting in its own right, the translation presented in this chapter does not really help us to answer the question about the usefulness of process calculus techniques to reason about objects. The two main problems are related to our source calculus — it is *untyped* and *functional*. As we have mentioned several times, type systems are quite important in object-orientation, and most object-oriented languages are imperative, not functional. A translation incorporating these aspects will be the topic of the following chapter.

Chapter 6

Translating a Typed Imperative ς-calculus

In the previous chapter we looked at a translation of the *untyped Functional ς-calculus* into the π-calculus. The stateless functional semantics for the ς-calculus is the semantics normally used when studying the ς-calculus. The motivation given by Abadi and Cardelli for focusing on the functional semantics is that the simplicity of its semantics make the study of type systems easier.

However, most "real world" programming languages are imperative. Usually, objects encapsulate a state, which can be manipulated by activating the methods of the object; that is, method calls can have side effects on the state of the object. In order to show that the type systems developed for the Functional ς-calculus are also applicable to imperative languages, Abadi and Cardelli give an imperative semantics to the ς-calculus and show how the type systems can be adapted to it.

Since most object-oriented languages are imperative, it seems highly relevant to consider a translation of the Imperative ς-calculus, especially, as the lack of mathematical techniques for giving the semantics to, and proving properties about, imperative object-oriented languages is even more pronounced than for functional object-oriented languages. For instance, it seems difficult to come up with reasonable notions of bisimulation for the Imperative ς-calculus. This contrasts with the Functional ς-calculus, for which one such a notion has been developed (see Chapter 7).

The study of the Imperative ς-calculus can provide a solid basis for investigating more complex languages, that may include also, for instance, constructs for distribution and concurrency (like Obliq [Car95], that, indeed, contains the untyped Imperative ς-calculus as a sublanguage).

As mentioned several times, most interesting problems/challenges specific to object-oriented languages are related to types. Therefore, it is relevant to take typing issues into account in the translation. So, in this chapter, we shall study the interpretation of the (first order) Imperative ς-calculus within a *typed π-calculus*.

The typed π-calculus we use is the one Sangiorgi introduced in [San98] in order to give a typed translation of the Functional ς-calculus with type system $\mathbf{Ob}_{1<:}$ (see Section 6.6 for a description of Sangiorgi's translation of the Functional ς-calculus). This π-calculus extends the polyadic π-calculus with variants and variant types. Variant types, together with input/output capabilities, allow very useful subtyping rules, that turn out to be essential in order to give a typed translation of the ς-calculus.

We end this introduction with a brief road-map of the contents of the chapter. Section 6.1 introduces the syntax, semantics, and type system of the Imperative ς-calculus. Section 6.2 is devoted to the typed π-calculus; we present its syntax, semantics, and type system. The translation of the Imperative ς-calculus into the π-calculus is given in Section 6.3. In Section 6.4, we give an alternative encoding that simplifies the proof of operational correctness. In Section 6.5.1 we show the correctness of the translation. First we show the correctness with respect to typing and subtyping for the original encoding. Then we establish operational correctness for the alternative encoding and finally we show operational correctness for the original encoding. In Section 6.6 we briefly discuss how our encoding is related to Sangiorgi's encoding of the Functional ς-calculus into the π-calculus from [San98]. Finally, Section 6.7 shows how the translation of the Imperative ς-calculus to the π-calculus can be used to justify program transformations rules; we give examples of untyped transformations and transformations that are only valid in a typed setting.

6.1 The Imperative ς-calculus

In this section we present syntax, semantics and type system of the (first-order) Imperative ς-calculus from [AC96, Chapter 10-11].

6.1.1 Syntax

The syntax of the Imperative ς-calculus, given by the grammar in Table 6.1, is not much different from the syntax of the Functional ς-calculus found in Table 3.1; the only difference is the addition of a clone- and a let-construct. Cloning creates a copy of the original object. A $\mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b$ expression first

evaluates the let-part, binding the result to x, then the in-part is evaluated with the variable x in scope. Sequential evaluation of objects $a; b$ can be defined thus: let $x{:}A{=}a$ in b for some $x \notin \mathsf{fv}(b)$. The informal semantics for the rest of the operators are the same as for the Functional ς-calculus, with the essential difference that method override now *modifies* the object bound to the self variable, instead of generating a *new* object. Another related difference is that objects can be used several times for activation.

a, b	::=	$[\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}]$	object value
	|	x	self variable
	|	$a.\mathrm{l}$	method activation
	|	$a.\mathrm{l}{\Leftarrow}\varsigma(x{:}A)b$	method override
	|	$\mathsf{clone}(a)$	cloning
	|	let $x{:}A{=}a$ in b	local definition

Table 6.1: Syntax of the Imperative ς-calculus.

6.1.2 Operational semantics

We recall the operational semantics of the Imperative ς-calculus from Abadi and Cardelli's book.

The semantics for the Imperative ς-calculus is given as a relation that relates a store σ, a stack $\mathcal{S}$ (usually called an environment, but we stick to the name used by Abadi and Cardelli) and a term a with a value v and an updated store σ'. The relation is written $\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v \cdot \sigma'$ and is read: "the term a will, under the store σ and stack $\mathcal{S}$, reduce to the *value* v and store σ'.

Values v are of the form $[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]$ mapping method names l_i to locations ι_i. A store σ maps locations to closures. A closure is the pair $\langle \varsigma(x)b, \mathcal{S}\rangle$ of a method body $\varsigma(x)b$ together with a local stack. A stack $\mathcal{S}$ maps variables to values. For any mapping f we let $\mathsf{dom}(f)$ denote the domain of f.

Figure 6.1 illustrates the relations between the different parts of the operational semantics.

A stack $\mathcal{S}$ is well-formed w.r.t. a store σ, written $\sigma \cdot \mathcal{S} \vdash \diamond$, if for all $x \in \mathsf{dom}(\mathcal{S})$ $\mathcal{S}(x) = [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]$ where all ι_i are defined in σ. A store σ is well-formed, written $\sigma \vdash \diamond$, if the stacks in all closures in the store are well-formed w.r.t. the store.

If σ is a store, then we let $\sigma, \iota \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle$ denote the extension of σ with the new entry $\langle \varsigma(x)b, \mathcal{S}\rangle$ on the *new* location ι, assuming that $\sigma \cdot \mathcal{S} \vdash \diamond$. We

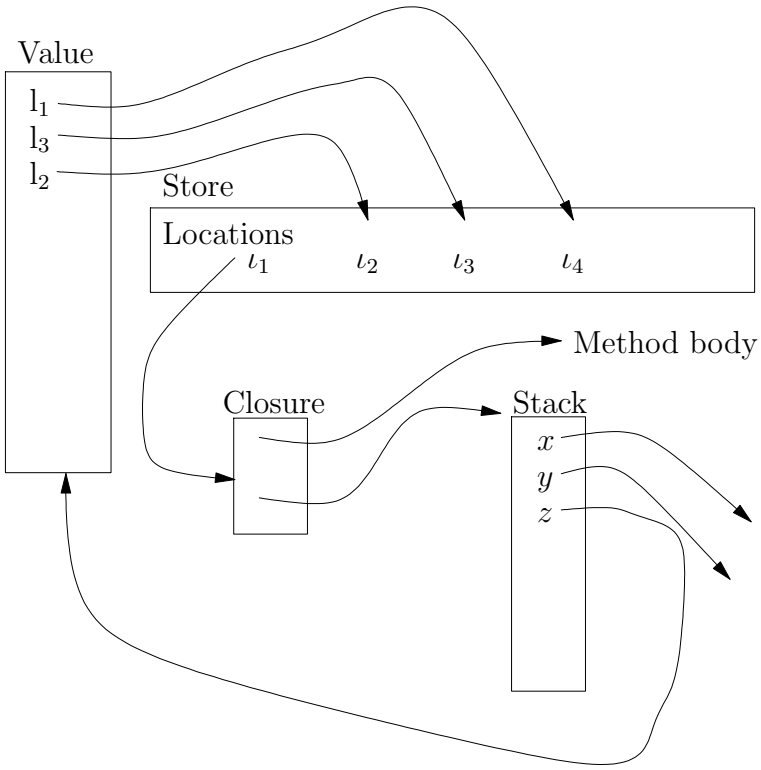


Figure 6.1: Illustration of the relations in the operational semantics.

write $\sigma[\iota \mapsto \langle \varsigma(x)b, s\rangle]$ for the update of location ι of store σ, assuming that $\iota \in \mathsf{dom}(\sigma)$ and again $\sigma \cdot \mathcal{S} \vdash \diamond$.

The operational semantics, given in Table 6.2, is untyped; type annotations are simply removed when evaluating terms.

We write $a{\Downarrow}v \cdot \sigma$ ("a converges to the value v and store σ") if we can deduce $\emptyset \cdot \emptyset \vdash a \rightsquigarrow v \cdot \sigma$ and $a{\Downarrow}$ iff $a{\Downarrow}v \cdot \sigma$ for some v and σ. If there is no v and σ such that $a{\Downarrow}v \cdot \sigma$ we write $a{\Uparrow}$ ("a diverges").

Example 6.1 To illustrate the operational semantics we give a small example from [AC96]. Consider the object $a = [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}{\Leftarrow}\varsigma(y)x.\mathrm{l}]$ and let us see what happens when we evaluate $a.\mathrm{l}$

Let $\sigma_0 = \iota \mapsto \langle \varsigma(x)x.\mathrm{l}{\Leftarrow}\varsigma(y)x, \emptyset\rangle$ and $\sigma_1 = \iota \mapsto \langle \varsigma(y)x, x \mapsto [\mathrm{l}{=}\iota]\rangle$

$$\frac{\emptyset \cdot \emptyset \vdash a \rightsquigarrow [\mathrm{l}{=}\iota] \cdot \sigma_0 \quad \dfrac{\sigma_0 \cdot x \mapsto [\mathrm{l}{=}\iota] \vdash x \rightsquigarrow [\mathrm{l}{=}\iota] \cdot \sigma_0}{\sigma_0 \cdot x \mapsto [\mathrm{l}{=}\iota] \vdash x.\mathrm{l}{\Leftarrow}\varsigma(y)x \rightsquigarrow [\mathrm{l}{=}\iota] \cdot \sigma_1}}{\emptyset \cdot \emptyset \vdash a.\mathrm{l} \rightsquigarrow [\mathrm{l}{=}\iota] \cdot \sigma_1}$$

(VarI)
$$\frac{\sigma \cdot \mathcal{S} \vdash \diamond \quad x \in \mathsf{dom}(\mathcal{S})}{\sigma \cdot \mathcal{S} \vdash x \rightsquigarrow \mathcal{S}(x)}$$

(ObjI) where $\sigma' = (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, \mathcal{S} \rangle^{\ i \in 1..n})$
$$\frac{\sigma \cdot \mathcal{S} \vdash \diamond \quad \iota_i \notin \mathsf{dom}(\sigma) \quad \iota_i \text{ distinct } \forall i \in 1..n}{\sigma \cdot \mathcal{S} \vdash [\mathrm{l}_i = \varsigma(x_i{:}A) b_i^{\ i \in 1..n}] \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot \sigma'}$$

(SelI)
$$\frac{\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot \sigma' \quad \sigma'(\iota_j) = \langle \varsigma(x_j) b_j, \mathcal{S}' \rangle \quad x_j \notin \mathsf{dom}(\mathcal{S}') \quad j \in 1..n \quad \sigma' \cdot \mathcal{S}', x_j \mapsto [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \vdash b_j \rightsquigarrow v \cdot \sigma''}{\sigma \cdot \mathcal{S} \vdash a.\mathrm{l}_j \rightsquigarrow v \cdot \sigma''}$$

(UpdI)
$$\frac{\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot \sigma' \quad j \in 1..n \quad \iota_j \in \mathsf{dom}(\sigma')}{\sigma \cdot \mathcal{S} \vdash a.\mathrm{l}_j \Leftarrow \varsigma(x{:}A) b \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot \sigma'[\iota_j \mapsto \langle \varsigma(x) b, \mathcal{S} \rangle]}$$

(CloneI)
$$\frac{\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot \sigma' \quad \forall i \in 1..n \quad \iota'_i \notin \mathsf{dom}(\sigma') \quad \iota'_i \text{ distinct}}{\sigma \cdot \mathcal{S} \vdash \mathsf{clone}(a) \rightsquigarrow [\mathrm{l}_i = \iota_i^{\ i \in 1..n}] \cdot (\sigma', \iota'_i \mapsto \sigma(\iota'_i)^{\ i \in 1..n})}$$

(LetI)
$$\frac{\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot \mathcal{S}, x \mapsto v' \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot \mathcal{S} \vdash \mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b \rightsquigarrow v'' \cdot \sigma''}$$

Table 6.2: The operational semantics for the Imperative ς-calculus

This example shows how we can create loops in the store. The store σ_1 contains a loop because it maps index ι to a closure, that binds x to a closure that contains index ι. □

6.1.3 Type system

The type system is a straightforward adaptation of $\mathbf{Ob}_{1<:}$ in Table 3.3 on page 33 to the Imperative ς-calculus. The only change is the addition of the two rules found in Table 6.3, which are needed to type the clone- and let-constructs.

Just as for the Functional ς-calculus, where Theorem 3.1 states the soundness of the type-system, we have a similar theorem for the Imperative ς-calculus. This theorem is, of course, a little more involved since the result of evaluat-

$$\frac{\Gamma \vdash a{:}A}{\Gamma \vdash \mathsf{clone}(a){:}A}\ (\textsc{Clone}) \qquad \frac{\Gamma \vdash a{:}A \quad \Gamma, x{:}A \vdash b{:}B}{\Gamma \vdash \mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b : B}\ (\textsc{Let})$$

Table 6.3: Additional typing rules for the Imperative ς-calculus

ing an imperative object term is *a value and a store*. We shall not give the precise statement of the subject reduction theorem here, but just note that what is essential for our use, is that if a configuration $\sigma \cdot \mathcal{S} \vdash a$ is well-typed and $\sigma \cdot \mathcal{S} \vdash a{\Uparrow}$, then it is *not* because we get stuck trying to access a non-existing method. (Subject reduction for the Imperative ς-calculus with the type system $\mathbf{Ob}_{1<:}$ can be found in [AC96, Section 11.4].)

Example 6.2 Consider the following two objects:

$$a = [\,].\mathrm{l} \qquad b = [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}].\mathrm{l}$$

Both objects diverge but for different reasons.

When we try to build an inference tree for a then we see that we cannot complete the rule (SelI):

$$(\textsc{SelI})\frac{(\textsc{ObjI})\dfrac{-}{\emptyset \cdot \emptyset \vdash [\,] \rightsquigarrow [\,] \cdot \emptyset}}{\emptyset \cdot \emptyset \vdash [\,].\mathrm{l} \rightsquigarrow}$$

The reason is that the object $[\,]$ does not contain the method l.

The reason why b diverges is that we can indefinitely apply the rule (ObjI).

$$(\textsc{SelI})\frac{(\textsc{ObjI})\dfrac{-}{\emptyset \cdot \emptyset \vdash [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}] \rightsquigarrow [\mathrm{l}{=}\iota] \cdot \sigma} \quad (\textsc{SelI})\dfrac{\dfrac{\vdots}{\sigma \cdot x \mapsto [\mathrm{l}{=}\iota] \vdash x.\mathrm{l} \rightsquigarrow}}{\sigma \cdot x \mapsto [\mathrm{l}{=}\iota] \vdash x.\mathrm{l} \rightsquigarrow}}{\emptyset \cdot \emptyset \vdash [\mathrm{l}{=}\varsigma(x)x.\mathrm{l}].\mathrm{l} \rightsquigarrow}\ \text{where } \sigma = \iota \mapsto \langle \varsigma(x)x.\mathrm{l}, \emptyset \rangle$$

What well-typedness ensures is that divergence caused by the inability to apply any inference rule does not occur[1]. □

[1]The need to make the distinction between the two types of divergence could be avoided by using a small step semantics.

6.1.4 A first look at a translation

Although the imperative semantics in Table 6.2 looks quite involved compared to the functional semantics in Table 3.2, the essential difference is simply that method update in the imperative semantics *modifies the object*, whereas method update in the functional semantics *generates a new object*. So, at a first glance, one might expect that it should be straightforward to change the translation from the previous chapter to work for the imperative semantics. But this is not as easy as it might seem.

Let us consider the translation of $a.\mathrm{l}\Leftarrow\varsigma(x{:}A)b$. As before we would first evaluate a on some name q private to the translation of the total expression, and over that name get the location, say s, of the object a. Now we get to the difficult part — we need to modify the existing object. That is, the result of a method override must be located at s (otherwise we would create a new object). So a first attempt of a translation using the relay construction might be:

$$[\![a.\mathrm{l}\Leftarrow\varsigma(x{:}A)b]\!]_p \triangleq (\nu q)\Big([\![a]\!]_q \mid q(z).(\overline{p}\langle z\rangle \mid z{:=}z.[\![\mathrm{l}\Leftarrow\varsigma(x{:}A)b]\!])\Big)$$

This will, of course, not work, because we get two receivers of requests to the resulting object (the one from the translation of a and the one from the relay construct). This introduces a nondeterministic choice between the old and new version of l, plus the possibility of divergence if requests for methods other than l are always received by the relay construct. A way to fix this is to make objects relocatable, so that we, when overriding a method, move the original object to a location private to the relay construct. This leads to the following proposal:

$$\begin{aligned}[\![a.\mathrm{l}\Leftarrow\varsigma(x{:}A)b]\!]_p \triangleq\ & (\nu q)\Big([\![a]\!]_q \mid q(z).(\nu s, r)(\overline{z}\langle \mathrm{relocate}, s, r\rangle \mid \\ & r(y).(\overline{p}\langle z\rangle \mid z{:=}s.[\![\mathrm{l}\Leftarrow\varsigma(x{:}A)b]\!]))\Big)\end{aligned}$$

Of course, the relay construct must be modified so that it on a relocate request will relocate itself and forward the request.

Having dealt with method override, we now need to take care of cloning. The expression $\mathsf{clone}(a)$ should create a copy of a — how can we do that? At first, one might think that we should just create a new object location s and locate a relay process at s that forwards all requests for method activation to the original object (also called the donor object). This relay construct should also react to relocate requests, but not forward them. This would have worked, had it not been for the possibility of overrides on the donor object, as in the following program:

$$\begin{array}{l}\mathsf{let}\ x{=}a\ \mathsf{in}\\ \quad \mathsf{let}\ y{=}\mathsf{clone}(x)\ \mathsf{in}\\ \quad\quad x.\mathrm{l}{\Leftarrow}\varsigma(z)b;\\ \quad\quad y.\mathrm{l}\end{array}$$

The problem is that the override on a also would affect the clone. Therefore $y.\mathrm{l}$ ends up activating b instead of what was originally bound to l in a. This indicates that we cannot use a relay to deal with cloning — we *need* to detach the clone from its donor. To do this, we need to *extract* the methods from the original object and install them in a new object. It would be no problem to add to the translation the possibility of extracting methods from objects. But this leads us to the same problem as the one we discussed in Section 5.1 — Which methods does an object contain? In Section 5.1 we avoided the need to know the names of the methods in an object by using mismatch. But now, we are less privileged, because we need precisely *all* the methods in the donor object. One way to obtain exactly the methods in the original object, would be to implement some kind of protocol, asking the donor object for a list of pairs of method names and bodies. And from this list we could then build the clone.

Although cumbersome, we believe that a translation built this way would work. But proving the correctness of such a translation would not be an easy task. This indicates that the way we translated the Functional ς-calculus is not the way one should deal with the Imperative ς-calculus. This leads us to consider another translation, where we abandon relays. Before presenting the translation, we proceed with the presentation of typed π-calculus that will allow a translation of types, too.

6.2 A typed mobile calculus

In this section, we present the typed π-calculus from [San98] in which we shall interpret the Imperative ς-calculus.

The type language we use for the π-calculus is taken from [PS96, San98]. In these type systems, as well as other type systems for the π-calculus like [Ode95, KPT96, Bor98, Yos96, VH93, VT93, Hon96], types are assigned to names. The types in [PS96] show the arity and the directionality of a name and, recursively, of the names carried by that name; the difference in [San98] is that *variant types* are used instead of tupling.

For simplicity, we take tuples as primitive in the grammar for values, although they can be coded up using variants [San98].

6.2.1 Syntax

The syntax of the typed π-calculus is given in Table 6.4. We let **Proc** denote the set of π-calculus processes and **Names** denote the set of names and **Value** the set of values.

The process constructs are those of the monadic π-calculus [MPW92] with matching replaced by a case construct. The latter can be thought of as a more disciplined form of matching, in which all tests on a given name are localized to a single place. The syntax for case is reminiscent of an analogous construct in [MPW92]. In the untyped calculus, matching and case are interderivable, but in the typed calculus case allows us simple but powerful typing and subtyping rules for which, moreover, any misuse of variant values in communications is easy to detect (rule (Case-W), Table 6.7).

We have omitted summation, since we will not need it in the interpretation of the Imperative ς-calculus. Restriction, is following [San98], explicitly typed.

The most important difference with respect to the monadic π-calculus is the addition of variant values. This introduces a vertical dimension on values, as opposed to the tupling construct of the polyadic π-calculus, which introduces an horizontal dimension. We should stress that variant values are rather simple, in that they are constructed out of names and variant tags only — they do not contain terms of the language. The construct wrong stands for a process in which a run-time type error has occurred — i.e., for instance, a communication in which the variant tag or the arity of the transmitted value was unexpected by its recipient. A soundness theorem guarantees that a well-typed process expression cannot reduce to an expression containing wrong.

Bound and free names, alpha conversion, and substitutions are defined in the expected way. We stipulate that $P\{\tilde{v}/\tilde{x}\}$ is wrong if replacing the $\tilde{v}$'s for the $\tilde{x}$'s in P does not yield a process expression according to the grammar of Table 6.4 (thus, for instance, $(\overline{x}w.\mathbf{0})\{\langle p,q\rangle/x\} = \mathsf{wrong}$).

6.2.2 Semantics

In Table 6.5, we give the semantics for the π-calculus as a labelled transition system. The advantage of a labelled semantics, compared to a reduction semantics [Mil93, San98], is that it easily allows us to define labelled forms of bisimulation. Process transitions are of the form $P \xrightarrow{\mu} P'$, where μ is a label given by the following syntax:

$$\alpha, \beta, \mu ::= (\nu\tilde{n}{:}\tilde{T})\overline{p}v \mid pv \mid \tau \mid \mathsf{wrong}$$

Names

$p, q, r \ldots x, y, z$

Variant Tags

l

Types

T	::=	$\mu(X)T$	recursive type
	|	X	type variable
	|	$[\,\mathrm{l}_1\_T_1..\mathrm{l}_n\_T_n\,]$	variant type
	|	$\langle T_1 \ldots T_n \rangle$	tuple type
	|	T^I	channel type

I/O Tags

I	::=	r	input only
	|	w	output only
	|	b	either

Values

v	::=	x	name
	|	$\langle v_1..v_n \rangle$	tuple value
	|	l_v	variant value

Processes

P	::=	$\mathbf{0}$	nil process
	|	$P \mid P$	parallel
	|	$(\nu \tilde{x}{:}\tilde{T})P$	restriction
	|	$p(x).P$	input
	|	$\overline{p}v.P$	synchronous output
	|	$!P$	replication
	|	$\mathsf{let}\ x_1..x_n{=}v\ \mathsf{in}\ P$	tuple destructor
	|	$\mathsf{case}\ v\ \mathsf{of}\ [\,\mathrm{l}_1\_(x_1) \triangleright P_1; \ldots; \mathrm{l}_n\_(x_n) \triangleright P_n\,]$	case
	|	wrong	error

where:

- In a recursive type $\mu(X)T$, variable X must be guarded in T, i.e., occur underneath an I/O-tag or underneath a variant tag;
- in a case statement, the tags l_i $(i \in 1..n)$ are pairwise distinct.

Table 6.4: Syntax of the typed π-calculus

The label $(\nu\tilde{n}{:}\tilde{T})\overline{p}v$ denotes the output of the value v on the name p. The restriction, $(\nu\tilde{n}{:}\tilde{T})$, where $\tilde{n}$ must be a subset of the names in v indicates that the names $\tilde{n}$ are bound names having types $\tilde{T}$; if there are no names bound we omit the restriction on the label. The label pv denotes the input of the value v over the name p. The action τ denotes an internal action. Finally, wrong denotes a run-time error. The soundness theorem (Theorem 6.1) guarantees that a well-typed program will never reduce to a term containing wrong.

Table 6.5 shows the inference rules with the symmetric versions of (Sync-L) and (Comp-L) omitted.

The rules for the operational semantics are standard rules of the π-calculus. The new, but expected, rules are those for let and case, in which run-time errors may be generated.

We write $P \xrightarrow{\mu}_d Q$ if $P \xrightarrow{\mu} Q$ is the only transition that P can perform. And as in Chapter 4 we let $\stackrel{\mu}{\Longrightarrow}$ denote weak transitions, and $P \Longrightarrow P'$ means "$P \stackrel{\tau}{\Longrightarrow} P'$ or $P = P'$".

6.2.3 Type system

The syntax for types is given in Table 6.4.

We recall that I/O annotations [PS96] separate the capabilities of reading and writing on a channel (we use "read" and "write" as synonyms for "input" and "output", respectively). For instance, a type $p : \langle S^{\mathsf{r}}\, T^{\mathsf{w}}\rangle^{\mathsf{b}}$ (for appropriate type expressions S and T) says that name p can be used *both* to read and to write and that any message at p carries a pair of names; moreover, the first component of the pair can be used by the recipient *only to read*, the second *only to write*.

Subtyping judgments, shown in Table 6.6, are of the form $\Sigma \vdash S \leq T$, where Σ represents the subtyping assumptions. Whenever the Σ is extended with the additional assumption $S \leq T$, we write this as $\Sigma[S \leq T]$, and if $S \leq T$ is an assumption in Σ, we write this as $(S \leq T) \in \Sigma$. We often write $S \leq T$, when the subtyping assumptions are empty. Note that type annotation r (an input capability) gives covariance, w (an output capability) gives contravariance, and b (both capabilities) gives invariance. Moreover, since a tag b gives more freedom in the use of a name, for each type T we have $T^{\mathsf{b}} \leq T^{\mathsf{r}}$ and $T^{\mathsf{b}} \leq T^{\mathsf{w}}$.

A *typing judgment* $\Gamma \vdash P$ (Table 6.7) asserts that process P is well-typed in Γ, and $\Gamma \vdash v : T$ (Table 6.8) that value v has type T in Γ (as usual is a *type environment* Γ a finite assignment of types to names). There is one typing rule for each process construct except wrong. The interesting rules are those for input and output prefixes and for case. In the rules for input and output prefixes, the subject of the prefix is checked to possess the appropriate input

(Inp)
$$\frac{-}{p(x).P \xrightarrow{pv} P\{v/x\}} \quad \forall v \in \textbf{Values}$$

(Out)
$$\frac{-}{\overline{p}v.P \xrightarrow{\overline{p}v} P}$$

(Open) where $p \notin \tilde{q},\ \tilde{q} \cap \tilde{n} = \emptyset$
$\tilde{n}{:}\tilde{T}' = (\tilde{n}{:}\tilde{T})(\tilde{q}{:}\tilde{S} \cap \mathsf{n}(v))$
$\tilde{q}'{:}\tilde{S}' = (\tilde{q}{:}\tilde{S}) \setminus \mathsf{n}(v)$
$$\frac{P \xrightarrow{(\nu\tilde{n}:\tilde{T})\overline{p}v} P'}{(\nu\tilde{q}{:}\tilde{S})P \xrightarrow{(\nu\tilde{n}':\tilde{T}')\overline{p}v} (\nu\tilde{q}'{:}\tilde{S}')P'}$$

(Res) where $\tilde{q} \cap \mathsf{n}(\mu) = \emptyset$
$$\frac{P \xrightarrow{\mu} P'}{(\nu\tilde{q}{:}\tilde{T})P \xrightarrow{\mu} (\nu\tilde{q}{:}\tilde{T})P'}$$

(Sync-L) where $\tilde{n} \cap \mathsf{fn}(Q) = \emptyset$
$$\frac{P \xrightarrow{(\nu\tilde{n}:\tilde{T})\overline{p}v} P' \quad Q \xrightarrow{pv} Q'}{P \mid Q \xrightarrow{\tau} (\nu\tilde{n}{:}\tilde{T})(P' \mid Q')}$$

(Comp-L) where $\mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset$
$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}$$

(Repl)
$$\frac{P \mid !P \xrightarrow{\mu} Q}{!P \xrightarrow{\mu} Q}$$

(Let)
$$\frac{-}{\mathsf{let}\ x_1..x_n{=}\langle v_1..v_n\rangle\ \mathsf{in}\ P \xrightarrow{\tau} P\{v_1..v_n/x_1..x_n\}}$$

(Let-W)
$$\frac{v \neq \langle v_1..v_m\rangle \text{ or } v = \langle v_1..v_m\rangle \text{ and } m \neq n}{\mathsf{let}\ x_1..x_n{=}v\ \mathsf{in}\ P \xrightarrow{\mathsf{wrong}} \mathsf{wrong}}$$

(Case)
$$\frac{\mathrm{l}_j \in \{\mathrm{l}_1 \ldots \mathrm{l}_n\}}{\mathsf{case}\ \mathrm{l}_j\_v\ \mathsf{of}\ [\,\mathrm{l}_1\_(x_1) \rhd P_1; \ldots; \mathrm{l}_n\_(x_n) \rhd P_n\,] \xrightarrow{\tau} P_j\{v/x_j\}}$$

(Case-W)
$$\frac{v \neq \mathrm{l}_j\_v' \text{ or } \mathrm{l}_j \notin \{\mathrm{l}_1 \ldots \mathrm{l}_n\}}{\mathsf{case}\ v\ \mathsf{of}\ [\,\mathrm{l}_1\_(x_1) \rhd P_1; \ldots; \mathrm{l}_n\_(x_n) \rhd P_n\,] \xrightarrow{\mathsf{wrong}} \mathsf{wrong}}$$

Table 6.5: The labelled transition system for the π-calculus.

$$\text{(A-BB)}\quad \frac{\Sigma \vdash S \leq T \;\; \Sigma \vdash T \leq S}{\Sigma \vdash S^{\mathsf{b}} \leq T^{\mathsf{b}}} \qquad \text{(A-XR)}\quad \frac{I \in \{\mathsf{b},\mathsf{r}\} \;\; \Sigma \vdash S \leq T}{\Sigma \vdash S^{I} \leq T^{\mathsf{r}}}$$

$$\text{(A-XW)}\quad \frac{I \in \{\mathsf{b},\mathsf{w}\} \;\; \Sigma \vdash T \leq S}{\Sigma \vdash S^{I} \leq T^{\mathsf{w}}} \qquad \text{(A-Case)}\quad \frac{\Sigma \vdash S_i \leq T_i \; i \in 1..n}{\Sigma \vdash [\,\mathrm{l}_1\_S_1..\mathrm{l}_n\_S_n\,] \leq [\,\mathrm{l}_1\_T_1..\mathrm{l}_{n+m}\_T_{n+m}\,]}$$

$$\text{(A-Tuple)}\quad \frac{\Sigma \vdash S_i \leq T_i \; \forall i \in 1..n}{\Sigma \vdash \langle S_1..S_n\rangle \leq \langle T_1..T_n\rangle} \qquad \text{(A-Rec-L)}\quad \frac{\Sigma[\mu(X)S \leq T] \vdash S\{^{\mu(X)S}/_X\} \leq T}{\Sigma \vdash \mu(X)S \leq T}$$

$$\text{(A-Ass)}\quad \frac{(S \leq T) \in \Sigma}{\Sigma \vdash S \leq T} \qquad \text{(A-Rec-R)}\quad \frac{\Sigma[S \leq \mu(X)T] \vdash S \leq T\{^{\mu(X)T}/_X\}}{\Sigma \vdash S \leq \mu(X)T}$$

Table 6.6: Subtyping rules

or output capability in the type environment. (Tv-sub) is the only rule which explicitly uses subtyping. We let $\mathbf{Proc}_\Gamma$ denote the class of processes well-typed in Γ.

Theorem 6.1 (Soundness of Type System) *If $\Gamma \vdash P$, all names in Γ have channel type and $P \Longrightarrow Q$ then Q does not contain the process* wrong.

The restriction to channel types is necessary to ensure that variant and tuple destruction on a name x is guarded underneath an input with x as object. For instance, without the restriction, the process $\mathsf{let}\ x_1, x_2{=}y\ \mathsf{in}\ \mathbf{0}$ would be well-typed under the assumption $y{:}\langle T_1, T_2\rangle$ (for any types T_1 and T_2) although $\mathsf{let}\ x_1, x_2{=}y\ \mathsf{in}\ \mathbf{0} \xrightarrow{\mathsf{wrong}} \mathsf{wrong}$.

We can argue for the soundness of the subtyping rules as follows. If $a{:}S$ and $S \leq T$, then we should be able to use a in settings where it is expected that a has type T. If $a : T^{\mathsf{b}}$, it means that a can be used for both output and input. Obviously, a can also be used in settings that only allow use of a for input or output. Therefore $T^{\mathsf{b}} \leq T^{\mathsf{w}}$, and similar for the input annotation.

(A-Case) express that a longer variant type is a supertype of a shorter, this is the opposite of (Sub Obj). To see why this is the right rule, one has to look at (T-Case). Consider, how we can finish the typing of:

$$v : [\mathrm{l}_1\_T_1,\ \mathrm{l}_2\_T_2] \vdash \mathsf{case}\ v\ \mathsf{of}[\mathrm{l}_1(x_1) \rhd P_1,\ \mathrm{l}_2(x_2) \rhd P_2,\ \mathrm{l}_3(x_3) \rhd P_3]$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ (\text{T-Nil}) \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P|Q}\ (\text{T-Par}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{T-Repl})$$

$$\frac{\Gamma[x{:}S^I] \vdash P}{\Gamma \vdash (\nu x{:}S^I)P}\ (\text{T-Restr}) \qquad \frac{\Gamma \vdash p{:}S^{\mathsf{r}} \quad \Gamma[x{:}S] \vdash P}{\Gamma \vdash p(x).P}\ (\text{T-In})$$

$$\frac{\Gamma \vdash p{:}S^{\mathsf{w}} \quad \Gamma \vdash w{:}S \quad \Gamma \vdash P}{\Gamma \vdash \overline{p}w.P}\ (\text{T-Out})$$

$$\frac{\Gamma \vdash v{:}\langle T_1..T_n\rangle \quad \Gamma[x_1{:}T_1, \ldots, x_n{:}T_n] \vdash P}{\Gamma \vdash \mathsf{let}\ x_1..x_n{=}v\ \mathsf{in}\ P}\ (\text{T-Let})$$

$$\frac{\Gamma \vdash v{:}[\,\mathrm{l}_1\_T_1..\mathrm{l}_n\_T_n\,] \quad \text{for each } i \in 1..n,\ \Gamma[x_i{:}T_i] \vdash P_i}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [\,\mathrm{l}_1\_(x_1) \rhd P_1; \ldots; \mathrm{l}_n\_(x_n) \rhd P_n\,]}\ (\text{T-Case})$$

Table 6.7: Process typing

This assumption $v : [\mathrm{l}_1\_T_1,\ \mathrm{l}_2\_T_2]$ says that v is a variant value, which can be either $\mathrm{l}_1\_v_1$ or $\mathrm{l}_2\_v_2$. On the other hand, the case construct tests for three possible tags. The rule (A-Case) allows us to assume that the value v range over a larger range, allowing us to apply (T-Case) (which we could not do if (A-Case) looked like (Sub Obj)).

The reason why the rule (A-XW) is contravariant can be explained as follows. If $a : [_{i\in I}\ \mathrm{l}_i\_T_i]^{\mathsf{w}}$, it means that we on the name a can pass values of type $[_{i\in I}\ \mathrm{l}_i\_T_i]$, that a receiver can test in a case construct. If (A-XW) was covariant, we would be allowed to send on a values that were out of range of the case construct on the receiving side, whereas contravariance allows us to restrict the set of possible tags.

Example 6.3 Consider the processes $P = !a(x).(\nu b{:}T)\overline{x}\langle a, b\rangle$. How should it be typed?

Obviously the name received on a should have a type that expresses, that it is used to output a tuple, so a good guess would be something like $x : \langle S, T\rangle^{\mathsf{w}}$. It is easy to see that the second component of the tuple type is T (because of the type annotation in the restriction). The type of the first component is a little bit more difficult, because a (on which we receive) is

$$\frac{\Gamma(p) = T}{\Gamma \vdash p : T}\ (\textsc{Tv-base}) \qquad \frac{\Gamma \vdash v : S \quad S \leq T}{\Gamma \vdash v : T}\ (\textsc{Tv-sub})$$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \mathrm{l\_}v : [\mathrm{l\_}T]}\ (\textsc{Tv-var}) \qquad \frac{\Gamma \vdash v_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash \langle x_1..x_n \rangle : \langle T_1..T_n \rangle}\ (\textsc{Tv-tuple})$$

Table 6.8: Value typing

the first component of the tuple. This indicates that S should be a recursive type.

If we assume that the receiver of a can use a only for writing, we can make the following guess of a type assumption: $a : (\mu(X)\langle X^{\mathsf{w}}, T\rangle^{\mathsf{w}})^{\mathsf{b}}$. It says, a is a name on which we receive a name, that is used to output a tuple, where the first component can be used for the same as a, and the second has type T.

The following inference shows that P is well-typed under this assumption. We let $S = \mu(X)\langle X^{\mathsf{w}}, T\rangle^{\mathsf{w}}$, $\Gamma = a{:}S^{\mathsf{b}}$, $\Gamma' = a{:}S^{\mathsf{b}},\ x{:}S$, and $\Gamma'' = a{:}S^{\mathsf{b}},\ x{:}S,\ b{:}T$.

$$\dfrac{\dfrac{\Gamma \vdash a{:}S^{\mathsf{b}} \ S^{\mathsf{b}} \leq S^{\mathsf{r}}}{\Gamma \vdash a{:}S^{\mathsf{r}}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\langle S^{\mathsf{w}}, T\rangle \leq S}{\Gamma'' \vdash x{:}S \ S \leq \langle S^{\mathsf{w}}, T\rangle^{\mathsf{w}}}}{\Gamma'' \vdash x{:}\langle S^{\mathsf{w}}, T\rangle^{\mathsf{w}}} \qquad \dfrac{\dfrac{\Gamma'' \vdash a{:}S^{\mathsf{b}} \ S^{\mathsf{b}} \leq S^{\mathsf{w}}}{\Gamma'' \vdash a{:}S^{\mathsf{w}}} \quad \Gamma'' \vdash b{:}T}{\Gamma'' \vdash \langle a, b\rangle : \langle S^{\mathsf{w}}, T\rangle}}{\Gamma'' \vdash \overline{x}\langle a, b\rangle}}{\Gamma' \vdash (\nu b{:}T)\overline{x}\langle a, b\rangle}}{\Gamma \vdash a(x).(\nu b{:}T)\overline{x}\langle a, b\rangle}}{\Gamma \vdash\ !a(x).(\nu b{:}T)\overline{x}\langle a, b\rangle}$$

□

6.2.4 Some derived constructs

In the translation, we shall use:

Recursive definitions, $A(\tilde{x}) \triangleq P$ which can be defined in a standard fashion from replication (c.f. [Mil93]);

Polyadic inputs, $a(x_1 \ldots x_n).P$, defined as $a(y).\mathsf{let}\ \langle x_1 \ldots x_n\rangle{=}y\ \mathsf{in}\ P$ for $y \notin \mathsf{fn}(P)$;

Variant inputs, such as $p[_{j\in 1..n}\ \mathrm{l}_j\text{-}[_{i\in 1..m}\ \mathrm{l}_{i,j}\text{-}(\tilde{x}_{i,j}) \triangleright P_{i,j}]]$, defined as

$$p(y).\mathsf{case}\ y\ \mathsf{of}\ [_{j\in 1..n}\ \mathrm{l}_j(y_j) \triangleright \mathsf{case}\ y_j\ \mathsf{of}\ [_{i\in 1..m}\ \mathrm{l}_{i,j}(\tilde{x}_{i,j}) \triangleright P_{i,j}]],$$

where $y \notin \mathsf{fv} \cup_{j\in 1..n,\ i\in 1..m} P_{i,j}$ and $y_j \notin \mathsf{fv} \cup_{i\in 1..m} P_{i,j}$.

This abbreviation allows us to go down two levels into the structure of a variant value received in an input at p; in fact, this term interacts with output particles of the form $\overline{p}\mathrm{l}_r\text{-}\mathrm{l}_{s,r}\text{-}\tilde{w}$ (with $r \in 1..n$, $s \in 1..m$, and tuple $\tilde{w}$ of the same length as $\tilde{x}_{r,s}$) and, in doing so, it reduces in 3 reductions to $P_{s,r}\{\tilde{w}/\tilde{x}_{s,r}\}$.

6.2.5 Barbed bisimulation and congruence

The behavioral equality we adopt for the π-calculus is *barbed congruence*.

In an untyped calculus, no constraint is made on processes or contexts. In a typed calculus, it makes sense to compare only processes that obey the same typing. This lead to the following definition of typed barbed bisimulation:

Definition 6.2 (Typed Barbed Bisimulation) *A symmetric relation* $\mathcal{R} \subseteq \mathbf{Proc}_\Delta \times \mathbf{Proc}_\Delta$ *is a* barbed Δ-bisimulation *if* $P\ \mathcal{R}\ Q$ *implies:*

i. If $P \xrightarrow{\tau} P'$ *then there exists a* Q' *such that* $Q \xrightarrow{\tau} Q'$ *and* $P'\ \mathcal{R}\ Q'$.

ii. for each name p $P{\downarrow_p}$ *iff* $Q{\downarrow_p}$.

Two processes P *and* Q *are* barbed Δ-bisimilar*, written* $P \dot{\sim}_b^\Delta Q$*, if* $P\ \mathcal{R}\ Q$*, for some barbed* Δ*-bisimulation* $\mathcal{R}$.

The only change compared to Definition 4.4 is that we only consider pairs of processes that has the same typing.

The changes needed to the definition of barbed congruence are somewhat more involved. The contexts in which (typed) processes are tested should be compatible with their types. We call a (Γ/Δ)-context a context which, when filled in with processes obeying the typing Δ, becomes a process obeying the typing Γ. The typing Γ might contain names not in Δ; the converse might be true too, because of binders in the context which embrace the hole.

Definition 6.3 *We write* $\Gamma <_{\mathrm{E}} \Delta$ *if, for each* x *on which* Δ *is defined, also* Γ *is defined and* $\Gamma(x) \leq \Delta(x)$.

Definition 6.4 ((Γ/Δ)-context) *Given type environments Γ and Δ and a process context C, we say that C is a (Γ/Δ)-context if $\Gamma \vdash C$ assuming the following additional typing rule for the hole $[\,]$ of C:*

$$\frac{\Gamma' <_{\mathrm{E}} \Delta}{\Gamma' \vdash [\,]}$$

(where Γ' is a metavariable over type environments).

Example 6.4 Consider the process $\overline{a}x|P$. If $\Gamma \vdash \overline{a}x|P$ then $a \in \mathsf{dom}(\Gamma)$. Given the context $C[-] = (\nu a{:}T)([-] \mid Q)$, the environment Δ used to type C, should contain assumptions about the new free names introduced by Q, but it would not contain a (because of the restriction). □

Definition 6.5 (Barbed Congruence) *Two processes $P, Q \in \mathbf{Proc}_\Delta$ are* barbed Δ-congruent*, written $P \sim_b^\Delta Q$, if, for each type environment Γ and (Γ/Δ)-context C, we have $C[P] \dot{\sim}_b^\Gamma C[Q]$.*

In the remainder of the chapter, we write $P \sim_b^\Delta Q$ without recalling the assumption that P and Q are well-typed in Δ.

Similarly, we can define weak barbed bisimulation ($\dot{\approx}_b^\Delta$) and weak barbed congruence ($\approx_b^\Delta$), by allowing that one reduction can be matched by a series of reductions (possibly zero) and using weak barbs in the definition of barbed bisimulation. In the proof of operational correctness, we also need the untyped version of barbed congruence, denoted by $\sim_b$.

Example 6.5 Consider the following two processes:

$$\begin{aligned} R_1 &= a(x).\mathsf{case}\ x\ \mathsf{of}\ [\,\mathrm{l}_1\_(y) \rhd P_1,\ \mathrm{l}_2\_(y) \rhd P_2] \\ R_2 &= a(x).\mathsf{case}\ x\ \mathsf{of}\ [\,\mathrm{l}_1\_(y) \rhd P_1,\ \mathrm{l}_2\_(y) \rhd P_2'] \end{aligned}$$

Using untyped barbed congruence we have $R_1 \not\sim_b R_2$ (unless $P_2 \sim_b P_2'$). But, if we compare R_1 and R_2 under the typing $\Gamma', a{:}[\,\mathrm{l}_1\_T_1\,]^{\mathsf{r}}$, R_1 and R_2 are equivalent because the typing prevents P_2/P_2' from being started (by "hiding" l_2). If we instead use the typing $\Gamma = \Gamma', a{:}[\,\mathrm{l}_1\_T_1,\ \mathrm{l}_2\_T_2]^{\mathsf{r}}$, then P_2/P_2' can activated, and just as for untyped barbed congruence, $R_1 \not\sim_b^\Gamma R_2$. □

Barbed congruence requires a quantification over all contexts. Therefore proving process equalities can be tedious. Thus, it is important to have powerful proof techniques. One such a technique consists in using *labelled*

bisimilarities whose definition does not require context quantification. For this, the labelled bisimilarity relation must be contained in barbed congruence.

For instance, in the untyped π-calculus, barbed congruence can be recovered using the well-known *early labelled bisimilarity*. Its typed version, *typed early labelled bisimulation*, is studied by Boreale and Sangiorgi [BS98]. In Appendix A.2, we present a simplified form of their definition that will be enough for our purposes. Below we give a brief sketch of the construction.

A typed labelled bisimulation contains triples (Λ, P, Q) where Λ is, roughly, a type environment (more precisely, it is a multiset type environment). Intuitively, (Λ, P, Q) being in a typed bisimulation means that P and Q are indistinguishable by an observer whose use of names respect the type informations in Λ. The type environment gives us an estimate of the type information that an observer interacting with the process may know. An observer can not know more than what the type environment indicates. Consequently, the observer may only see those actions of the processes that are well typed w.r.t. the type environment. When P and Q perform actions, the observer's knowledge changes as well; therefore Λ has to be updated. For instance, if $\Lambda \vdash p : T^{\mathsf{r}}$ meaning that the environment can see outputs on p, and P and Q export a local name, performing actions $P \xrightarrow{(\nu n{:}S)\overline{p}n} P'$ and $Q \xrightarrow{(\nu n{:}S')\overline{p}n} Q'$, respectively, then the update of Λ is $\Lambda, n{:}T$ (because T represents the observer's view of names that he/she receives along p).

The technique of labelled bisimulation can be made more powerful by combining it with up-to techniques, like "up to parallel composition" and "up to injective substitutions".

6.3 The interpretation

In Section 6.1 we discussed why our translation of the Functional ς-calculus in Chapter 5 is not easily adaptable to the Imperative ς-calculus. The problems encountered were related to how method override is handled by a relay in the translation of the Functional ς-calculus. Instead, when dealing with method override our solution is reminiscent of that for method activation. In the translation of the Functional ς-calculus, we send a request $\overline{s}\langle \mathrm{l}, s, r\rangle$ to a process responsible for activating the method bound to l, when we want to activate the method named l. Now, what if we do the same thing when wanting to override a method — i.e. we send a request to the object, asking it to update the method body.

To understand the π-calculus interpretation, it may be helpful to see

first an intermediate interpretation into the *Higher-Order π-calculus* (HOπ) [San93], an extension of the π-calculus where arguments of communications and recursive definitions may be, besides names, also *abstractions*, i.e., parameterized processes. For the interpretation of the Imperative ς-calculus, we only need abstractions in recursive definitions. More precisely, we need certain parameters of recursive definitions to be functions from names to processes. An example of such a recursive definition is

$$K(f,p) \triangleq p(x).(f\langle p\rangle | K\langle f,x\rangle)$$

Here, f is a function parameter, and p a name parameter; $f\langle p\rangle$ is the process obtained by applying function f to name p. We write functions from names to processes using a lambda notation, like in $\lambda(x,y).P$. The interpretation into HOπ is shown in Table 6.9. We have omitted the type annotations[2].

$$
\begin{array}{rcl}
\{\!|[\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]|\!\}_p & \triangleq & (\nu s)(\overline{p}s \mid \mathsf{OB}\langle\lambda(x_1,r).\{\!|b_1|\!\}_r,\ldots,\lambda(x_n,r).\{\!|b_n|\!\}_r,s\rangle) \\
\{\!|a.\mathrm{l}_j|\!\}_p & \triangleq & (\nu q)(\{\!|a|\!\}_q \mid q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p) \\
\{\!|a.\mathrm{l}_j{\Leftarrow}\varsigma(x_j)b|\!\}_p & \triangleq & (\nu q)(\{\!|a|\!\}_q \mid q(x).\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle p,\lambda(x_j,r).\{\!|b|\!\}_r\rangle) \\
\{\!|x|\!\}_p & \triangleq & \overline{p}x \\
\{\!|\mathsf{clone}(a)|\!\}_p & \triangleq & (\nu q)(\{\!|a|\!\}_q \mid q(x).\overline{x}\mathsf{clone}\_p) \\
\{\!|\mathsf{let}\ x{=}a\ \mathsf{in}\ b|\!\}^E_p & \triangleq & (\nu q)(\{\!|a|\!\}_q \mid q(x).\{\!|b|\!\}_p)
\end{array}
$$

where OB is defined as:

$$
\begin{array}{l}
\mathsf{OB}(f_1\ldots f_n,s) \triangleq \\
\quad s\Big[_{j\in 1..n}\ \mathrm{l}_j\_[\ \mathsf{sel}\_(x) \rhd f_j\langle s,x\rangle \mid \mathsf{OB}\langle f_1\ldots f_n,s\rangle; \\
\qquad\qquad \mathsf{upd}\_(x,y) \rhd \overline{x}s \mid \mathsf{OB}\langle f_1\ldots f_{j-i},y,f_{j+1}\ldots f_n,s\rangle\ \ ]; \\
\qquad\qquad \mathsf{clone}\_(x) \rhd \mathsf{OB}\langle f_1\ldots f_n,s\rangle \mid (\nu s')(\overline{x}s' \mid \mathsf{OB}\langle f_1\ldots f_n,s'\rangle)\Big]
\end{array}
$$

Table 6.9: The intermediate translation into HOπ (sketch)

The translation $\{\!|a|\!\}_p$ of an Imperative ς-calculus term a is located at a

[2]We are omitting type annotations for ease of reading; they would however be necessary to make the definition formal. Types are taken into account in the interpretations into the π-calculus in Table 6.10.

channel p. When a is an object value, with methods l_i ($i \in 1..n$), its translation is a process whose first action is to signal its valuehood by providing an access name s to its value-core, which is a process of the form $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$. This process is ready to accept along the access name s requests of selection, update and cloning for the methods l_j. The body of a method l_j is the function f_j; the set of these functions form the state of $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$.

We explain the behavior of $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$ on operations of selection, update and cloning. In case of a select operation $\mathrm{l}_j\_\mathsf{sel}\_p$ (which reads "activate method l_j and use p as location for the resulting object") the body f_j of method l_j is activated, with arguments $\langle s, p\rangle$; argument s, the access name of the value-core $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$, represents the self-parameter. An update request $\mathrm{l}_j\_\mathsf{upd}\_\langle p, f\rangle$ (which reads "replace current method body for l_j with f, and use p as the location of the resulting object value") results in a side effect on OB, whereby the j-th component of its state is updated to f. In a clone request clone_p (which reads "create a copy of the current object at location p"), a new object is created that has the same value-core $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$. Note the recursive definition of $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$, which shows that $\mathsf{OB}\langle f_1 \dots f_n, s\rangle$ may accept arbitrarily many requests at s.

Now, following the translation of HOπ into π-calculus [San93], we can turn the previous interpretation into a π-calculus one. Therefore, it suffices to make a recursive call with a functional argument, like $K\langle\lambda x.P\rangle$, into a first-order recursive call whose argument is a pointer to the function, like in:

$$(\nu b)(K\langle b\rangle \mid !b(x).P)$$

Correspondingly, a function application becomes an output of the arguments of the function along the pointer to the function. The result of this transformation, with the addition of type annotations, is presented in Table 6.10.

Just as in the interpretation of ς-calculi into the λ-calculus [ACV96], our translation of the terms has an environment E as parameter in order to put the necessary type annotations in the translation of method selection. This parameter could be avoided by providing, for instance, more type information on the syntax of method selection. We assume that $p, q, r, b, s \dots$ are not ς-calculus variables.

In the remainder of the thesis, we call a process of the form $\mathsf{OB}^A\langle b_1 \dots b_n, s\rangle$ an *object manager* (in the interpretation of an object, $\mathsf{OB}^A\langle b_1 \dots b_n, s\rangle$ acts like an administrator for the object; it "owns" the object methods, in the sense that it is only object managers which can reach them, via names b_i's).

Finally we need to show how to translate types. For this it suffices to follow the definition of the object manager. The translation of an object type

$$
\begin{array}{rcl}
\{\!|[\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]|\!\}_p^E & \triangleq & (\nu s{:}\{\!|A|\!\}^{\mathsf{b}})\Big(\overline{p}s \mid (\nu b_i{:}T_{A,i}^{\mathsf{b}}\ ^{i\in 1..n})\Big(\\
& & \qquad \mathsf{OB}^A\langle b_1 \ldots b_n, s\rangle \mid \prod_{i\in 1..n} !b_i(x_i, r).\{\!|b_i|\!\}_r^{E,x_i:A}\Big)\Big) \\
\{\!|a.\mathrm{l}_j|\!\}_p^E & \triangleq & (\nu q{:}\{\!|[\mathrm{l}_j : B_j]|\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!|a|\!\}_q^E \mid q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p) \\
\{\!|a.\mathrm{l}_j \Leftarrow \varsigma(x_j{:}A).b|\!\}_p^E & \triangleq & (\nu q{:}\{\!|A|\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!|a|\!\}_q^E \mid q(x).(\nu b{:}T_{A,j}^{\mathsf{b}}) \\
& & \qquad (\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle p, b\rangle \mid !b(x_j, r).\{\!|b|\!\}_r^{E,x_j:A})) \\
\{\!|x|\!\}_p^E & \triangleq & \overline{p}x \\
\{\!|\mathsf{clone}(a)|\!\}_p^E & \triangleq & (\nu q{:}\{\!|A|\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!|a|\!\}_q^E \mid q(x).\overline{x}\mathsf{clone}\_p) \\
\{\!|\mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b|\!\}_p^E & \triangleq & (\nu q{:}\{\!|A|\!\}^{\mathsf{w}^{\mathsf{b}}})(\{\!|a|\!\}_q^E \mid q(x).\{\!|b|\!\}_p^{E,x:A})
\end{array}
$$

The object manager OB^A is defined as:

$$
\begin{array}{l}
\mathsf{OB}^A(b_1{:}T_{A,1}^{\mathsf{w}} \ldots b_n{:}T_{A,n}^{\mathsf{w}}, s{:}\{\!|A|\!\}^{\mathsf{b}}) \triangleq \\
\quad s\Big[_{i\in 1..n}\ \mathrm{l}_i\_[\ \mathsf{sel}\_(x) \rhd \overline{b}_i\langle s, x\rangle \mid \mathsf{OB}^A\langle b_1 \ldots b_n, s\rangle; \\
\qquad\qquad \mathsf{upd}\_(x, y) \rhd \overline{x}s \mid \mathsf{OB}\langle b_1 \ldots b_{i-1}, y, b_{i+1} \ldots b_n, s\rangle \quad]; \\
\qquad\qquad \mathsf{clone}\_(x) \rhd \mathsf{OB}^A\langle b_1 \ldots b_n, s\rangle \mid (\nu s'{:}\{\!|A|\!\}^{\mathsf{b}})(\overline{x}s' \mid \mathsf{OB}^A\langle b_1 \ldots b_n, s'\rangle)\Big]
\end{array}
$$

and where

- $A = [\mathrm{l}_j : B_j]$ and $T_{A,j} \triangleq \langle \{\!|A|\!\}^{\mathsf{w}}, \{\!|B_j|\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle$
- in the encoding of selection, B_j is the unique type s.t. $E \vdash a{:}[\ldots, \mathrm{l}_j{:}B_j, \ldots]$ holds, if one such judgment exists (the unicity of this type if a consequence of the minimum-type property of $\mathbf{Ob}_{1<:}$), B_j can be any type otherwise;
- in the rule for update, x does not occur free in b.

Table 6.10: The interpretation of the Imperative ς-calculus into the π-calculus

must be a type that specifies repeated selection, update and clone operations.

$$\{\!|[\mathrm{l}_j{:}B_j\ ^{j\in 1..n}]|\!\} \triangleq \mu X.\Big[_{j\in 1..n}\ \mathrm{l}_j\_[\ \mathsf{sel}\_\{\!|B_j|\!\}^{\mathsf{w}^{\mathsf{w}}};\ \mathsf{upd}\_\langle X^{\mathsf{w}^{\mathsf{w}}}, \langle X^{\mathsf{w}}, \{\!|B_j|\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle^{\mathsf{w}}\rangle\ ];\ \mathsf{clone}\_X^{\mathsf{w}^{\mathsf{w}}}\ \Big]$$

The pattern of occurrences of w tags is determined by the protocol which implements select and update operations. What is important, however, is the level of nesting of w tags: An even number of nesting gives covariance, whereas an odd number of nesting gives contravariance. Thus, the component $\{\!|B_j|\!\}$ is in covariant position on selection, and in contravariant position on update: This explains the invariance of object types on the common components, in rule (Sub Obj) (the interpretation of the Imperative ς-calculus into the λ-calculus [ACV96] does the same). Type environments are then interpreted componentwise:

$$\begin{aligned} \{\!|\emptyset|\!\} &\triangleq \emptyset \\ \{\!|\Gamma, x{:}A|\!\} &\triangleq \{\!|\Gamma|\!\}, x{:}\{\!|A|\!\}^{\mathsf{w}} \end{aligned}$$

6.4 Simplifying the imperative part of the encoding

In the interpretation of the Imperative ς-calculus in Section 6.3, the key component is the object manager OB. This process is given as a recursive definition in which certain parameters may change over time. Having a state, this process may be regarded as "imperative".

In this section, we modify the encoding, so that the only imperative processes are cell-like processes, each of which just stores a name. All remaining processes will be stateless, and may therefore be regarded as "functional". We thus obtain a clean separation of the interpretation into a functional part and a very simple imperative part. This factorization is useful because:

i. It allows us take full advantage of various π-calculus proof techniques for functional names and functional processes.

ii. It shows what — we believe — are the simplest non-functional π-calculus processes that are needed for translating the imperative features of the Imperative ς-calculus.

We discuss functional names and processes, informally, below.

6.4.1 Functional names and functional processes

A π-calculus name is *functional*, if its response to incoming messages does not change over time. To spell this out more clearly, a name a is functional if the input offer at a is persistent and uniform. That is, an input at a is always available — at least as long as there are processes that could send messages at a — and the continuation after an input on a is always syntactically the same. Thus, messages sent at a can be processed immediately, and all messages are processed in the same way. Typically, a name a is functional if it only appears in subexpressions of the form

$$(\nu a)(!a(p).P \mid Q) \tag{6.1}$$

where P and Q only possess the output capability on a (this is the Lπ constraint on names), or of the form

$$(\nu a)(a(p).P \mid Q) \tag{6.2}$$

if, in addition, a can only be used once in output. (In [San99b] names obeying these constraints are called *uniformly receptive*.)

Functional names have advantages. First, they can be implemented more efficiently than arbitrary names. For instance, in PICT [PT99], a programming language based on the π-calculus, the compiler may recognize functional names and will then perform optimizations in the code that implements communications.

Another advantage of functional names is their algebraic properties, among which:

i. Copying or distributivity laws such as

$$(\nu a)(!a(b).P \mid Q|R) = (\nu a)(!a(b).P|Q) \mid (\nu a)(!a(b).Q|R).$$

Their effect is to localize computation. In this way, analyzing a process behavior becomes easier.

ii. τ-insensitiveness: Interactions along a functional name may not affect the future behavior of a process. As a consequence, when comparing the behavior of two processes, there are fewer configurations to take into account.

Let us call a process *functional*, if all inputs made by the process during its lifetime are at functional names. Functional languages may be interpreted into a sublanguage of the π-calculus in which all names are functional. For instance, in the encodings of call-by-name and call-by-value λ-calculus [Mil92] and Sangiorgi's encoding of the Functional ς-calculus [San98], all input prefixes are of, or can be put into, the format (6.1) or (6.2).

6.4.2 The factorized encoding

The new, factorized, encoding is defined in Table 6.11. To enhance readability we omit types, as they are the same as in the previous encoding. Only the clause for evaluated objects changes. Previously, the access name of the methods was part of (the state of) the object manager. By contrast, now a level of indirection is introduced, such that when updating a method, the indirection, instead of the object manager, is changed; this way the object manager becomes functional.

A cell $\mathsf{Cell}\langle\iota, n\rangle$ stores a pointer n to a method; and can be accessed for read and write operations at ι. The cells are the only imperative processes in the encoding. Being imperative, a cell may be *shared* by several clients, but may not be *copied* among them. By contrast, all other resources are functional and they may be copied among their clients. A cell will accept two messages: read and write, with the expected meaning. A read operation has a first step in which a client sends a return channel, and a second step where the cell communicates its value along the return channel.

The copy operator Copy is used to create new cells in a clone operation. The operator $\mathsf{Copy}^n(\tilde{\iota}, \tilde{\iota}', x, s)$ creates *sequentially* n new cells located at the names $\tilde{\iota}'$ with the contents of the cells located at $\tilde{\iota}$, signaling completion by sending the name s on x. One might think that the creation of the cell can be made parallel, thus:

$$\mathsf{Copy}^n(\tilde{\iota}, \tilde{\iota}', x, s) \triangleq \prod_{i\in 1..n} (\nu g)(\overline{\iota_i}\mathsf{read}\_g.g(m).\mathsf{Cell}\langle\iota'_i, m\rangle) \mid \overline{x}s,$$

that is, as a parallel composition of updates. Unfortunately, this will free the acknowledgment of the clone operation too early, which can lead to unexpected effects; for instance in the translation of the expression:

$$\mathsf{let}\ x{=}a\ \mathsf{in}\ \mathsf{let}\ y{=}\mathsf{clone}(x)\ \mathsf{in}\ x.\mathrm{l}_j{\Leftarrow}\varsigma(z)b$$

Since the acknowledgment of the clone operation can be sent before the cell for the method l_j has been copied, y may end up having a cell pointing to the new l_j method of x instead of the old. Therefore the copy operation has to ensure that the acknowledgment is sent *after* all the new cells have been created.

This factorized encoding is less compact, but has a simpler correctness proof than that of the previous encoding of Section 6.3. In the next section, we use this factorized encoding for proving the correctness of the original one.

Below, we let $\tilde{\iota}$ denote $\iota_1 \ldots \iota_n$, and $\tilde{\iota}'$ denote $\iota'_1 \ldots \iota'_n$.

$$[\![[\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]]\!]_p \ \triangleq\ (\nu s)\Big(\overline{p}s \mid (\nu\tilde{\iota})(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \prod_{i\in 1..n} (\nu b_i)(\mathsf{Cell}\langle \iota_i, b_i\rangle \mid !b_i(x_i, r).[\![b_i]\!]_r))\Big)$$

The other clauses are as for the original translation in Table 6.10.

The object manager is now defined thus:

$$\begin{array}{l}\mathsf{OB}_f(\tilde{\iota}, s) \ \triangleq \\ \quad !s\Big[_{i\in 1..n}\ \mathrm{l}_i\_[\ \mathsf{sel}\_(x) \rhd (\nu g)(\overline{\iota_i}\mathsf{read}\_g.g(m).\overline{m}\langle s, x\rangle); \\ \qquad\qquad\quad \mathsf{upd}\_(x, y) \rhd \overline{\iota_i}\mathsf{write}\_y.\overline{x}s) \qquad\qquad\quad]; \\ \qquad\qquad \mathsf{clone}\_(x) \rhd (\nu\tilde{\iota}', s')(\mathsf{OB}_f\langle \tilde{\iota}', s'\rangle \mid \mathsf{Copy}^n\langle \tilde{\iota}, \tilde{\iota}', x, s'\rangle)\quad \Big]\end{array}$$

where $\mathsf{Cell}(\iota, m)$ and Copy^n $(n \geq 0)$ are defined as:

$$\begin{array}{rcl}\mathsf{Cell}(\iota, m) & \triangleq & \iota[\ \mathsf{read}\_(x) \rhd \overline{x}m \mid \mathsf{Cell}\langle \iota, m\rangle; \\ & & \quad \mathsf{write}\_(y) \rhd \mathsf{Cell}\langle \iota, y\rangle] \\ \mathsf{Copy}^n(\tilde{\iota}, \tilde{\iota}', x, s) & \triangleq & (\nu g)\overline{\iota_n}\mathsf{read}\_g.g(m).(\mathsf{Cell}\langle \iota'_n, m\rangle \mid \mathsf{Copy}^{n-1}\langle \tilde{\iota}, \tilde{\iota}', x, s\rangle) \\ \mathsf{Copy}^0(\tilde{\iota}, \tilde{\iota}', x, s) & \triangleq & \overline{x}s\end{array}$$

Table 6.11: The factorized encoding

6.5 Correctness of the interpretation

6.5.1 Correctness of the interpretation of types

For proving the correctness of our translation of types we can, essentially, reuse the analogous proofs for Sangiorgi's interpretation of the Functional ς-calculus [San98], discussed in Section 6.6. We shall therefore only state the results of correctness of the translation of types and refer the interested reader to [San98].

Theorem 6.6 (Correctness of Subtyping) *For all A, B, it holds that $\vdash A <: B$ iff $\emptyset \vdash \{\!|A|\!\} \leq \{\!|B|\!\}$.*

This result allows us to reason about the $\mathbf{Ob}_{1<:}$ subtype relation using its translation into the type system for the π-calculus. It is, of course, also essential for the following theorem.

Theorem 6.7 (Correctness of Type Judgements)

i. If $E \vdash a{:}A$ then, for all p, it holds that $\{\!|E|\!\}, p{:}\{\!|A|\!\}^{\mathsf{w}^{\mathsf{w}}} \vdash \{\!|a|\!\}_p^E$.

ii. If $\{\!|E|\!\}, p{:}\{\!|A|\!\}^{\mathsf{w}^{\mathsf{w}}} \vdash \{\!|a|\!\}_p^E$, then $E \vdash a{:}A$.

The relation between type judgements in the Imperative ς-calculus and in the translation allows us to prove properties about the Imperative ς-calculus that rely on types using the translation (see Section 6.7).

6.5.2 Operational correctness

The proof of the correctness of the interpretation with respect to Abadi and Cardelli's operational semantics has a few tricky points. First of all, we need to extend the translation to deal with configurations (objects plus stores plus stacks) in the semantics. We therefore need to add translations of stores (σ), stacks ($\mathcal{S}$) and values (v), such that we can relate the semantics of the Imperative ς-calculus to the behavior of the encoding. We found it quite difficult to make such an extension of the original encoding $\{\!|-|\!\}$ of Section 6.3, due to sharing of closures, which is inevitable in the Imperative ς-calculus.

To understand the problem, let us try to come up with a translation for a configuration $\sigma \cdot \mathcal{S} \vdash a$. A stack maps variables to values and values are the object managers in our encoding. Finally, stores maps locations to closures. This suggests the following encoding of the elements of stacks, stores and configurations respectively:

$$\begin{array}{rcl}
\{\!|x \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]|\!\} & \triangleq & \mathsf{OB}\langle \iota_1 \ldots \iota_n, x\rangle \\
\{\!|\iota \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle|\!\} & \triangleq & (\nu \mathsf{dom}(\mathcal{S}))(\{\!|\mathcal{S}|\!\} \mid !\iota(x,r).\{\!|b|\!\}_r) \\
\{\!|\sigma \cdot \mathcal{S} \vdash a|\!\}_p & = & (\nu \mathsf{dom}(\sigma))(\{\!|\sigma|\!\} \mid (\nu \mathsf{dom}(\mathcal{S}))(\{\!|\mathcal{S}|\!\} \mid \{\!|a|\!\}_p))
\end{array}$$

Unfortunately, this does not work due to sharing and method override. To see the problem, consider the configuration

$$\sigma \cdot \mathcal{S} \vdash [\mathrm{l}_1{=}\varsigma(x)b,\ \mathrm{l}_2{=}\varsigma(x)(y.\mathrm{l}_j \Leftarrow \varsigma(x)b'; x)].\mathrm{l}_2$$

with $\mathcal{S} = y \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]$. When evaluating this expression, we have the following configuration $\sigma' \cdot \mathcal{S}, x \mapsto [\mathrm{l}_1{=}\iota_1,\ \mathrm{l}_2{=}\iota_2] \vdash y.\mathrm{l}_j \Leftarrow \varsigma(x)b'; x$ with $\sigma' = \sigma, \iota_1 \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle, \iota_2 \mapsto \langle \varsigma(x)y.\mathrm{l}_j \Leftarrow \varsigma(x)b'; x, \mathcal{S}\rangle$. Now, in our proposed translation, only the object manager in the currently active stack is updated, the object managers hidden in the store will remain the same and refer to the original l_j method, so the method update will only occur locally.

We solve this problem by using instead the factorized encoding $[\![-]\!]$ of Section 6.4.

6.5.3 Operational correctness of the factorized encoding

In the factorized encoding, stores and stacks can be translated as proposed above; a store binds closures to locations, so here we locate a cell at the location holding the address of the method closure (which is the translation of a method together with a private stack). One can consider a cell as implementing a single entry of a store — A cell is located at some name ι and has a contents which can be read and overridden.

$$\begin{array}{rcl}
[\![\emptyset]\!] & \triangleq & \mathbf{0} \\
[\![\iota \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle, \sigma]\!] & \triangleq & (\nu \mathsf{dom}(\mathcal{S}), m)(\mathsf{Cell}\langle \iota, m\rangle \mid !m(x,r).[\![b]\!]_r \mid [\![\mathcal{S}]\!]) \mid [\![\sigma]\!]
\end{array}$$

A stack binds values to variables, so we translate it as an object manager located on the variable name.

$$\begin{array}{rcl}
[\![\emptyset]\!] & \triangleq & \mathbf{0} \\
[\![x \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}], \mathcal{S}]\!] & \triangleq & \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle \mid [\![\mathcal{S}]\!]
\end{array}$$

We have two types of configurations in the operational semantics of the Imperative ς-calculus; initial configurations on the form $\sigma \cdot \mathcal{S} \vdash a$ and final configurations on the form $v \cdot \sigma$. They are translated as follows:

$$\begin{array}{rcl} [\![\sigma \cdot \mathcal{S} \vdash a]\!]_p & \triangleq & (\nu \mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu \mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![a]\!]_p)) \\ [\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma]\!]_p & \triangleq & (\nu s, \mathsf{dom}(\sigma))(\overline{p}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid [\![\sigma]\!]) \end{array}$$

That is, in both cases, we simply translate each component of the configurations and appropriately hide their access.

Before we start on the "real" proof of operational correctness, we need the following definition and lemma, stating that the translation of configurations in the semantics of the Imperative ς-calculus can always be transformed in a normal form. The reason why we do this, is because then we know exactly which part of the translation that can bring the system to evolve.

In the following we shall, unless otherwise mentioned, assume that when we consider configurations $\sigma \cdot \mathcal{S} \vdash a$ and $[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma$ then they are well-formed. This is, $\sigma \cdot \mathcal{S} \vdash \diamond$, $\mathsf{fv}(a) \subseteq \mathsf{dom}(\mathcal{S})$, and $\iota_1, \ldots, \iota_n \in \mathsf{dom}(\sigma)$. Below, $\sim_b$ is the strong version of barbed congruence.

Definition 6.8 *Let $\mathcal{P}_q^p$ be given by the following grammar (where p and q binds names in terms generated by the grammar):*

$$\begin{array}{rcl} \mathcal{P}_q^p & ::= & q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p \\ & \mid & q(x).\overline{x}\mathsf{clone}\_p \\ & \mid & (\nu \mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid q(x).(\nu b)(\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle p, b\rangle \mid !b(x,r).[\![b]\!]_r)) \\ & \mid & (\nu \mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid q(x).[\![b]\!]_p) \end{array}$$

Let $\mathcal{C}_q^p$ be given by: $\mathcal{C}_q^p ::= (\nu r)(\mathcal{P}_q^r \mid \mathcal{C}_r^p) \mid q \rhd p$, where is a destructive link from q to p defined by $q \rhd p = q(x).\overline{p}x$.

In the above definition $\mathcal{P}_q^p$ represents single actions that can happen to an object and $\mathcal{C}_p^q$ represents a sequential list of actions. The reason why the two last cases in the definition of $\mathcal{P}_q^p$ contains the translation of a stack $\mathcal{S}$, is because method update and the let construct both includes an object, that needs a stack.

The following lemma states that the translation of any configuration $\sigma \cdot \mathcal{S} \vdash a$ is barbed congruent to the translation of an object value $[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]$, an extension of the store σ and a $\mathcal{C}_p^q$.

Lemma 6.9 *For all σ, $\mathcal{S}$, and a, there exists a $\mathcal{C}^p_{q'}$ and a store σ' with $\sigma \subseteq \sigma'$ and $\iota_1, \ldots \iota_n \subseteq \mathsf{dom}(\sigma')$ such that*

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)(((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![a]\!]_q) \mid q \rhd p)) \\ \sim_b & (\nu\mathsf{dom}(\sigma'))([\![\sigma']\!] \mid (\nu q')((\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \overline{q}'s) \mid \mathcal{C}^p_{q'})) \end{array}$$

Proof. Straightforward structural induction in a for the more general case:

$$(\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![a]\!]_q) \mid \mathcal{C}^p_q\Big)\Big) \tag{6.3}$$

using the distributivity of the functional processes that represent object managers in $\mathcal{S}$. Below, we show a few of the cases.

$a = [\mathrm{l}_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$: The translation of (6.3) is:

$$\begin{array}{l} (\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid (\nu s)(\overline{q}s \mid \\ (\nu\tilde{\iota})(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \prod_{i\in 1..n}(\nu b_i)\underbrace{(\mathsf{Cell}\langle \iota_i, b_i\rangle \mid !b_i(x_i, r).[\![b_i]\!]_r)}_{P_i})))) \mid \mathcal{C}^p_q\Big)\Big) \end{array}$$

We can now distribute $\mathcal{S}$ into each $(\nu b_i)(P_i)$ and get $(\nu\mathsf{dom}(\mathcal{S}), b_i)(P_i \mid [\![\mathcal{S}]\!])$ which is the translation of a store σ''. If we assume that $\{\iota_1, \ldots, \iota_n\} \cap \mathsf{dom}(\sigma) = \emptyset$ we "move" σ'' next to σ and get:

$$\begin{array}{l} (\nu\mathsf{dom}(\sigma), \iota_1, \ldots, \iota_n)\Big([\![\sigma]\!] \mid \prod_{i\in 1..n}(\nu\mathsf{dom}(\mathcal{S}), b_i)(P_i \mid [\![\mathcal{S}]\!]) \mid \\ (\nu q)\Big((\nu s)(\overline{q}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle) \mid \mathcal{C}^p_q\Big)\Big) \end{array}$$

By letting $\sigma' = \sigma\sigma''$, $\mathcal{C}^p_{q'} = \mathcal{C}^p_q$, and $q' = q$ we get the result.

$a = x$: The translation of (6.3) is:

$$(\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid \overline{q}x) \mid \mathcal{C}^p_q\Big)\Big)$$

By assumption $x \in \mathsf{dom}(\mathcal{S})$, so there is an object manager $\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle$ in $[\![\mathcal{S}]\!]$. Since $\mathcal{S} \vdash \diamond$ we have

$$(\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid \overline{q}x) \sim_b (\nu x)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle \mid \overline{q}x)$$

and since $\sim_b$ is a congruence we have the result.

$a = \mathsf{let}\ x{=}b\ \mathsf{in}\ c$: The translation of (6.3) is:

$$(\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid (\nu r)([\![b]\!]_r \mid r(x).[\![c]\!]_q) \mid \mathcal{C}^p_q\Big)\Big)$$

We can use the functional nature of $\mathcal{S}$ and distribute it among b and c, getting

$$(\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu r)((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![b]\!]_r) \mid (\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid r(x).[\![c]\!]_q)) \mid \mathcal{C}_q^p\Big)\Big)$$

We can now build a new $\mathcal{C}_r^p = (\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid r(x).[\![c]\!]_q) \mid \mathcal{C}_q^p$. By induction there exists a $\mathcal{C}_{q'}^p$, σ', $\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle$ such that

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))\Big([\![\sigma]\!] \mid (\nu q)\Big((\nu r)((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![b]\!]_r) \mid \mathcal{C}_r^p\Big)\Big) \\ \sim_b & (\nu\mathsf{dom}(\sigma'))([\![\sigma']\!] \mid (\nu q')((\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \overline{q}'s) \mid \mathcal{C}_{q'}^p)) \end{array}$$

□

We can now prove the operational correctness of the factorized encoding. The following lemma describes how each operation: method activation, cloning, method update, and let, on a object value is mimicked by a series of deterministic reductions in the translation.

Lemma 6.10 *For all stores σ, stacks $\mathcal{S}$, and objects a, with $\sigma \cdot \mathcal{S} \vdash \diamond$ and $\mathsf{fv}(a) \subseteq \mathsf{dom}(\mathcal{S})$, the following holds:*

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]_q \mid (\nu p)(q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p \mid \mathcal{C}_p^{p^*}))) \\ \xrightarrow{\tau}{}_d^{+} \sim_b & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu p)((\nu\mathsf{dom}(\mathcal{S}), x_j)([\![b_j]\!]_p \mid \\ & [\![\mathcal{S}, x_j \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]) \mid \mathcal{C}_p^{p^*})) \end{array}$$

with $\sigma(\iota_j) = \langle \varsigma(x_j)b_j, \mathcal{S}\rangle$ and $j \in 1..n$

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]_q \mid (\nu p)(q(x).\overline{x}\mathsf{clone}\_p \mid \mathcal{C}_p^{p^*}))) \\ \xrightarrow{\tau}{}_d^{+} \sim_b & (\nu\mathsf{dom}(\sigma), \iota_1' \ldots \iota_n')([\![\sigma, \iota_i' \mapsto \sigma(\iota_i)\ ^{i\in 1..n}]\!] \mid \\ & (\nu p)((\nu s')(\overline{p}s' \mid \mathsf{OB}\langle \iota_1' \ldots \iota_n', s'\rangle) \mid \mathcal{C}_p^{p^*})) \end{array}$$

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]_q \mid (\nu p)(\\ & (\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid q(x).(\nu b)(\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle b, p\rangle \mid !b(x, r).[\![b]\!]_r)) \mid \mathcal{C}_p^{p^*}))) \\ \xrightarrow{\tau}{}^{+} \sim_b & (\nu\mathsf{dom}(\sigma))([\![\sigma[\iota_j \mapsto \langle \varsigma(x)b, \mathcal{S}\rangle]]\!] \mid \\ & (\nu p)((\nu s)(\overline{p}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle) \mid \mathcal{C}_p^{p^*})) \end{array}$$

with $j \in 1..n$

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]_q \mid (\nu p)(\\ & (\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid q(x).[\![b]\!]_p) \mid \mathcal{C}_p^{p^*}))) \\ \xrightarrow{\tau}{}_d \sim_b & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu p)((\nu\mathsf{dom}(\mathcal{S}), x)([\![b]\!]_p \mid \\ & [\![S, x \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]) \mid \mathcal{C}_p^{p^*})) \end{array}$$

PROOF. By inspection of the possible transitions for each case, here we only consider the first.

$$
\begin{array}{rl}
 & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]_q \mid (\nu p)(q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p \mid \mathcal{C}_p^{p^*}))) \\
= & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)((\nu s)(\overline{q}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle) \mid \\
 & (\nu p)(q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_p \mid \mathcal{C}_p^{p^*}))) \\
\xrightarrow{\tau}_d & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \\
 & (\nu p)(\overline{s}\mathrm{l}_j\_\mathsf{sel}\_p \mid \mathcal{C}_p^{p^*}))) \\
\xrightarrow{\tau}{}^{3}_d & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \\
 & (\nu p, g)(\overline{\iota_j}\mathsf{read}\_g.g(m).\overline{m}\langle s, p\rangle \mid \mathcal{C}_p^{p^*})))
\end{array}
$$

Now the object manager has started a process that will read the contents of the cell located at ι_j. By assumption there exist a $\iota_j \in \sigma$, with $\sigma(\iota_j) = \langle \varsigma(x_j)b_j, \mathcal{S}\rangle$. The translation $\sigma(\iota_j)$ is $(\nu b_j, \mathsf{dom}(\mathcal{S}))(\mathsf{Cell}\langle \iota_j, b_j\rangle \mid !b_j(x,r).[\![b_j]\!]_r \mid [\![\mathcal{S}]\!])$. We can now use the functional nature of $[\![\mathcal{S}]\!]$ and $!b_j(x,r).[\![b_j]\!]_r$ to create private copies for the activation of the method b_j.

$$
\begin{array}{rl}
\xrightarrow{\tau}{}^{2}_d{\sim_b} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid (\nu p, g)((\nu b_j)(\overline{g}b_j \mid \\
 & (\nu\mathsf{dom}(\mathcal{S}))(!b_j(x,r).[\![b_j]\!]_r \mid [\![\mathcal{S}]\!]) \mid g(m).\overline{m}\langle s, p\rangle) \mid \mathcal{C}_p^{p^*}))) \\
\xrightarrow{\tau}_d{\sim_b} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid (\nu p)((\nu b_j)(\\
 & (\nu\mathsf{dom}(\mathcal{S}))(!b_j(x,r).[\![b_j]\!]_r \mid [\![\mathcal{S}]\!]) \mid \overline{b_j}\langle s, p\rangle) \mid \mathcal{C}_p^{p^*})))
\end{array}
$$

After having activated the method b_j, we can garbage collect $!b_j(x,r).[\![b_j]\!]_r$ and get the result.

$$
\begin{array}{rl}
\xrightarrow{\tau}_d{\sim_b} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu s)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid (\nu p)(\\
 & \quad (\nu\mathsf{dom}(\mathcal{S}))([\![b_j]\!]_p\{{}^{s}/_{x_j}\} \mid [\![\mathcal{S}]\!])) \mid \mathcal{C}_p^{p^*})) \\
\sim_b & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu p)((\nu\mathsf{dom}(\mathcal{S}), x_j)([\![b_j]\!]_p \mid [\![\mathcal{S}]\!] \\
 & \quad \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x_j\rangle) \mid \mathcal{C}_p^{p^*})) \\
= & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu p)((\nu\mathsf{dom}(\mathcal{S}), x_j)([\![b_j]\!]_p \mid \\
 & \quad [\![\mathcal{S}, x_j \mapsto [\mathrm{l}_i{=}\iota_i\ ^{i\in 1..n}]]\!]) \mid \mathcal{C}_p^{p^*}))
\end{array}
$$

□

After we have handled the basic operations on object values, we can now proceed by showing how the operational semantics of the Imperative ς-calculus is mimicked by the translation.

Lemma 6.11 *If $\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v \cdot \sigma'$ then:*

$$\begin{array}{rl} & (\nu\mathsf{dom}(\sigma))([\![\sigma]\!] \mid (\nu q)((\nu\mathsf{dom}(\mathcal{S}))([\![\mathcal{S}]\!] \mid [\![a]\!]_q) \mid \mathcal{C}^p_q)) \\ \xrightarrow{\tau}{}^*_d & (\nu\mathsf{dom}(\sigma'))([\![\sigma']\!] \mid (\nu q)([\![v]\!]_q \mid \mathcal{C}^p_q)) \end{array}$$

Proof. We prove the lemma using induction in the structure of the inference of $\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v \cdot \sigma'$. To increase readability, we let $(\nu\sigma)$ denote $(\nu\mathsf{dom}(\sigma))$ and $(\nu\mathcal{S})$ denote $(\nu\mathsf{dom}(\mathcal{S}))$.

(VarI): By the rule (VarI), the stack $\mathcal{S}$ must be of the form $\mathcal{S}', x \mapsto [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]$. Using the strong bisimulation congruence and the fact $(\nu p)(!p(x).P) \sim_b \mathbf{0}$ we get:

$$\begin{array}{rl} & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S}', x)([\![\mathcal{S}']\!] \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle \mid \overline{q}x) \mid \mathcal{C}^p_q)) \\ \sim_b & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu x)(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, x\rangle \mid \overline{q}x) \mid \mathcal{C}^p_q)) \\ = & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_q \mid \mathcal{C}^p_q)) \end{array}$$

(ObjI): Following the encoding of objects we have:

$$\begin{array}{l} (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid (\nu\iota_1 \ldots \iota_n, s)(\overline{q}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \mid \\ \prod_{i\in 1..n}(\nu b_i)(\mathsf{Cell}\langle \iota_i, b_i\rangle \mid !b_i(x_i, r).[\![b_i]\!]_r)) \mid \mathcal{C}^p_q)) \end{array} \tag{6.4}$$

Since stacks are functional we can distribute the stack to each method, so expression (6.4) must be bisimilar to:

$$\begin{array}{rl} & (\nu\sigma, \iota_1 \ldots \iota_n)([\![\sigma]\!] \mid \prod_{i\in 1..n}(\nu b_i)(\mathsf{Cell}\langle \iota_i, b_i\rangle \mid (\nu\mathcal{S})([\![\mathcal{S}]\!] \mid \\ & \quad !b_i(x_i, r).[\![b_i]\!]_r)) \mid (\nu q)((\nu s)(\overline{q}s \mid \mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle) \mid \mathcal{C}^p_q)) \\ = & (\nu\sigma')([\![\sigma']\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_q \mid \mathcal{C}^p_q)) \end{array}$$

with $\sigma' = \sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, \mathcal{S}\rangle$ for all $i \in 1..n$.

(SelI): By the encoding of method activation we have:

$$\begin{array}{rl} & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid (\nu q')([\![a]\!]_{q'} \mid q'(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_q)) \mid \mathcal{C}^p_q)) \\ \sim_b & (\nu\sigma)([\![\sigma]\!] \mid (\nu q')((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid [\![a]\!]_{q'}) \mid (\nu q)(q(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_q \mid \mathcal{C}^p_q))) \end{array}$$

By induction this reduces to:

$$(\nu\sigma')([\![\sigma']\!] \mid (\nu q')([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_{q'} \mid (\nu q)(q'(x).\overline{x}\mathrm{l}_j\_\mathsf{sel}\_q \mid \mathcal{C}^p_q)))$$

which again by Lemma 6.10 reduces to:

$$(\nu\sigma')([\![\sigma']\!] \mid (\nu q)((\nu\mathcal{S}', x_j \mapsto [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}])([\![\mathcal{S}']\!] \mid [\![b_j]\!]_q) \mid \mathcal{C}_q^p))$$
$$\text{with } \sigma'(\iota_j) = \langle \varsigma(x_j) b_j, \mathcal{S}' \rangle$$

And again we can use the induction hypothesis to deduce that this reduces to something bisimilar to:

$$(\nu\sigma'')([\![\sigma'']\!] \mid (\nu q)([\![v]\!]_q \mid \mathcal{C}_q^p))$$

(UpdI): By the encoding of method update and distributivity of the stack we have:

$$\begin{array}{rl} & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid (\nu q')([\![a]\!]_{q'} \mid \\ & \quad q'(x).(\nu b)\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle b, q\rangle.!b(x,r).[\![b]\!]_r)) \mid \mathcal{C}_q^p)) \\ \sim_b & (\nu\sigma)([\![\sigma]\!] \mid (\nu q')((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid [\![a]\!]_{q'}) \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid \\ & \quad q'(x).(\nu b)(\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle b, q\rangle \mid !b(x,r).[\![b]\!]_r)) \mid \mathcal{C}_q^p))) \end{array}$$

By induction this reduces to something bisimilar to:

$$\begin{array}{l} (\nu\sigma')([\![\sigma']\!] \mid (\nu q')([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_{q'} \mid (\nu q)(\\ \quad (\nu\mathcal{S})([\![\mathcal{S}]\!] \mid q'(x).(\nu b)(\overline{x}\mathrm{l}_j\_\mathsf{upd}\_\langle b, q\rangle \mid !b(x,r).[\![b]\!]_r)) \mid \mathcal{C}_q^p))) \end{array}$$

which by Lemma 6.10 reduces to:

$$(\nu\sigma')([\![\sigma'[\iota_j \mapsto \langle \varsigma(x) b, \mathcal{S}\rangle]]\!] \mid (\nu q)([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_q \mid \mathcal{C}_q^p))$$

(CloneI): By the encoding of cloning we get:

$$\begin{array}{rl} & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid (\nu q')([\![a]\!]_{q'} \mid q'(x).\overline{x}\mathsf{clone}\_q) \mid \mathcal{C}_q^p)) \\ \equiv & (\nu\sigma)([\![\sigma]\!] \mid (\nu q')((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid [\![a]\!]_{q'}) \mid (\nu q)(q'(x).\overline{x}\mathsf{clone}\_q \mid \mathcal{C}_q^p))) \end{array}$$

By induction this reduces to something bisimilar to:

$$(\nu\sigma')([\![\sigma']\!] \mid (\nu q')([\![[\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}]]\!]_{q'} \mid (\nu q)(q'(x).\overline{x}\mathsf{clone}\_q \mid \mathcal{C}_q^p)))$$

which by Lemma 6.10 reduces to:

$$(\nu\sigma', \iota'_1 \ldots \iota'_n)([\![\sigma', \iota'_i \mapsto \sigma'(\iota_i) \ ^{i\in 1..n}]\!] \mid (\nu q)([\![[\mathrm{l}_i = \iota'_i \ ^{i\in 1..n}]]\!]_q \mid \mathcal{C}_q^p))$$

(LetI): By the encoding of the let construct and distributivity of the stack we have:

$$\begin{array}{rl} & (\nu\sigma)([\![\sigma]\!] \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid (\nu q')([\![a]\!]_{q'} \mid q'(x).[\![b]\!]_q) \mid \mathcal{C}_q^p)) \\ \sim_b & (\nu\sigma)([\![\sigma]\!] \mid (\nu q')((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid [\![a]\!]_{q'}) \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid q'(x).[\![b]\!]_q) \mid \mathcal{C}_q^p))) \end{array}$$

By induction this reduces to something bisimilar to:

$$(\nu\sigma')([\![\sigma']\!] \mid (\nu q')([\![v]\!]_{q'} \mid (\nu q)((\nu\mathcal{S})([\![\mathcal{S}]\!] \mid q'(x).[\![b]\!]_q) \mid \mathcal{C}_q^p)))$$

which by Lemma 6.10 is reduces to:

$$(\nu\sigma')([\![\sigma']\!] \mid (\nu q)((\nu\mathcal{S}, x)([\![b]\!]_p \mid [\![S, x \mapsto v]\!]) \mid \mathcal{C}_q^p))$$

and again by induction we know that this reduces to something bisimilar to:

$$(\nu\sigma'')([\![\sigma'']\!] \mid (\nu q)([\![v']\!]_q \mid \mathcal{C}_q^p))$$

□

As an immediate consequence of Lemma 6.11 we get the following corollary:

Corollary 6.12 *If $\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v \cdot \sigma'$ then: $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p \Longrightarrow_d \sim_b [\![v \cdot \sigma']\!]_p$.*

Lemma 6.13 *If $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p{\downarrow_q}$, then a is either an object (value) or a variable.*

Proof. By inspection of the encoding. □

Lemma 6.14 *Let $\mathcal{R} = \{[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p \mid \sigma \cdot \mathcal{S} \vdash a{\Uparrow},\ \sigma \cdot \mathcal{S} \vdash a$ well-typed$\}$. Then for all $P \in \mathcal{R}$ there exists a P' such that $P \xrightarrow{\tau}\Longrightarrow_d \sim_b P'$ with $P' \in \mathcal{R}$*

Proof. We argue by induction of the length of a with a case analysis of the structure of a. The length on a term is defined the obvious way.

For terms with length 0, the result follows trivially, as there are no terms with length 0. Now consider terms with length n, in principle a term can be on the following forms:

x: Is always convergent.

$[\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]$: Is always convergent.

$b.\mathrm{l}_j$: If $\sigma \cdot \mathcal{S} \vdash b.\mathrm{l}_j$ diverges it can be for one of two reasons; either:

- $\sigma \cdot \mathcal{S} \vdash b$ diverges, and by induction there exists a Q' such that $[\![\sigma \cdot \mathcal{S} \vdash b]\!]_p \xrightarrow{\tau}{}^{+}_d \sim_b Q'$ with $Q' \in \mathcal{R}$. Since $Q' \in \mathcal{R}$ this means that Q' is of the form $[\![\sigma' \cdot \mathcal{S}' \vdash b']\!]_p$. Therefore will $\sigma' \cdot \mathcal{S}' \vdash b'.\mathrm{l}_j$ also diverge and its encoding be contained in $\mathcal{R}$.

- $\sigma \cdot \mathcal{S} \vdash b \rightsquigarrow [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}] \cdot \sigma'$ and $\sigma' \cdot \mathcal{S}', x \mapsto [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}] \vdash c$ diverges, where $\sigma'(\iota_j) = \langle \varsigma(x)c, \mathcal{S}' \rangle$. By Lemma 6.11 and 6.10 we have $[\![\sigma \cdot \mathcal{S} \vdash b.\mathrm{l}]\!]_p \xrightarrow{\tau}{}^{+}_{d} \sim_b [\![\sigma' \cdot \mathcal{S}', x \mapsto [\mathrm{l}_i{=}\iota_i \ ^{i\in 1..n}] \vdash c]\!]_p$ which must be contained in $\mathcal{R}$.

$b.\mathrm{l}{\Leftarrow}\varsigma(x)c$: If $\sigma \cdot \mathcal{S} \vdash b.\mathrm{l}{\Leftarrow}\varsigma(x)c$ diverges is must be because $\sigma \cdot \mathcal{S} \vdash b$ diverges, and by induction there exists a Q' such that $[\![\sigma \cdot \mathcal{S} \vdash b]\!]_p \xrightarrow{\tau}{}^{+}_{d} \sim_b Q'$ with $Q' \in \mathcal{R}$. Since $Q' \in \mathcal{R}$ this must mean that Q' is on the form $[\![\sigma' \cdot \mathcal{S}' \vdash b']\!]$. Therefore will $\sigma' \cdot \mathcal{S}' \vdash b'.\mathrm{l}{\Leftarrow}\varsigma(x)c$ also diverge and its encoding be contained in $\mathcal{R}$.

$\mathsf{clone}(b)$: Same argument as in the previous case.

$\mathsf{let}\ x{=}b\ \mathsf{in}\ c$: If $\sigma \cdot \mathcal{S} \vdash \mathsf{let}\ x{=}b\ \mathsf{in}\ c$ diverges, then it is either because:

- $\sigma \cdot \mathcal{S} \vdash b$ diverges. By induction there exists a Q' such that $[\![\sigma \cdot \mathcal{S} \vdash b]\!]_p \xrightarrow{\tau}{}^{+}_{d} Q'$ with $Q' \in \mathcal{R}$. This means that Q' is on the form $[\![\sigma' \cdot \mathcal{S}' \vdash b']\!]$ and $\sigma' \cdot \mathcal{S}' \vdash b'$ diverges. Therefore will $\sigma' \cdot \mathcal{S}' \vdash \mathsf{let}\ x{=}b'\ \mathsf{in}\ c$ also diverge and its encoding be contained in $\mathcal{R}$.

- $\sigma \cdot \mathcal{S} \vdash b \rightsquigarrow v \cdot \sigma'$ and $\sigma' \cdot \mathcal{S}, x \mapsto v \vdash c$ diverges. By Lemma 6.11 and a reduction, we get $[\![\sigma \cdot \mathcal{S} \vdash \mathsf{let}\ x{=}b\ \mathsf{in}\ c]\!]_p \xrightarrow{\tau}{}^{+}_{d} [\![\sigma' \cdot \mathcal{S}, x \mapsto v \vdash b]\!]_p \in \mathcal{R}$

□

Theorem 6.15 *Suppose $\sigma \cdot \mathcal{S} \vdash a$ is well-typed. If $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p \Downarrow_p$, there exists a value v, and a store σ', s.t. $\sigma \cdot \mathcal{S} \vdash a \rightsquigarrow v \cdot \sigma'$*

Proof. We argue by contradiction. Assume that $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p \Downarrow$ converges but $\sigma \cdot \mathcal{S} \vdash a$ diverges. But then by Lemma 6.14 $[\![\sigma \cdot \mathcal{S} \vdash a]\!]_p$ should also diverge, which contradicts our initial assumption. □

In the above statements of operational correspondence, we often require that the configurations are well-typed. This is because the divergence predicate $\Uparrow$ does not distinguish between "real" divergence and run-time errors. For instance, we have $\emptyset \cdot \emptyset \vdash [\,].\mathrm{l} \Uparrow$, although one would not claim that this term diverges. But if we only consider well-typed terms, such cases are ruled out.

Another possibility would have been to work with a small-step semantics, like the one Gordon, Hankin and Lassen [GHL97] give to the Imperative ς-calculus, where it is possible to define divergence of a term as having an infinite reduction sequence.

6.5.4 Relating the original and the factorized encodings

In this section, we study the correctness of the transformation that leads us from the original encoding of Table 6.10 to the factorized one of Section 6.4.

We have not been able to prove that the translations of an imperative object according to the factorized encoding $[\![-]\!]$ and to the original encoding $\{\!|-|\!\}$ are equated by some known behavioral equivalence of the π-calculus — the factorized encoding yields richer behaviors.

The transformation only affects the object manager. So, for instance, one might hope to prove that the original object manager and its transformed are equated by one of the known notions of equivalence for the π-calculus, i.e. proving:

$$\mathsf{OB}\langle b_1 \dots b_n, s\rangle = (\nu \iota_1 \dots \iota_n)\Big(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b\rangle\Big)$$

But this is not true because the process on the right-hand side has a richer behavior, as shown by the following example.

Example 6.6 The functional object manager can perform the following transition sequence, which cannot be performed by the imperative counterpart:

$$\xrightarrow{sl_1\_\mathsf{sel}\_p} \quad \xrightarrow{sl_1\_\mathsf{upd}\_\langle p',b'\rangle} \quad \Longrightarrow \quad \xrightarrow{\overline{b'}\langle s,p\rangle}$$

What can happen is that a request for a method update can overtake a request for selection of that same method, resulting in the activation of the new method. This can happen because the functional object manager is a replicated process that is always ready to accept new request even if it has not finished updating or reading its cells. Observe, this is only a problem for the proof of operational correctness — the Imperative ς-calculus is sequential, so π-calculus terms generated by the factorized translation cannot create cases like this. □

The problem is that the original object manager responds in a different manner to concurrent requests than the functional. However, the contexts in which object managers work are sequential, so that should not really make a difference. What we need to do is to ensure that the original object manager has enough behavior to function correctly.

We solve this problem by extending the notion of *ready simulation* [BIM95, LS91] (in the latter paper called 2/3-simulation) to the π-calculus and prove

that the two translations of an object are in a ready simulation relation. This is a rather weak result, but, together with the correctness of the factorized encoding, it will suffice to prove the operational correctness of the original encoding. (As far as we know, ready simulation has never been defined in a weak version and never been used in calculi of mobile processes.)

Definition 6.16 (Ready Simulation) *A relation $\mathcal{R}$ is a (weak)* ready simulation *if $P\ \mathcal{R}\ Q$ implies, for any μ (with bound names of μ not free in P, Q):*

i. If $P \xrightarrow{\mu} P'$, then there exists a Q' s.t. $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$.

ii. If $Q \xrightarrow{\mu} Q'$, then there exist a P' s.t. $P \stackrel{\mu}{\Longrightarrow} P'$

We say that Q ready-simulates *P, written $P \prec Q$, if $P\ \mathcal{R}\ Q$ for some ready simulation $\mathcal{R}$.*

Observe that, in the above untyped relation, types are ignored, and therefore in the derivatives P', Q' there could be run-time errors. (We could also have a typed version of ready simulation, but the untyped version will suffice for our purpose). The only problem is what happens on input of something that is not well typed. We have two cases to consider.

- Input of a value that results in a grammatically incorrect process, which by the way substitution is defined, will result in the term wrong; this term has no transitions and will therefore be ready similar to $\mathbf{0}$. Intuitively this is acceptable in the untyped case, as the input of something resulting in a grammatically incorrect term should stop the system.

- Input of a value that results in grammatically correct terms, but leads to a runtime error later. But this does not matter since run-time errors are signaled as wrong-actions; therefore for two processes to be ready similar they must have the same behavior w.r.t. run-time errors.

Furthermore, we shall only compare well-typed processes where run-time errors are guaranteed not to occur.

Observe, in clause ii of Definition 6.16 we require that if $Q \xrightarrow{\mu} Q'$ then $P \stackrel{\mu}{\Longrightarrow} P'$. One might have expected $P \stackrel{\hat{\mu}}{\Longrightarrow} P'$, but if we did that then P could match a τ-move by Q by deadlocking, which would make the relation unfit for our use.

In Appendix A.1 we establish some basic theory for ready simulation. We prove (Theorem A.2) that ready simulation is a precongruence w.r.t. parallel composition and restriction, and show the soundness (Theorem A.4)

of the powerful proof technique of up-to restriction and parallel composition (Definition A.3). With these results at hand we are now able to prove the key result for the relation between the original and the factorized translation of the Imperative ς-calculus.

Theorem 6.17

$$\mathsf{OB}\langle b_1 \dots b_n, s\rangle \prec (\nu \iota_1 \dots \iota_n)\Big(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \;\Big|\; \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle\Big)$$

Proof. Let

$$\mathcal{R} = \left\{ \left(\mathsf{OB}\langle b_1 \dots b_n, s\rangle, (\nu \iota_1 \dots \iota_n)\Big(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \;\Big|\; \textstyle\prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle\Big)\right) \;\middle|\; \forall b_1, \dots, b_n, s \in \mathbf{Names} \right\}$$

We claim that $\mathcal{R}$ is an up-to context ready simulation. That together with Theorem A.4 proves the theorem.

If $\mathsf{OB}\langle b_1 \dots b_n, s\rangle \xrightarrow{\mu} S$ there are 3 possibilities:

i. $\mu = sl_j\_\mathsf{sel}\_p$ and $S = \mathsf{OB}\langle b_1 \dots b_n, s\rangle \mid \overline{b}\langle s, p\rangle$

ii. $\mu = sl_j\_\mathsf{upd}\_\langle p, b\rangle$ and $S = \mathsf{OB}\langle b_1 \dots b \dots b_n, s\rangle \mid \overline{p}s$

iii. $\mu = s\mathsf{clone}\_p$ and $S = \mathsf{OB}\langle b_1 \dots b_n, s\rangle \mid (\nu s')(\mathsf{OB}\langle b_1 \dots b_n, s'\rangle \mid \overline{p}s')$

In each case it is easy to determine that this can be matched in the required way by $(\nu \iota_1 \dots \iota_n)(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle)$.

For instance in case iii we have

$$\begin{array}{ll} & (\nu \iota_1 \dots \iota_n)(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle) \\ \stackrel{\mu}{\Longrightarrow} & (\nu \iota_1 \dots \iota_n)(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle) \mid \\ & \quad (\nu s')((\nu \iota'_1 \dots \iota'_n)(\mathsf{OB}_f\langle \iota'_1 \dots \iota'_n, s'\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota'_i, b_i\rangle) \mid \overline{p}s') \end{array}$$

Since we only need to show that $\mathcal{R}$ is an up-to context ready simulation, we can cancel the common parallel component and restriction and only need to check that

$$\mathsf{OB}\langle b_1 \dots b_n, s\rangle \;\mathcal{R}\; (\nu \iota_1 \dots \iota_n)(\mathsf{OB}_f\langle \iota_1 \dots \iota_n, s\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle)$$

and

$$\mathsf{OB}\langle b_1 \dots b_n, s'\rangle \;\mathcal{R}\; (\nu \iota'_1 \dots \iota'_n)(\mathsf{OB}_f\langle \iota'_1 \dots \iota'_n, s'\rangle \mid \prod_{i\in 1..n} \mathsf{Cell}\langle \iota'_i, b_i\rangle)$$

which is easily seen to be true.

Conversely, a transition

$$(\nu\iota_1 \ldots \iota_n)\Big(\mathsf{OB}_f\langle \iota_1 \ldots \iota_n, s\rangle \;\Big|\; \prod_{i\in 1..n} \mathsf{Cell}\langle \iota_i, b_i\rangle\Big) \xrightarrow{\mu}$$

is easily matched as μ must be one three possible actions of $\mathsf{OB}\langle b_1 \ldots b_n, s\rangle$. □

This is a rather weak result, but, together with the correctness of the factorized encoding, it suffices to prove the operational correctness of the original encoding.

From the previous theorems, we can conclude that the original encoding is ready simulated by the factorized one:

Corollary 6.18 $\{\!|a|\!\}_p \prec [\![\emptyset \cdot \emptyset \vdash a]\!]_p$.

Proof. Follows from Theorem 6.17 and Theorem A.2. □

Lemma 6.19 *Suppose a is well-typed. Then $\{\!|a|\!\}_p$ cannot deadlock, i.e. whenever $\{\!|a|\!\}_p \Rightarrow P$ then there are μ, P' s.t. $P \xrightarrow{\mu} P'$.*

Proof. Assume that $\{\!|a|\!\}_p$ can deadlock, that is there exists a P such that $\{\!|a|\!\}_p \Longrightarrow P \not\xrightarrow{}$. Now since $\{\!|a|\!\} \prec [\![\emptyset \cdot \emptyset \vdash a]\!]_p$, there must exist a Q such that $[\![\emptyset \cdot \emptyset \vdash a]\!]_p \Longrightarrow Q$ with $P \prec Q$.

By Corollary 6.12, if a converges, then $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ will also converge, and by Lemma 6.14, if a diverges, then $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ will also diverge; so $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ cannot deadlock. Then there must exist a μ s.t. $Q \xrightarrow{\mu}$. By the second clause of the definition of ready simulation, we infer $P \xRightarrow{\mu}$, which is a contradiction. □

Corollary 6.20 *Suppose a is well-typed.*

- $\{\!|a|\!\}_p\Uparrow$ *iff* $[\![\emptyset \cdot \emptyset \vdash a]\!]_p\Uparrow$
- $\{\!|a|\!\}_p\Downarrow$ *iff* $[\![\emptyset \cdot \emptyset \vdash a]\!]_p\Downarrow$

Proof. Both $\{\!|a|\!\}_p$ and $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ are deadlock-free. Then the corollary follows from Corollary 6.18 and the fact that $[\![\emptyset \cdot \emptyset \vdash a]\!]_p$ either converges or diverges (it cannot do both, because of Corollary 6.12 and Lemma 6.14). □

6.5.5 Adequacy and soundness of the original interpretation

From Corollary 6.12, Theorem 6.15 and Corollary 6.20, we infer:

Corollary 6.21 (Computational Adequacy) *If $\emptyset \vdash a : A$ for some type A, then $a{\Downarrow}$ iff $\{\!|a|\!\}_p{\Downarrow_p}$.*

Behavioral equivalences such as barbed congruence or the Morris-style contextual equivalence can also be defined in the Imperative ς-calculus (in fact, since the reduction relation for the Imperative ς-calculus is confluent, the two equivalences coincide).

A context $C[-]$ in the ς-calculus is an (A/B)-context if $-{:}B \vdash C[-] : A$ (that is, we can deduce that $C[-]$ has type A if its hole has type B).

Definition 6.22 (Contextual Equivalence) *For closed ς-calculus terms a and b of type B, we write $a \simeq_B b$ if for all types A and (A/B)-contexts C, it holds that $C[a]{\Downarrow}$ iff $C[b]{\Downarrow}$.*

Just as in Definition 5.8, we follow Gordon and Rees [GR96] by only considering closed terms.

We can show soundness of the translation using compositionality of the encoding and adequacy. This tells us that the equalities that can be proven using the translation are valid equalities.

Theorem 6.23 (Soundness) *Assume $a{:}B$ and $b{:}B$. If $\{\!|a|\!\}_p \approx_b^{p:\{B\}^{\mathsf{ww}}} \{\!|b|\!\}_p$ then $a \simeq_B b$. (Where $\approx_b^{\Lambda}$ is π-calculus weak typed barbed congruence).*

Proof. Consider some (A/B)-context $C[-]$ and assume that $C[a]{\Downarrow}$. We must now show that $C[b]{\Downarrow}$.

By Corollary 6.21 if $C[a]{\Downarrow}$ then $\{\!|C[a]|\!\}_q{\Downarrow}$. Since the translation is compositional, it is easy to see that $\{\!|C[a]|\!\}_q = \mathcal{C}[\{\!|a|\!\}_p]$ for some π-calculus (Γ/Δ)-context $\mathcal{C}[-]$ and name p, where $\Gamma = q : \{\!|A|\!\}^{\mathsf{ww}}$ and $\Delta = p : \{\!|B|\!\}^{\mathsf{ww}}$.

By assumption $\{\!|a|\!\}_p \approx_b^{p:\{B\}^{\mathsf{ww}}} \{\!|b|\!\}_p$, and therefore $\mathcal{C}[\{\!|a|\!\}_p]{\Downarrow}$ implies $\{\!|C[b]|\!\}_p = \mathcal{C}[\{\!|b|\!\}_p]{\Downarrow}$. Finally, we apply Corollary 6.21 to conclude that $C[b]{\Downarrow}$.

If $C[a]{\Uparrow}$ the same line of reasoning is used to conclude that $C[b]{\Uparrow}$. □

As for the encodings of the λ-calculi into the π-calculus, the converse of soundness does not hold for the Imperative ς-calculus.

Example 6.7 Consider the following two objects (from [San98]):

$$\begin{aligned} a &= [\mathrm{l}_1{=}\varsigma(x)b,\ \mathrm{l}_2{=}\varsigma(x)((x.\mathrm{l}_1).\mathrm{h}{\Leftarrow}\varsigma(y)b').\mathrm{h}{\Leftarrow}\varsigma(y)b'] \\ a' &= [\mathrm{l}_1{=}\varsigma(x)b,\ \mathrm{l}_2{=}\varsigma(x)(x.\mathrm{l}_1).\mathrm{h}{\Leftarrow}\varsigma(y)b'] \end{aligned}$$

with b having a method named h. These two objects are contextually equivalent in the ς-calculus, but their translations are not barbed congruent. The reason is that an external observer can update the method l_1 with a new method body, that behaves nondeterministically w.r.t. method updates. For instance, the new method body can decide to diverge on the second request for an update of the h method. By doing so, a will diverge, whereas a' will converge. □

Just as we did for the translation of the Functional ς-calculus in Theorem 5.10, if we restrict the set of π-calculus contexts considered in barbed congruence to contexts that are translations of ς-calculus contexts, we can get a full abstraction result. But again this is not especially useful without an alternative characterization of the resulting equivalence.

6.6 Comparisons with the Functional ς-calculus

In [San98], the Functional ς-calculus is translated into the same typed π-calculus that we have used for the translation of the Imperative ς-calculus in this chapter. Remarkably, despite the strong differences in the operational semantics, our interpretation of the Imperative and Sangiorgi's of the Functional ς-calculus into the π-calculus are structurally very close. Roughly, the only difference between the two translations is in the object manager. The manager for the Functional ς-calculus, reported below, is a functional process (it is replicated, see the discussion on functional processes in Section 6.4). This difference has consequences on the update requests: On such a request, the manager for the Functional ς-calculus always generates a new object manager, whereas the one for the Imperative ς-calculus works by having side effects on itself[3]. Below we present the object manager for the translation of the Functional ς-calculus.

[3]This applies to both the original and the factorized object manager if we consider the cells a part of the functional object manager

$$\mathsf{OB}^{A}_{func}(b_1{:}T^{\mathsf{w}}_{A,1} \ldots b_n{:}T^{\mathsf{w}}_{A,n}, s{:}\{\!|A|\!\}^{\mathsf{b}}) \triangleq$$
$$!s\Big[_{j\in 1..n}\ \mathrm{l}_j\text{-}[\ \mathsf{sel}_{\_}(x) \rhd \overline{b}_j\langle s, x\rangle;$$
$$\mathsf{upd}_{\_}(x,y) \rhd (\nu s'{:}\{\!|A|\!\}^{\mathsf{b}})(\overline{x}s' \mid$$
$$\mathsf{OB}\langle b_1 \ldots b_{j-i}, y, b_{j+1} \ldots b_n, s'\rangle)\ \ ]\Big]$$

The translation of types in the translation of the Functional ς-calculus is also very similar to the one for the Imperative ς-calculus. The only change is that in translation of types in the functional case we do not have the clone tag in the resulting π-calculus type.

The functional and the imperative nature of, respectively, the Functional and Imperative ς-calculus, is reflected in the functional and the imperative nature of the object managers of the π-calculus interpretations. As a consequence, we can use the π-calculus interpretations to compare and contrast the Imperative and Functional ς-calculus — for instance, their discriminating power — and to prove properties about them, within a single framework.

The resemblance between the interpretations also allow us to reuse certain proofs, most notably those on types (see Theorem 6.6 and 6.7), but also certain proofs about behavioral properties of objects (a good example of this is the proof of the law (Eq Sub Obj), in Section 6.7).

On the other hand, as a consequence of the differences in the operational semantics, certain basic laws of for the Functional ς-calculus, such as $a.\mathrm{l}_k = b_k\{a/x_k\}$ $(k \in 1..n)$, do not hold in the Imperative ς-calculus.

In [AC96, Section 10.1] a translation of the Functional ς-calculus to the Imperative ς-calculus is sketched as the identity translation except for method override, which is translated as $\|a.\mathrm{l}\Leftarrow\varsigma(x)b\| \triangleq \mathsf{clone}(\|a\|).\mathrm{l}\Leftarrow\varsigma(x)\|b\|$. It is interesting to note, that this is also the difference between the object managers for the Imperative and Functional ς-calculus — On an override operation, the object manager for the Functional ς-calculus will create a new updated object manager, leaving the original object manager unchanged, whereas the object manager for the Imperative ς-calculus will change its internal state on method override.

6.7 Reasoning about objects

Our main motivation for studying translations of the ς-calculus into the π-calculus was to investigate the possibility of using the π-calculus as a tool for reasoning about objects. We are now in a position to present a number of

examples of how the π-calculus interpretation can be used to validate some basic behavioral properties for the Imperative ς-calculus.

Lemma 6.24 *Let* $o = [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]$, $C = [\mathrm{l}_i{:}B_i\ ^{i\in 1..n}]$, *and* $o{:}C$. *Then*

i. $o.\mathrm{l}_j \simeq_{B_j} \mathsf{let}\ x_j{:}C{=}o\ \mathsf{in}\ b_j$

ii. *If* $x{:}C \vdash b{:}B_j$ *then*

$$o.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}C)b \simeq_C [\mathrm{l}_j{=}\varsigma(x{:}C)b,\ \mathrm{l}_i{=}\varsigma(x_i{:}C)b_i\ ^{i\in 1..n\setminus\{j\}}]$$

iii. $\mathsf{clone}(o) \simeq_C o$

iv. *If* $a{\Downarrow}$, $a{:}A$, $b{:}B$ $x \notin \mathsf{fv}(b)$, *then* $\mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b \simeq_B b$.

v. *If* $x \notin \mathsf{fv}(o)$, $y \notin \mathsf{fv}(a)$, $a{:}A$ *and* $\mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ \mathsf{let}\ y{:}C{=}o\ \mathsf{in}\ b : B$, *then*

$$(\mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ \mathsf{let}\ y{:}C{=}o\ \mathsf{in}\ b) \simeq_B (\mathsf{let}\ y{:}C{=}o\ \mathsf{in}\ \mathsf{let}\ x{:}A{=}a\ \mathsf{in}\ b)$$

vi. *(law (*Eq Sub Object*)):*
If $D = [\mathrm{l}_i{:}B_i\ ^{i\in 1..m}]$, $m \geq n$ *and* $[\mathrm{l}_i{=}\varsigma(x_i{:}D)b_i\ ^{i\in 1..m}]{:}D$, *then*

$$o \simeq_C [\mathrm{l}_i{=}\varsigma(x_i{:}D)b_i\ ^{i\in 1..m}].$$

In the following we shall show the proof of some of the properties stated in Lemma 6.24.

Proof of law v. Laws i-v can all be validated using the theory of the untyped π-calculus. For instance, using laws such as the expansion law, $(\nu p)(p(x).P \mid \overline{p}v.Q) \approx_b (\nu p)(P\{v/x\} \mid Q)$ and $(\nu p)(!p(x).P \mid Q) \approx_b Q$ if p is not free in Q, we can prove law v as follows.

The translation of $\mathsf{let}\ y{=}o\ \mathsf{in}\ \mathsf{let}\ x{=}a\ \mathsf{in}\ b$ is:

$$\begin{aligned} & \{\!|\mathsf{let}\ y{=}o\ \mathsf{in}\ \mathsf{let}\ x{=}a\ \mathsf{in}\ b|\!\}_p \\ = \ & (\nu q)\Big((\nu s, b_1 \ldots b_n)(\overline{q}s \mid \mathsf{OB}\langle b_1 \ldots b_n, s\rangle \mid \\ & \prod_{i\in 1..n} !b_i(x_i, r).\{\!|b_i|\!\}_r) \mid q(y).(\nu q')([\![a]\!]_{q'} \mid q'(x).\{\!|b|\!\}_p)\Big) \end{aligned} \tag{6.5}$$

When we consider (6.5) we see that the only possible transition for the expression is an internal communication on the restricted name q resulting in the exchange of the name y in the subexpression $\{\!|b|\!\}_p$ with the private name

s (remember that y is not free in a). Therefore (6.5) must be barbed congruent to the following expression, where we have replaced the communication with a τ-prefix and changed the name y in $\{\!|b|\!\}_p$ to s.

$$\tau.(\nu q')\Big(\{\!|a|\!\}_{q'} \mid q'(x).(\nu s, b_1 \ldots b_n)(\{\!|b|\!\}_p\{^s/_y\} \mid \mathsf{OB}\langle b_1 \ldots b_n, s\rangle \mid \prod_{i\in 1..n} !b_i(x_i, r).\{\!|b_i|\!\}_r)\Big) \tag{6.6}$$

We can now remove the τ prefix and exchange the substitution of s for y in $\{\!|b|\!\}_p$ with a communication on a private name and thereby obtain the following expression, which is weakly barbed congruent to (6.6).

$$\begin{aligned} & (\nu q')\Big(\{\!|a|\!\}_{q'} \mid q(x).(\nu q)((\nu s)(\overline{q}s \mid \mathsf{OB}\langle b_1 \ldots b_n, s\rangle \mid \\ & \quad \prod_{i\in 1..n} !b_i(x_i, r).\{\!|b_i|\!\}_r) \mid q(y).\{\!|b|\!\}_p)\Big) \\ = \quad & \{\!|\mathsf{let}\ x{=}a\ \mathsf{in}\ \mathsf{let}\ y{=}o\ \mathsf{in}\ b|\!\}_p \end{aligned}$$

□

It is interesting to look at the difference between the Functional and Imperative ς-calculus for objects of the form $o.\mathrm{l}_j$, for $o = [\mathrm{l}_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}]$ $(j \in 1..n)$, using the π-calculus interpretations. In the case of the Functional ς-calculus, o is interpreted as a functional process (see Section 6.4) and we can therefore apply copy laws to derive $\{\!|o.\mathrm{l}_j|\!\}_p \approx_b \{\!|b_j\{^o/_{x_j}\}|\!\}_p$. By contrast, in the case of the Imperative ς-calculus, object o is interpreted as a process with a state, and we can only infer $\{\!|o.\mathrm{l}_j|\!\}_p \approx_b \{\!|\mathsf{let}\ x_j{:}A{=}o\ \mathsf{in}\ b_j|\!\}_p$, as by Lemma 6.24(i).

Law vi (Eq Sub Object), which is an adaptation of the corresponding law of the same name of the Functional ς-calculus [AC96, Chapter 8], allows one to prove equalities between objects with different collections of methods, and relies on the type information; so of course here, we need to consider the types in the proof of that law.

Proof of law (Eq Sub Object). Law (Eq Sub Object) of Lemma 6.24 can be validated using the typed labelled bisimulation technique. We describe a typed bisimulation for proving (Eq Sub Object).

Let $a = [\mathrm{l}_i{=}\varsigma(x_i{:}C)b_i\ ^{i\in 1..n}]$, $b = [\mathrm{l}_i{=}\varsigma(x_i{:}D)b_i\ ^{i\in 1..m}]$, $C = [\mathrm{l}_i{:}B_i\ ^{i\in 1..n}]$, and $D = [\mathrm{l}_i{:}B_i\ ^{i\in 1..m}]$ with $n \le m$. Furthermore assume that $a : C$ and $b : D$.

We want to show $a \simeq_C b$. By Theorem 6.23, it suffices to prove $\{\!|a|\!\}_p \simeq_\Gamma \{\!|b|\!\}_p$, where $\Gamma = p{:}\{\!|C|\!\}^{\mathsf{w}^{\mathsf{w}}}$, and by Theorem A.11 we can use typed labelled bisimulation establish the result, by showing that $\{\!|a|\!\}_p \simeq_{p:\{C\}^{\mathsf{w}^{\mathsf{r}}}} \{\!|b|\!\}_p$.

Having fixed a and b, their translations are:

$$\begin{aligned}
\{\!|a|\!\}_p &= (\nu s{:}\{\!|C|\!\}^{\mathsf{b}})\Big(\overline{p}s \mid (\nu b_i{:}T^{\mathsf{b}}_{C,i}{}^{\,i\in 1..n})(\mathsf{OB}^C\langle s, b_1 \ldots b_n\rangle \mid \textstyle\prod_{i\in 1..n} !b_i(x_i, r).\{\!|b_i|\!\}_r)\Big) \\
\{\!|b|\!\}_p &= (\nu s{:}\{\!|D|\!\}^{\mathsf{b}})\Big(\overline{p}s \mid (\nu b_i{:}T^{\mathsf{b}}_{D,i}{}^{\,i\in 1..m})(\mathsf{OB}^D\langle s, b_1 \ldots b_m\rangle \mid \textstyle\prod_{i\in 1..m} !b_i(x_i, r).\{\!|b_i|\!\}_r)\Big)
\end{aligned}$$

Now, let

$$\Gamma' = s{:}\{\!|C|\!\}^{\mathsf{w}}, b_1{:}T^{\mathsf{r}}_{C,1}, \ldots, b_n{:}T^{\mathsf{r}}_{C,n}$$

and

$$\Gamma'' = p{:}\{\!|C|\!\}^{\mathsf{w}^{\mathsf{r}}}, b_1{:}T^{\mathsf{r}}_{C,1}, \ldots, b_n{:}T^{\mathsf{r}}_{C,n},$$

where $T_{C,j} \triangleq \langle \{\!|C|\!\}^{\mathsf{w}}, \{\!|B_j|\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle$. Consider the relation consisting of these two triples:

1) $$\Big(\Gamma', \mathsf{OB}^C\langle b_1 \ldots b_n, s\rangle, (\nu b_i{:}T^{\mathsf{b}}_{C,i}{}^{\,i\in n+1..m})(\mathsf{OB}^D\langle b_1 \ldots b_m, s\rangle \mid \prod_{i\in n+1..m} !b_i(x_i, r).\{\!|b_i|\!\}_r)\Big)$$

2) $$\Big(\Gamma'', (\nu s{:}\{\!|C|\!\}^{\mathsf{b}})(\overline{p}s \mid \mathsf{OB}^C\langle b_1 \ldots b_n, s\rangle, (\nu s{:}\{\!|D|\!\}^{\mathsf{b}})(\nu b_i{:}T^{\mathsf{b}}_{D,i}{}^{\,i\in n+1..m})(\overline{p}s \mid \mathsf{OB}^D\langle b_1 \ldots b_m, s\rangle \mid \prod_{i\in n+1..m} !b_i(x_i, r).\{\!|b_i|\!\}_r)\Big)$$

This relation is a typed bisimulation up to parallel composition and up to injective substitutions. Using the congruence property of typed bisimulation (precisely Corollary A.10 and Lemma A.14), we get the desired result. □

This is the first proof of (Eq Sub Object) for the Imperative ς-calculus we are aware of. The proof can be easily adapted to the analogous law of the Functional ς-calculus.

Proofs of laws i to v have also been given by Gordon, Hankin and Lassen [GHL97], using a variant of Mason and Talcott's CIU equivalence.

We are not aware of other coinductive proofs of laws or behavioral equalities of the Imperative ς-calculus. By contrast, coinductive techniques for

the Functional ς-calculus have been carried out by Gordon and Rees [GR96] (see Chapter 8). Gordon and Rees propose a labelled bisimulation for the Functional ς-calculus (following the idea of Abramsky's applicative bisimulation for λ-calculi) and show that it coincides with contextual equivalence, which they show validates the equational theory for the Functional ς-calculus (The equational theory for the Functional ς-calculus has also been validated using denotational methods [AC96, Chapter 14]). It is worth mentioning that Gordon and Rees's applicative bisimulation, however, contains a form of universal quantification on terms in its definition. For this reason

- applicative bisimulation relations always consist of an infinite number of pairs (unless all objects are divergent);
- applicative bisimulation does not work well for proving properties like (Eq Sub Object); indeed, Gordon and Rees's proof of (Eq Sub Object) for the Functional ς-calculus does not use applicative bisimulation, but it is done directly in terms of the contextual equivalence.

Advantages of using π-calculus for the proofs are that the bisimulation relations needed may be finite (for instance, the bisimulation we use for the proof of (Eq Sub Object) has just two pairs); one can take advantage of the already available theory for π-calculus, including its algebraic laws (as we did for the proof of law 5 above).

6.8 Final remarks and comments

In this chapter, we have shown a typed translation of the Imperative ς-calculus into the π-calculus for which we have shown soundness of the type translations and operational adequate. Furthermore, we have compared our translation with a similar translation of the Functional ς-calculus. Finally, we gave some examples of how the translations could be used to reason about properties for the Imperative ς-calculus.

This chapter is an extended version of a paper [KS98], which to our knowledge is one of the first to look at the Imperative ς-calculus of Abadi and Cardelli.

6.8.1 Extensions

The interpretation of the Imperative ς-calculus can be extended to accommodate other type features discussed in [AC96], such as variant tags, recursive types, and polymorphic types.

Variant tags are tags on method names which allow only selection or update operations on a method, so as to have a richer subtyping relation. These tags yield the same form of subtyping on ς-calculus types as that induced by the tags $\{\mathsf{r}, \mathsf{w}, \mathsf{b}\}$ on the π-calculus types. We can capture them with a simple refinement of the encoding of types. Variant tags $(+, -, \mathsf{b})$ allow a more refined subtyping relation. A positive tag $(+)$ forbids method update, but allows covariant subtyping in the components of an object type. A negative tag $(-)$ prevents method activation, but allows contravariant subtyping. Finally, the tag b allows both update and activation, but requires invariance of the method types.

To accommodate variant tags, we change the encoding of types to:

$$\begin{aligned}
\{\!\{\{l_i^I : B_i\}\}\!\} &= \mu X.[\mathrm{l}_{i_}\{\!\{I{:}B_i\}\!\}, \mathsf{clone}_X^{\mathsf{w}^{\mathsf{w}}}] \\
\{\!\{\mathsf{b} : B_i\}\!\} &= [\mathsf{sel}__[\![B_i]\!]^{\mathsf{w}^{\mathsf{w}}}; \mathsf{upd}__\langle X^{\mathsf{w}^{\mathsf{w}}}, \{\!\{B_i\}\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle^{\mathsf{w}}] \\
\{\!\{+ : B_i\}\!\} &= [\mathsf{sel}__[\![B_i]\!]^{\mathsf{w}^{\mathsf{w}}}] \\
\{\!\{- : B_i\}\!\} &= [\mathsf{upd}__\langle X^{\mathsf{w}^{\mathsf{w}}}, \{\!\{B_i\}\!\}^{\mathsf{w}^{\mathsf{w}}}\rangle^{\mathsf{w}}]
\end{aligned}$$

Also, the addition of a recursive type system like the one in [AC96, Chapter 9] is easily handled. We just map type variables of an Imperative ς-calculus type to type variables of the π-calculus types. Similarly, it is possible to handle *polymorphic types*, using polymorphic extensions of the π-calculus [Tur96].

Chapter 7

Relation between Operational and the Denotational Semantics

In the previous chapters we have considered the translations of different versions of ς-calculi to the π-calculus. Our main motivation for these translations was to use them to support reasoning about objects.

In Section 3.4 we saw another way of supporting reasoning about objects; namely via equational theories. For an equational theory to make any sense, it must of course be sound w.r.t. the semantic model(s) of the language. The strategy employed by Abadi and Cardelli was to show that the equational theories are sound with respect to a denotational semantics based on partial equivalence relations [AC96, Chapter"nobreakspace –"14].

Notions of program equivalence are central to the theory and practice of programming languages, and we have already briefly discussed equivalences for the ς-calculus in the previous chapters. They form the basis for program optimization, and can be used to justify correctness preserving transformations performed by program manipulation systems. Program equivalences are typically defined according to the following paradigm:

i. A collection of terms that are considered to be directly executable and observable are designated as programs, and their behavior is defined;

ii. Two arbitrary terms are defined to be equivalent iff they have the same behavior in every program context.

The resulting notion of program equivalence is usually referred to as *observational congruence* [Mor68, Mey88]. Observational congruence for the first order ς-calculus with subtyping $\mathbf{Ob}_{1<:\mu}$ (see also Section 3.3) has been defined in [GR96] thus: Two programs are observationally congruent iff they

have the same termination behavior in all contexts of boolean type. Following earlier work on functional languages, Gordon and Rees equip the calculus $\mathbf{Ob}_{1<:\mu}$ with a labelled transition system semantics, and its associated notion of bisimulation equivalence is proven to coincide with observational congruence. Like the denotational model presented in [AC96, Chapter 14], observational congruence soundly models Abadi and Cardelli's equational theory for objects (cf. [GR96, Theorem"nobreakspace −"3]).

The work of Gordon and Rees shows that process calculus techniques can be applied directly to the ς-calculus. In their case, firstly for showing the equational theories sound, and secondly for showing equalities between objects, that cannot be derived using the equational theories, using bisimulation directly.

The results discussed so far provide two different semantic models for the calculus $\mathbf{Ob}_{1<:\mu}$ that soundly model the equational theory underlying that version of the ς-calculus. Just as for translations, the acid test for the goodness of any denotational model for programming languages is the nature of the connection between the mathematical meaning it assigns to programs and their computational behavior. In particular, a denotational model should be *correct* [Sto88] in the sense that it identifies only terms that are related by observational congruence. Models with the ideal property of identifying exactly those terms that are observationally congruent are called *fully abstract* with respect to the former. Perhaps surprisingly, the literature on the object calculi lacks a study of the relationship between Abadi and Cardelli's denotational semantics and observational congruence, as studied by Gordon and Rees. The aim of this chapter is to provide such an analysis.

In this chapter we study Abadi and Cardelli's denotational semantics vis-à-vis observational congruence over the ς-calculus with type system $\mathbf{Ob}_{1<:\mu}$. In particular, we prove that the denotational semantics based on partial equivalence relations of [AC96, Chapter 14] is correct with respect to observational congruence of objects (Theorem 7.11). As an important stepping stone towards this correctness result, we show that the denotational semantics is computationally adequate with respect to the reduction semantics (Theorem 7.8), and that a program of boolean type evaluates to a boolean value v iff its denotation equals that of v (Corollary 7.10). By means of a counter-example, we argue that the denotational model is *not* fully abstract with respect to observational congruence. In fact, the model is able to distinguish objects that have the same behavior in every $\mathbf{Ob}_{1<:\mu}$-context. As a byproduct of our results we obtain an alternative proof of the soundness of the equational theory with respect to bisimulation (Proposition 7.12).

We end this introduction with a brief road-map to the contents of this chapter. The labelled transition semantics of the ς-calculus and the notion

of bisimulation equivalence are introduced in Section 7.1. Section 7.2 gives a brief overview of the denotational model of $\mathbf{Ob}_{1<:\mu}$ and its types. Finally, Section 7.3 presents our main result, viz. that the denotational model is correct, but not fully abstract.

7.1 A labelled transition semantics

In this section we shall give a short review of the labelled transition semantics proposed by Gordon and Rees in [GR96]. Only terms of matching types are considered to be related semantically. This is formalized by introducing the notion of *proved programs*, i.e. elements of the form a_A where a is a program of type A. We let $\mathbf{Prog} \triangleq \{a_A \mid a{:}A\}$ denote the set of well-typed closed objects annotated with types (called *proved programs* by Gordon and Rees) and let *Rel* be the universal relation on proved programs of the same type, i.e.

$$\mathit{Rel} \triangleq \{(a_A, b_A) \mid a{:}A \text{ and } b{:}A\}.$$

The types of $\mathbf{Ob}_{1<:\mu}$ are divided into two classes, *active* and *passive*. Active types are the types of values. Only Bool is active, so in our presentation all values are booleans. Recursive types, object types and Top are passive types. At active types a program must converge to a value before it can be observed; at passive types a program performs observable actions unconditionally, whether or not it converges.

The observable actions, $\alpha \in \mathit{Act}$, take the following forms:

$$\alpha ::= \mathsf{true} \mid \mathsf{false} \mid \mathrm{l} \mid \mathrm{l}{\Leftarrow}\varsigma(x)b \mid \mathsf{unfold}.$$

These observations should be interpreted as follows: An object term allows the observation true (resp. false) if the term is of type Bool and has the value true (resp. false). An object term allows the observation l if it has a method labelled l. An object term allows the observation $\mathrm{l}{\Leftarrow}\varsigma(x)b$ if the object can have its method labelled l redefined as $\varsigma(x)b$. And finally, an object term allows the observation unfold if it is the fold of an object.

The family $\{\xrightarrow{\alpha} \mid \alpha \in \mathit{Act}\}$ of transition relations over proved programs is defined as the set of the least relations satisfying the rules in Table 7.1.

The definition of bisimulation equivalence over proved programs is then basically as for the π-calculus (Definition 4.1).

Definition 7.1 (Bisimulation) *Bisimilarity $\sim$ is the greatest subset of Rel that satisfies the following: $a_A \sim b_A$ if and only if*

(Trans Bool)
$$\frac{a \Downarrow v \in \{\mathsf{true}, \mathsf{false}\}}{a_{\mathsf{Bool}} \xrightarrow{v} a_{\mathsf{Top}}}$$

(Trans Select)
$$\frac{j \in I \quad A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]}{a_A \xrightarrow{\mathrm{l}_j} a.\mathrm{l}_j\ _{B_j}}$$

(Trans Update)
$$\frac{x{:}A \vdash b{:}B_j \quad j \in I \quad A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]}{a_A \xrightarrow{\mathrm{l}_j \Leftarrow \varsigma(x)b} a.\mathrm{l}_j \Leftarrow \varsigma(x{:}A)b_A}$$

(Trans Unfold)
$$\frac{A = \mu(X)B \quad C = B\{{}^{A}/_{X}\}}{a_A \xrightarrow{\mathsf{unfold}} \mathsf{unfold}(a)_C}$$

Table 7.1: The rules of the labelled transition semantics

i. $a_A \xrightarrow{\alpha} a'_{A'} \Rightarrow \exists\ b'_{A'}\ .\ (b_A \xrightarrow{\alpha} b'_{A'} \wedge a'_{A'} \sim b'_{A'})$ *and*

ii. $b_A \xrightarrow{\alpha} b'_{A'} \Rightarrow \exists\ a'_{A'}\ .\ (a_A \xrightarrow{\alpha} a'_{A'} \wedge a'_{A'} \sim b'_{A'})$.

If $a_A \sim b_A$ *we say that* a_A *and* b_A *are bisimilar.*

It should be noted that Definition 7.1 not only relates terms a_A and b_A on their type A, but also on all of their supertypes. That is, if the subtype relation

$$A <: B_1 <: B_2 <: \cdots <: B_n$$

holds, then Definition 7.1 states that the following must hold:

$$a_A \sim b_A \text{ implies } a_{B_1} \sim b_{B_1}, \ldots, a_{B_n} \sim b_{B_n}.$$

Intuitively, this is true because if $A <: B$ then a_A has more transitions than a_B, so if $a_A \sim b_A$ then they must also be equivalent when condering the fewer transitions that type B allows (see [GR96] for details).

The notion of equivalence chosen by Gordon and Rees for the object calculus is an observational congruence where two terms are considered equivalent if they have the same termination behavior in all contexts of type Bool. As usual, we shall only consider well typed contexts and we write $-{:}B \vdash C[-]{:}A$ if the context C has type A under the assumption that the hole has type B.

Definition 7.2 (Observational congruence) *We write* $a_B \overset{A}{\simeq} b_B$ *iff for all contexts satisfying* $-{:}B \vdash C[-]{:}A$ *we have* $C[a]\Downarrow$ *iff* $C[b]\Downarrow$.

Intuitively, contexts should be considered as the possible tests that an object can be subjected to. One should note that the naturalness of the notion

of observational congruence crucially depends upon the choice of observable types. For instance, it is easy to see that $\mathsf{true} \overset{\mathsf{Top}}{\not\simeq} \Omega_{\mathsf{Top}}$, which violates the rule (Eq Top) from Table 3.11 expressing that all objects are to be considered equal at type Top. Amongst the relations $\overset{A}{\simeq}$, congruence at type Top, viz. $\overset{\mathsf{Top}}{\simeq}$, is the most discriminating and $\overset{\mathsf{Bool}}{\simeq}$ the least. Rule (Eq Top) holds for $\overset{\mathsf{Bool}}{\simeq}$ and, for that reason and by analogy with [Plo77], Gordon and Rees choose $\overset{\mathsf{Bool}}{\simeq}$ as the appropriate notion of observational congruence for $\mathbf{Ob}_{1<:\mu}$.

In [GR96] Gordon and Rees show that bisimulation coincides with observational congruence and that these relations validate the equational theory of Tables 3.9–3.11.

The following lemma collects some basic properties of the type assignment, and of its interaction with the reduction relation which we shall need later.

Lemma 7.3 ([GR96])

i. If $\Gamma \vdash a{:}A$, *then* $\mathsf{fv}(a) \subseteq \mathsf{dom}(\Gamma)$.

ii. If $a{:}A$ *and* $a \rightsquigarrow b$, *then* $b{:}A$.

7.2 The denotational semantics

In this section we shall give a short description of the denotational semantics given in [AC96, Chapter 14], where the interested reader will find more details.

The denotational semantics is based on a two-level approach. The first level consists of a standard cpo model for interpreting untyped objects. Types are then modelled by certain kinds of partial equivalence relations (pers) over the object domain. In this two-level semantics the objects a and b are considered equal in the type A if $(\llbracket a \rrbracket, \llbracket b \rrbracket) \in \llbracket A \rrbracket$, where $\llbracket a \rrbracket, \llbracket b \rrbracket$ and $\llbracket A \rrbracket$ are the corresponding interpretations.

7.2.1 The untyped model

The untyped model is a cpo obtained as a solution to the domain equation

$$D = \{*\}_\perp + \{tt, ff\}_\perp + (D \to D) + (\mathbf{MNames} \to D)_\perp$$

where $\mathbf{MNames} = \{\mathrm{l}_1, \mathrm{l}_2 \ldots\}$ is a countable set of method names, $D \to D$ and $(\mathbf{MNames} \to D)_\perp$ have the usual meaning and $+$ is coalesced sum

[Plo90]. The solution is obtained as the limit of the following sequence of iterates:

$$\begin{array}{lcl} D_0 & = & \{\bot\} \\ D_{n+1} & = & \{*\}_\bot + \{\mathit{tt}, \mathit{ff}\}_\bot + (D_n \to D_n) + (\mathbf{MNames}_n \to D_n)_\bot \\ & & \text{where } \mathbf{MNames}_n = \{\mathrm{l}_1, \ldots, \mathrm{l}_n\} \end{array}$$

We consider D_i as being a subset of D.

There is an increasing sequence, $p_n : D \to D_n$, of projections related to the model with the identity map as its least upper bound. If $x \in D$ and $p_n(x) = x$ for some n, then x is said to be *finite*. The *rank* of a finite element x is the least n such that $p_n(x) = x$.

We use $\langle\!\langle \mathrm{l}_1{=}x_1, \ldots, \mathrm{l}_n{=}x_n \rangle\!\rangle$ to denote the function in $\mathbf{MNames} \to D$ that maps l_i to x_i for $i \leq n$ and all other labels to $*$. The notation $\langle\!\langle \mathrm{l}_1{=}x_1, \ldots, \mathrm{l}_n{=}x_n \rangle\!\rangle \langle \mathrm{l} \mapsto x \rangle$ will stand for the function in $\mathbf{MNames} \to D$ that maps l to x, and agrees with $\langle\!\langle \mathrm{l}_1{=}x_1, \ldots, \mathrm{l}_n{=}x_n \rangle\!\rangle$ on all the other inputs.

The semantic function for terms $[\![\cdot]\!] : (\mathbf{SVar} \to D) \to (\mathbf{Obj} \to D)$ is defined in Table 7.2. In the definition, the symbol $\Subset$ is a strict membership

---

$$[\![x]\!]_\rho = \rho(x)$$

$$[\![[\mathrm{l}_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in I}]]\!]_\rho = \langle\!\langle \mathrm{l}_i = \lambda v.[\![b_i]\!]_{\rho\langle x_i \mapsto v\rangle}^{i\in I} \rangle\!\rangle$$

$$[\![a.\mathrm{l}]\!]_\rho = \begin{cases} [\![a]\!]_\rho(\mathrm{l})([\![a]\!]_\rho) & \text{if } [\![a]\!]_\rho \Subset \mathbf{MNames} \to D \text{ and } [\![a]\!]_\rho(\mathrm{l}) \Subset D \to D \\ * & \text{otherwise} \end{cases}$$

$$[\![a.\mathrm{l}{\Leftarrow}\varsigma(x{:}A)b]\!]_\rho = \begin{cases} [\![a]\!]_\rho\langle \mathrm{l} \mapsto \lambda v.[\![b]\!]_{\rho\langle x \mapsto v\rangle} \rangle & \text{if } [\![a]\!]_\rho \Subset \mathbf{MNames} \to D \\ * & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fold}(A, a)]\!]_\rho = \lambda v.[\![A]\!]_\rho$$

$$[\![\mathsf{unfold}(a)]\!]_\rho = [\![a]\!]_\rho(\bot)$$

$$[\![\mathsf{true}]\!]_\rho = \mathit{tt},\ [\![\mathsf{false}]\!]_\rho = \mathit{ff}$$

$$[\![\mathsf{if}(a, b_1, b_2)]\!]_\rho = \begin{cases} [\![b_1]\!]_\rho & \text{if } [\![a]\!]_\rho = \mathit{tt}, \\ [\![b_2]\!]_\rho & \text{if } [\![a]\!]_\rho = \mathit{ff} \\ \bot & \text{if } [\![a]\!]_\rho = \bot \\ * & \text{otherwise} \end{cases}$$

Table 7.2: The semantic function for terms

---

test (if $[\![a]\!]_\rho = \bot$, then $([\![a]\!]_\rho \Subset \mathbf{S}) = \bot$), and we use ρ to stand for an environment, i.e., a mapping from **SVar** to D. Moreover, conditionals and conjunctions are strict and evaluated left to right. If a is closed we write $[\![a]\!]$ instead of $[\![a]\!]_\rho$.

The following standard result will be useful in the remainder of the chapter. (Cf. Lemma C.4–6 on page 356 of [AC96].)

Lemma 7.4 (Substitution Lemma) *For object terms a, b and variable x,*

$$[\![a]\!]_{\rho\langle x \mapsto [\![b]\!]_\rho\rangle} = [\![a\{{}^{b}\!/_{x}\}]\!]_\rho \ .$$

7.2.2 Introducing types into the model

Types are modelled as certain binary relations over D. A *per* is a symmetric, transitive, binary relation on D that (by convention) does not have $*$ in its domain. A binary relation P is *uniform* if xPy implies $p_i(x)Pp_i(y)$ for all i. It is *complete* if $\bot P \bot$ and if whenever $\langle x_i \rangle$ and $\langle y_i \rangle$ are chains where $x_i P y_i$ for all i then $\sqcup x_i P \sqcup y_i$. A *cuper* is a complete uniform per. The set of all cupers is **Cuper** ranged over by R, S, T.

We use $p_r(R)$ to denote the restriction of the cuper R to those pairs whose rank is no greater than r. **Cuper** can be given the structure of a complete metric space with the metric $d : \mathbf{Cuper} \times \mathbf{Cuper} \to \mathbf{R}_+$ defined as

$$d(R,T) = \max(\{0\} \cup \{2^{-r} \mid p_r(R) \neq p_r(T)\}).$$

A function $F : \mathbf{Cuper} \to \mathbf{Cuper}$ is *contractive* if whenever $R, S \in \mathbf{Cuper}$, $d(F(R), F(S)) \leq 2^{-1} d(R,S)$. Banach's fixed point theorem guarantees that all contractive endofunctions in **Cuper** have a unique fixed point μF [Apo82].

The following operators over **Cuper** are used to define the semantics of types:

- $\mathbf{Univ} = (D \setminus \{*\}) \times (D \setminus \{*\})$
- $\mathbf{Bool} = \{(\bot,\bot), (tt, tt), (ff, ff)\}$
- $P \to Q = \{(f,g) \in (D \to D) \times (D \to D) \mid \forall x, y \ . \ xPy \Rightarrow f(x)Qg(y)\}$
- $\sqcup_{i \in I} P_i = \mathcal{C}(\cup_{i \in I} P_i)$, where $\mathcal{C}(P)$ is the least cuper that contains P
- $\langle\!\langle \mathrm{l}_i{:}B_i \ {}^{i \in I} \rangle\!\rangle = \{(\bot,\bot)\} \cup \{(o,o') \in (\mathbf{MNames} \to D) \times (\mathbf{MNames} \to D) \mid \forall i \in I \ . \ (o(\mathrm{l}_i), o'(\mathrm{l}_i)) \in B_i\}$

The function $\lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in I} \rangle\!\rangle$ is contractive and therefore has a unique fixed point. We say that $\lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in J} \rangle\!\rangle$ extends $\lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in I} \rangle\!\rangle$, written $\lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in J} \rangle\!\rangle \preceq \lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in I} \rangle\!\rangle$ if $I \subseteq J$. The set of all functions of the form $\lambda S. \langle\!\langle \mathrm{l}_i{:}S \to T_i \ {}^{i \in I} \rangle\!\rangle$ is called **Gen**. We have the following operator in **Cuper**:

$$\|\mathrm{l}_i{:}B_i \ {}^{i \in I}\| = \sqcup\{\mu F | F \in \mathbf{Gen}, F \preceq \lambda S. \langle\!\langle \mathrm{l}_i{:}S \to B_i \ {}^{i \in I} \rangle\!\rangle\}.$$

The semantic function for types

$$[\![\cdot]\!] : (\mathbf{TypeVar} \to \mathbf{Cuper}) \to (\mathbf{Type} \to \mathbf{Cuper})$$

is defined as follows:

$$\begin{array}{lcl} [\![X]\!]_\eta & = & \eta(X) \\ [\![\mathsf{Top}]\!]_\eta & = & \mathbf{Univ} \\ [\![[\mathrm{l}_i{:}B_i\ ^{i\in I}]]\!]_\eta & = & \|\mathrm{l}_i{:}B_i\ ^{i\in I}\| \\ [\![\mu(X)A]\!]_\eta & = & \mu T.(\mathbf{Univ} \to [\![A]\!]_{\eta\langle X \mapsto T\rangle}) \\ [\![\mathsf{Bool}]\!]_\eta & = & \mathbf{Bool} \end{array}$$

where η denotes a type environment, i.e., a mapping from **TypeVar** to **Cuper**. Again we write $[\![A]\!]$ instead of $[\![A]\!]_\eta$ for closed type expressions.

In later developments, we shall need the following result.

Lemma 7.5 *If* $\{(x,y),(x',y')\} \subseteq \|\mathrm{l}_i{:}T_i\ ^{i\in I}\|$ *then* $(x(\mathrm{l}_i)x', y(\mathrm{l}_i)y') \in T_i$ *for all* $i \in I$.

Proof. Similar to the proof of Proposition C.4-4 in [AC96]. □

7.2.3 Soundness of the type and equational theory

We can now define the semantic counterparts of type and subtyping judgments. In order to do this, we shall need a notion of consistency. We say that Γ, η and (ρ,ρ') are *consistent* if

- whenever $X <: A$ is in Γ then $\eta(X) \subseteq [\![A]\!]_\eta$ and
- whenever $x{:}A$ is in Γ then $(\rho(x),\rho'(x)) \in [\![A]\!]_\eta$.

Now for any consistent Γ and η, (ρ,ρ') and any A, B, a, a' we define

$$\begin{array}{lll} \Gamma \models_{\eta,(\rho,\rho')} A & \text{iff} & [\![A]\!]_\eta \in \mathbf{Cuper} \\ \Gamma \models_{\eta,(\rho,\rho')} A <: B & \text{iff} & [\![A]\!]_\eta \subseteq [\![B]\!]_\eta \\ \Gamma \models_{\eta,(\rho,\rho')} a{:}A & \text{iff} & ([\![a]\!]_\rho, [\![a]\!]_{\rho'}) \in [\![A]\!]_\eta \\ \Gamma \models_{\eta,(\rho,\rho')} a \leftrightarrow a' : A & \text{iff} & ([\![a]\!]_\rho, [\![a']\!]_{\rho'}) \in [\![A]\!]_\eta \end{array}$$

Let $\mathbf{cons}(\Gamma) = \{(\eta,(\rho,\rho')) \mid \Gamma \text{ and } (\eta,(\rho,\rho')) \text{ are consistent}\}$. For $\gamma \in \{A, A <: B, a{:}A, a \leftrightarrow b : A\}$ we say that

$$\Gamma \models \gamma \text{ iff } \forall(\eta,(\rho,\rho')) \in \mathbf{cons}(\Gamma)\ .\ \Gamma \models_{\eta,(\rho,\rho')} \gamma.$$

The soundness of the type and equational theory can now be stated as follows.

Theorem 7.6 ([AC96]) *The relation* $\models$ *is preserved by the rules in Tables 3.7, 3.8 and 3.9-3.11. Therefore, for all* Γ *and* $\gamma \in \{A, A <: B, a{:}A, a \leftrightarrow b : A\}$, $\Gamma \vdash \gamma$ *implies* $\Gamma \models \gamma$.

7.3 Correctness of the denotational model

We shall now investigate the relationship between the equivalence on programs induced by Abadi and Cardelli's denotational semantics and observational congruence. More precisely, we prove that the denotational semantics presented in Section 7.2 is correct with respect to observational congruence, i.e., that it identifies only terms that are related by observational congruence. By means of an example, we shall also argue that the denotational semantics is *not* fully abstract.

The proof of correctness of the denotational semantics will be delivered in three steps. We begin by showing a soundness result for the reduction relation with respect to the denotational semantics.

Proposition 7.7 *For every program a and value v, if $a \Downarrow v$ then $[\![a]\!] = [\![v]\!]$.*

Proof. We begin by proving that if $a \rightsquigarrow b$, then $[\![a]\!] = [\![b]\!]$. In light of Lemma 7.4, it is sufficient to establish the above claim for the basic reduction rules. We confine ourselves to examining two of these rules.

- Assume that $a = a'.\mathrm{l}_k \rightsquigarrow b_k\{{}^{a'}/_{x_k}\} = b$, where $a' = [\mathrm{l}_i{=}\varsigma(x_i{:}A_i)b_i \ {}^{i\in I}]$ and $k \in I$. First of all, note that

  $$[\![a']\!] = \langle\!\langle \mathrm{l}_i = \lambda u.[\![b_i]\!]_{\rho\langle x \mapsto u\rangle} \ {}^{i\in I} \rangle\!\rangle \ .$$

  Using the definition of the semantic function given in Table 7.2 and Lemma 7.4, we can now prove that $[\![a]\!] = [\![b]\!]$ thus:

  $$\begin{array}{rcl} [\![a]\!] & = & [\![a']\!](\mathrm{l}_k)([\![a']\!]) \\ & = & (\lambda u.[\![b_k]\!]_{\rho\langle x_k \mapsto u\rangle})([\![a']\!]) \\ & = & [\![b_k]\!]_{\rho\langle x_k \mapsto [\![a']\!]\rangle} \\ & = & [\![b_k\{{}^{a'}/_{x_k}\}]\!] \\ & = & [\![b]\!] \ . \end{array}$$

- Assume that $a = \mathsf{unfold}(\mathsf{fold}(A, v)) \rightsquigarrow v = b$. Using the definition of the semantic function given in Table 7.2, we can now prove that $[\![a]\!] = [\![b]\!]$ thus: $[\![a]\!] = (\lambda u.[\![v]\!])(\bot) = [\![v]\!] = [\![b]\!]$.

The statement now follows easily by induction on the length of the (unique) sequence of reductions leading from a to v. □

Of course, one cannot expect the converse of this soundness property to hold because objects are values whether or not the bodies of their methods are fully evaluated. For example, the objects $[\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}\mathsf{Bool}])\mathsf{true}]$ and

$[\mathrm{l}{=}\varsigma(x{:}[\mathrm{l}{:}\mathsf{Bool}])\mathsf{if}(\mathsf{true},\mathsf{true},\mathsf{true})]$ have the same denotation, but are different values. However, if a program has a denotation different from $\bot$, then it reduces to some value. In particular, at the observed type Bool a program evaluates to a value v if and only if its denotation is $[\![v]\!]$. This property is usually referred to as *computational adequacy* [Mey88], and is the essential connection between a denotational and an operationally based semantics.

Theorem 7.8 (Computational Adequacy) *Let $a{:}A$ be such that $[\![a]\!] \neq \bot$. Then $a{\Downarrow}v$ for some value v.*

The proof of the above result is based on an adaptation of a strategy due to Plotkin [Plo77]. We begin by defining a formal approximation relation $\lhd$ between elements of the domain D and programs with the following properties:

> For any $d \in D$ and program a, $d \lhd a$ (d approximates the value of a) iff
>
> i. $d = \bot$, or
>
> ii. $a{\Downarrow}v$ for some value v such that $d \lhd v$, where
>
> (a) $\mathit{tt} \lhd \mathsf{true}$ and $\mathit{ff} \lhd \mathsf{false}$,
>
> (b) $\langle\!\langle \mathrm{l}_i = d_i \ ^{i\in I} \rangle\!\rangle \lhd [\mathrm{l}_i{=}\varsigma(x_i{:}[\mathrm{l}_i{:}A_i \ ^{i\in I}])e_i \ ^{i\in I}]$ iff for every d' such that $(d', d') \in [\![[\mathrm{l}_i{:}A_i \ ^{i\in I}]]\!]$ and $a'{:}[\mathrm{l}_i{:}A_i \ ^{i\in I}]$, $d' \lhd a'$ implies $d_i(d') \lhd e_i\{{}^{a'}\!/_{x_i}\}$ for every $i \in I$,
>
> (c) $\lambda u.d \lhd \mathsf{fold}(\mu(X)A, v)$ iff $d \lhd v$.

The existence of a relation with these properties may be shown following the developments in [Pit94].

The key to the proof of Theorem 7.8 is the following technical result.

Lemma 7.9 *Assume that $x_1{:}A_1, \ldots, x_n{:}A_n \vdash e{:}A$. Let $d_1, \ldots, d_n$ and $a_1, \ldots, a_n$ be such that $(d_i, d_i) \in [\![A_i]\!]$, $a_i{:}A_i$ and $d_i \lhd a_i$, for every $i \in 1..n$. Then*

$$[\![e]\!]_{\langle x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\rangle} \lhd e\{{}^{a_i}\!/_{x_i}\}_{i=1}^{n} \ .$$

Proof. We prove the claim by induction on the depth of the proof of the type assignment $x_1{:}A_1, \ldots, x_n{:}A_n \vdash e{:}A$. For the sake of conciseness, throughout the proof we shall use ρ to stand for the environment $\langle x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\rangle$, and σ to denote the substitution $\{{}^{a_i}\!/_{x_i}\}_{i=1}^{n}$. The list of type assumptions $x_1{:}A_1, \ldots, x_n{:}A_n$ will be referred to as Γ. We proceed by a case analysis on the last rule used in the proof of the type assignment $\Gamma \vdash e{:}A$, and only detail the proof for the nontrivial cases. (The reader is referred to Table 3.7 for the list of the typing rules.)

(If) Assume that $\Gamma \vdash e{:}A$ because $e = \mathsf{if}(e_1, e_2, e_3)$ and, by shorter inferences,

$\Gamma \vdash e_1{:}\mathsf{Bool}$, and

$\Gamma \vdash e_i{:}A$, $i = 2, 3$.

The induction hypothesis now yields that, for $i \in \{1, 2, 3\}$,

$$[\![e_i]\!]_\rho \quad \lhd \quad e_i\sigma \ . \tag{7.1}$$

We proceed with the proof by distinguishing three cases, depending on whether $[\![e_1]\!]_\rho$ is equal to $\bot$, tt or ff.

- If $[\![e_1]\!]_\rho = \bot$, then $[\![e]\!]_\rho = \bot$. By the definition of $\lhd$, it follows immediately that $[\![e]\!]_\rho = \bot \lhd e\sigma$, which was to be shown.

- If $[\![e_1]\!]_\rho = \mathit{tt}$, then $[\![e]\!]_\rho = [\![e_2]\!]_\rho$. In case $[\![e_2]\!]_\rho = \bot$, it follows immediately that
$$[\![e]\!]_\rho = \bot \lhd e\sigma \ .$$
Assume therefore that $[\![e_2]\!]_\rho \neq \bot$. In light of (7.1) and the definition of $\lhd$, it follows that $e_1\sigma \Downarrow \mathsf{true}$, and that $e_2\sigma \Downarrow v$ for some value v such that $[\![e_2]\!]_\rho \lhd v$. Collecting the above information, we now derive that
$$e\sigma = \mathsf{if}(e_1\sigma, e_2\sigma, e_3\sigma) \rightsquigarrow^* \mathsf{if}(\mathsf{true}, e_2\sigma, e_3\sigma) \rightsquigarrow e_2\sigma \rightsquigarrow^* v \ .$$
Thus $e\sigma \Downarrow v$ and $[\![e]\!]_\rho = [\![e_2]\!]_\rho \lhd v$. By the definition of the relation $\lhd$, we may now infer that $[\![e]\!]_\rho \lhd e\sigma$, which was to be shown.

- The case $[\![e_1]\!]_\rho = \mathit{ff}$ is similar to the one above.

(Object) Assume that $\Gamma \vdash e{:}A$ because $e = [\mathrm{l}_i{=}\varsigma(y_i{:}A)e_i \ ^{i\in I}]$, $A = [\mathrm{l}_i{:}B_i \ ^{i\in I}]$ and, by shorter inferences,
$$\Gamma, y_i{:}A \vdash b_i{:}B_i \tag{7.2}$$
for every $i \in I$. Using the definition of the denotational semantics we find that:

$$[\![e]\!]_\rho \quad = \quad \langle\!\langle \mathrm{l}_i = \lambda v.[\![b_i]\!]_{\rho\langle y_i \mapsto v\rangle} \ ^{i\in I} \rangle\!\rangle \ .$$

As the list $\tilde{a}$ only contains closed terms and y_i is different from all the variables in $\{x_1, \ldots, x_n\}$ by (7.2), we obtain that

$$e\sigma \quad = \quad [\mathrm{l}_i{=}\varsigma(y_i{:}A)b_i\sigma \ ^{i\in I}] \ .$$

Applying the inductive hypothesis to (7.2), we may now infer that, for every $i \in I$, $d' \in D$ such that $(d', d') \in [\![A]\!]$ and $a'{:}A$ with $d' \lhd a'$,

$$[\![b_i]\!]_{\rho[y_i \mapsto d']} \lhd b_i\{a'/\sigma\}[y_i \mapsto a'] \ .$$

By the definition of the relation $\lhd$, we finally conclude that

$$\begin{array}{rcl} [\![e]\!]_\rho & = & \langle\!\langle \mathrm{l}_i = \lambda v.[\![b_i]\!]_{\rho\langle y_i \mapsto v\rangle}{}^{\ i\in I}\rangle\!\rangle \\ & \lhd & [\mathrm{l}_i = \varsigma(y_i{:}A)e_i\sigma^{\ i\in I}] \\ & = & e\sigma \end{array}$$

which was to be shown.

(Select) Assume that $\Gamma \vdash e{:}B_j$ because $e = e'.\mathrm{l}_j$ and, by a shorter inference,

$$\Gamma \vdash e'{:}[\mathrm{l}_i{:}B_i^{\ i\in I}] \quad (j \in I) \ . \tag{7.3}$$

If $[\![e]\!]_\rho = \bot$, then $[\![e]\!]_\rho \lhd e\sigma$ is immediate from the definition of the relation $\lhd$. Assume therefore that $[\![e]\!]_\rho \neq \bot$. Applying the induction hypothesis to (7.3), we obtain that $[\![e']\!]_\rho \lhd e'\sigma$. Note now that, as $[\![e]\!]_\rho \neq \bot$, it must be the case that $[\![e']\!]_\rho \neq \bot$. Thus, by the definition of the relation $\lhd$, there is a value v of type $B = [\mathrm{l}_i{:}B_i^{\ i\in I}]$ such that $e'\sigma \Downarrow v$ and $[\![e']\!]_\rho \lhd v$. Again using the definition of $\lhd$, it must be the case that:

- $[\![e']\!]_\rho = \langle\!\langle \mathrm{l}_k = d_k^{\ k\in K}\rangle\!\rangle$, for some superset K of I,
- $v = [\mathrm{l}_k{=}\varsigma(y_k{:}B)b_k^{\ k\in K}]$, and
- for every $\hat{d} \in D$ and closed term $\hat{a}$ such that $(\hat{d}, \hat{d}) \in [\![B]\!]$, $\hat{a}{:}B$ and $\hat{d} \lhd \hat{a}$, it holds that $d_k(\hat{d}) \lhd b_k\{\hat{a}/y_k\}$ for every $k \in K$.

As $e'{:}B$ and $e' \Downarrow v$, it follows that $v{:}B$ (Lemma 7.3(2)). In light of (7.3), Theorem 7.6 yields that $([\![e']\!]_\rho, [\![e']\!]_\rho) \in [\![B]\!]$. Using the fact that $[\![e']\!]_\rho \lhd e'\sigma$, we may therefore derive that

$$\begin{array}{rcl} [\![e]\!]_\rho & = & [\![e']\!]_\rho(\mathrm{l}_j)([\![e']\!]_\rho) \\ & = & d_j([\![e']\!]_\rho) \\ & \lhd & b_j\{v/y_k\} \ . \end{array}$$

As $e\sigma = (e'\sigma).\mathrm{l}_j \rightsquigarrow^* v.\mathrm{l}_j \rightsquigarrow b_j\{v/y_k\}$ and $[\![e]\!]_\rho \lhd b_j\{v/y_k\}$, we may finally conclude that $[\![e]\!]_\rho \lhd e\sigma$, which was to be shown.

(UPDATE) Assume that $\Gamma \vdash e{:}A$ because $e = e'.\mathrm{l}_j \Leftarrow \varsigma(x{:}A)b : A$, $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$, $j \in I$ and, by shorter inferences,

$$\Gamma \vdash e'{:}A \tag{7.4}$$
$$\Gamma, x{:}A \vdash b{:}B_j \quad . \tag{7.5}$$

Using the definition of the denotational semantics, we obtain that

$$[\![e]\!]_\rho = [\![e']\!]_\rho \langle \mathrm{l}_j \mapsto \lambda u.[\![b]\!]_{\rho\langle x \mapsto u\rangle}\rangle \ .$$

As the terms in $\tilde{a}$ are closed, and x does not occur in $\tilde{x}$, we infer that

$$e\sigma = (e'\sigma).\mathrm{l}_j \Leftarrow \varsigma(x{:}A)b\sigma \ .$$

Applying induction to (7.4), we derive that

$$[\![e']\!]_\rho \lhd e'\sigma \ .$$

If $[\![e']\!]_\rho = \bot$, then $[\![e]\!]_\rho$ is also equal to $\bot$, and the claim follows immediately by the definition of $\lhd$. Assume therefore that $[\![e']\!]_\rho \neq \bot$. As $[\![e']\!]_\rho \lhd e'\sigma$, it must be the case that $e'\sigma \Downarrow v$ for some value v of type A such that $[\![e']\!]_\rho \lhd v$. As $A = [\mathrm{l}_i{:}B_i\ ^{i\in I}]$ and $[\![e']\!]_\rho \lhd v$, we obtain that

i. $[\![e']\!]_\rho = \langle\!\langle \mathrm{l}_k = d_k\ ^{k\in K}\rangle\!\rangle$, for some superset K of I,

ii. $v = [\mathrm{l}_k{=}\varsigma(y_k{:}A)b_k\ ^{k\in K}]$, and

iii. for every $\hat{d} \in D$ and closed term $\hat{a}$ such that $(\hat{d}, \hat{d}) \in [\![A]\!]$, $\hat{a} : A$ and $\hat{d} \lhd \hat{a}$, it holds that $d_k(\hat{d}) \lhd b_k\{\hat{a}/y_k\}$ for every $k \in K$.

Applying the inductive hypothesis to (7.5), we have that, whenever $(d', d') \in [\![A]\!]$, $a'{:}A$ and $d' \lhd a'$,

$$[\![b]\!]_{\rho\langle x \mapsto d'\rangle} \lhd b\sigma[x \mapsto a'] \ .$$

This yields, together with clause iii above, that

$$\begin{array}{rcl} [\![e]\!]_\rho & = & \langle\!\langle \mathrm{l}_j = \lambda u.[\![b]\!]_{\rho\langle x\mapsto u\rangle}, \mathrm{l}_k = d_k\ ^{k\in K\setminus\{j\}}\rangle\!\rangle \\ & \lhd & [\mathrm{l}_j{=}\varsigma(x{:}A)b\sigma,\ \mathrm{l}_k{=}\varsigma(y_k{:}A)b_k\ ^{k\in K\setminus\{j\}}] \ . \end{array}$$

Collecting the information on reductions that we have accumulated so far, we derive that

$$e\sigma \rightsquigarrow^* v.\mathrm{l}_j \Leftarrow \varsigma(x{:}A)b \rightsquigarrow [\mathrm{l}_j{=}\varsigma(x : A)b\sigma, \mathrm{l}_k{=}\varsigma(y_k{:}B)b_k\ ^{k\in K\setminus\{j\}}] \ .$$

Thus we may finally conclude that $[\![e]\!]_\rho \lhd e\sigma$, which was to be shown.

(FOLD) Assume that $\Gamma \vdash e{:}A$ because $e = \mathsf{fold}(A, e')$, $A = \mu(X)E$ and, by a shorter inference,

$$\Gamma \vdash e'{:}E\{{}^{A}/_{X}\} \ . \tag{7.6}$$

By the definition of the denotational semantics, we derive that

$$[\![e]\!]_\rho = \lambda u.\ [\![e']\!]_\rho \ .$$

The inductive hypothesis, applied to (7.6), yields

$$[\![e']\!]_\rho \lhd e'\sigma \ . \tag{7.7}$$

If $[\![e']\!]_\rho = \bot$, then $[\![e]\!]_\rho = \lambda u.\bot$. This is the least element of $D \to D$, and, by the definition of D, is identified with $\bot$. In this case, $[\![e]\!]_\rho \lhd e\sigma$ follows immediately. Assume therefore that $[\![e']\!]_\rho \neq \bot$. As $[\![e']\!]_\rho \lhd e'\sigma$, it must be the case that $e'\sigma \Downarrow v$ for some v such that $[\![e']\!]_\rho \lhd v$. As $e'\sigma \Downarrow v$, it follows immediately that $e\sigma = \mathsf{fold}(A, e'\sigma) \Downarrow \mathsf{fold}(A, v)$. Moreover, using clause (ii)(c) of the definition of $\lhd$ and (7.7), we conclude that $[\![e]\!]_\rho \lhd \mathsf{fold}(A, v)$, which was to be shown.

(UNFOLD) Assume that $\Gamma \vdash e{:}A$ because $e = \mathsf{unfold}(e')$, $A = E\{{}^{\mu(X)E}/_{X}\}$ and, by a shorter inference, $\Gamma \vdash e'{:}\mu(X)E$. The definition of the denotational semantics yields

$$[\![e]\!]_\rho = [\![e']\!]_\rho(\bot) \ .$$

If $[\![e]\!]_\rho = \bot$, then the claim is immediate. Assume therefore that $[\![e]\!]_\rho \neq \bot$. By induction, we may derive that

$$[\![e']\!]_\rho \lhd e'\sigma \ . \tag{7.8}$$

Note, furthermore, that $[\![e']\!]_\rho \neq \bot$. Thus (7.8) yields that $e'\sigma \Downarrow v$ for some value v of type $\mu(X)E$ such that $[\![e']\!]_\rho \lhd v$. The value v must be of the form $\mathsf{fold}(\mu(X)E, v')$. By the definition of $\lhd$, if $[\![e']\!]_\rho \lhd \mathsf{fold}(\mu(X)E, v')$ then $[\![e']\!]_\rho$ is of the form $\lambda u.d'$ for some $d' \lhd v'$. It is now easy to see that $e\sigma \Downarrow v'$ and that

$$[\![e]\!]_\rho = [\![e']\!]_\rho(\bot) = \lambda u.d'(\bot) = d' \ .$$

As $d' \lhd v'$, it follows that $[\![e]\!]_\rho \lhd e\sigma$, which was to be shown.

(SUBSUMP) Assume that $\Gamma \vdash e{:}A$ because, by shorter inferences, $\Gamma \vdash e{:}B$ and $\Gamma \vdash B <: A$. Then the inductive hypothesis immediately yields that $[\![e]\!]_\rho \lhd e\sigma$.

This completes the proof of this statement. □

Theorem 7.8 now follows immediately by the above statement and the definition of the formal approximation relation $\lhd$.

The following consequence of Proposition 7.7 and Theorem 7.8 will be useful in the remainder of this section.

Corollary 7.10 *Let* a:Bool. *Then* $a{\Downarrow}v$ *iff* $[\![a]\!] = [\![v]\!]$.

PROOF. The "only if" implication is just Proposition 7.7. To establish the "if" implication, assume that $[\![a]\!] = [\![v]\!]$ and a:Bool. As $[\![a]\!] \neq \bot$, Theorem 7.8 yields that $a{\Downarrow}v'$ for some value v' of type Bool. By Proposition 7.7, it follows that $[\![v]\!] = [\![v']\!]$. As v and v' are of type Bool, this entails that $v = v'$ and thus that $a{\Downarrow}v$. □

We are now in a position to prove the main result of this chapter, viz. that the denotational semantics is correct with respect to observational congruence.

Theorem 7.11 *Let* $A \in \mathbf{Type}$ *and* a, b:A. *Then*

$$([\![a]\!], [\![b]\!]) \in [\![A]\!] \text{ implies } a_A \overset{\mathsf{Bool}}{\simeq} b_A \ .$$

PROOF. Assume that $A \in \mathbf{Type}$, a, b:A and $([\![a]\!], [\![b]\!]) \in [\![A]\!]$. In light of [GR96, Theorem"nobreakspace –"2], to prove that $a_A \overset{\mathsf{Bool}}{\simeq} b_A$ it is sufficient to show that $a_A \sim b_A$ holds. Let $\mathcal{X} = \{(a_A, b_A) \mid ([\![a]\!], [\![b]\!]) \in [\![A]\!]\}$. We prove that $\mathcal{X}$ is a bisimulation. To this end, assume that $(a_A, b_A) \in \mathcal{X}$ and $a_A \xrightarrow{\alpha} a'_{A'}$. By symmetry it is enough to prove that $b_A \xrightarrow{\alpha} b'_{A'}$ for some b':A' such that $(a'_{A'}, b'_{A'}) \in \mathcal{X}$. The proof of this claim proceeds by case analysis of the transition rule used in inferring the transition $a_A \xrightarrow{\alpha} a'_{A'}$.

(TRANS BOOL) Then $\alpha = v$ where $a{\Downarrow}v \in \{\mathsf{true}, \mathsf{false}\}$, $A = \mathsf{Bool}$, $A' = \mathsf{Top}$ and $a' = a$. Recall that $[\![\mathsf{Bool}]\!] = \{(\bot, \bot), (\mathit{tt}, \mathit{tt}), (\mathit{ff}, \mathit{ff})\}$ and that, for all programs a:Bool, $[\![a]\!] = [\![v]\!]$ iff $a{\Downarrow}v$ (Corollary 7.10). As $([\![a]\!], [\![b]\!]) \in [\![\mathsf{Bool}]\!]$ this implies that $[\![b]\!] = [\![v]\!]$. Again by Corollary 7.10, it follows that $b{\Downarrow}v$, and therefore that $b_{\mathsf{Bool}} \xrightarrow{v} b_{\mathsf{Top}}$. Furthermore a:Top, b:Top and $([\![a]\!], [\![b]\!]) \in [\![\mathsf{Top}]\!]$, i.e. $(a_{\mathsf{Top}}, b_{\mathsf{Top}}) \in \mathcal{X}$.

(TRANS SELECT) In this case $A = [\mathrm{l}_i{:}B_i \ ^{i\in I}]$, $\alpha = \mathrm{l}_j$, $a' = a.\mathrm{l}_j$ and $A' = B_j$ for some $j \in I$. As b:A we also have that $b_A \xrightarrow{\mathrm{l}_j} b.\mathrm{l}_{j\,B_j}$. By the type assignment rule (SELECT), $a.\mathrm{l}_j$:B_j and $b.\mathrm{l}_j$:B_j. Furthermore, by the equational rule (EQ SELECT) and the soundness of the equational theory, $([\![a.\mathrm{l}_j]\!], [\![b.\mathrm{l}_j]\!]) \in [\![B_j]\!]$. This proves that $(a.\mathrm{l}_{j\,B_j}, b.\mathrm{l}_{j\,B_j}) \in \mathcal{X}$.

(TRANS UPDATE) In this case $A = A' = [\mathrm{l}_i{:}B_i \ ^{i\in I}]$, $x{:}A \vdash e{:}B_j$, $\alpha = \mathrm{l}_j{\Leftarrow}\varsigma(x)e$ and $a' = a.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)e$. Also $b_A \xrightarrow{\mathrm{l}_j{\Leftarrow}\varsigma(x)e} b'_A$ where $b' = b.\mathrm{l}_j{\Leftarrow}\varsigma(x{:}A)e$. By the type assignment rule (UPDATE), a':A and b':A.

By the equational theory $x{:}A \vdash e{:}B_j$ implies $x{:}A \vdash e \leftrightarrow e : B_j$. Therefore, using the equational rule (Eq Override) and the soundness of the equational theory with respect to the model (Theorem 7.6), we infer that $(\llbracket a' \rrbracket, \llbracket b' \rrbracket) \in \llbracket A \rrbracket$. This proves that $(a'_A, b'_A) \in \mathcal{X}$.

(Trans Unfold) Then $A = \mu(X)B$, $C = B\{{}^{A}\!/_{X}\}$, $\alpha = \mathsf{unfold}$, $a' = \mathsf{unfold}(a)_C$. By assumption $a{:}A$ and therefore the type assignment rule (Fold) implies that $\mathsf{unfold}(a){:}B\{{}^{A}\!/_{X}\}$ and $\mathsf{unfold}(b){:}B\{{}^{A}\!/_{X}\}$. Furthermore by the equational rule (Eq Unfold) and soundness of the equational theory, we have $(\llbracket \mathsf{unfold}(a) \rrbracket, \llbracket \mathsf{unfold}(b) \rrbracket) \in \llbracket C \rrbracket$. This proves that $(\mathsf{unfold}(a)_C, \mathsf{unfold}(b)_C) \in \mathcal{X}$.

□

To see that the denotational model is not fully abstract, consider the following two objects (from [AC96]) of type $B = [\mathrm{l}_2{:}\mathsf{Bool}]$:

$$a = [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2{=}\mathsf{true}] \qquad b = [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2{=}\varsigma(x{:}[\mathrm{l}_1{:}\mathsf{Bool},\ \mathrm{l}_2{:}\mathsf{Bool}])x.\mathrm{l}_1]$$

where we have omitted the ς-binder in the methods that do not use self.

We shall now argue that $(\llbracket a \rrbracket, \llbracket b \rrbracket) \notin \llbracket B \rrbracket$. The denotations of a and b are:

$$\llbracket a \rrbracket \ = \ \langle\!\langle \mathrm{l}_1{=}\lambda(v)\mathit{tt},\ \mathrm{l}_2{=}\lambda(v)\mathit{tt} \rangle\!\rangle$$

and

$$\llbracket b \rrbracket \ = \ \langle\!\langle \mathrm{l}_1{=}\lambda(v)\mathit{tt},\ \mathrm{l}_2{=}\lambda(v)v(\mathrm{l}_1)v \rangle\!\rangle$$

Let $b^* = [\mathrm{l}_1{=}\mathsf{false},\ \mathrm{l}_2{=}\mathsf{true}]$. As b^* is a program of type B, Theorem 7.6 yields that $(\llbracket b^* \rrbracket, \llbracket b^* \rrbracket) \in \llbracket B \rrbracket$. If $(\llbracket a \rrbracket, \llbracket b \rrbracket) \in \llbracket B \rrbracket$, by Lemma 7.5 we would then be able to infer that

$$(\llbracket a \rrbracket(\mathrm{l}_2)\llbracket b^* \rrbracket, \llbracket b \rrbracket(\mathrm{l}_2)\llbracket b^* \rrbracket) \in \llbracket \mathsf{Bool} \rrbracket \ .$$

However, this is obviously not the case, because the denotation of b^* is

$$\langle\!\langle \mathrm{l}_1{=}\lambda(v)\mathit{ff},\ \mathrm{l}_2{=}\lambda(v)\mathit{tt} \rangle\!\rangle$$

and therefore

$$\llbracket a \rrbracket(\mathrm{l}_2)\llbracket b^* \rrbracket = \mathit{tt} \text{ and } \llbracket b \rrbracket(\mathrm{l}_2)\llbracket b^* \rrbracket = \mathit{ff} \ .$$

As a corollary of Theorem 7.11, we obtain an alternative proof of the weak soundness of the equational theory for $\mathbf{Ob}_{1<:\mu}$w.r.t. bisimulation equivalence, a result originally due to Gordon and Rees.

Proposition 7.12 *If $\emptyset \vdash a \leftrightarrow b : A$, then $a_A \sim b_A$.*

Proof. Suppose $\emptyset \vdash a \leftrightarrow b : A$. The soundness of the equational theory in the denotational model implies that $(\llbracket a \rrbracket, \llbracket b \rrbracket) \in \llbracket A \rrbracket$ and Theorem 7.11 in turn implies that $a_A \overset{\mathsf{Bool}}{\simeq} b_A$. As $\overset{\mathsf{Bool}}{\simeq}$ and $\sim$ coincide [GR96], the result now follows. □

7.4 Final comments

In this chapter we have shown that the denotational model proposed by Abadi and Cardelli [AC96] is correct, but not fully abstract with respect to the reduction semantics. The chapter is based on the paper [AHIK97], which is joint work between Luca Aceto, Anna Ingólfsdóttir, Hans Hüttel and myself, which was presented at Express'97 held in Santa Margrerita Ligure, Italy.

Chapter 8

Relation between Modal Logic and Types

In this chapter we use the labelled transition system of Section 7.1 to build a modal logic for describing dynamic properties of terms in the object calculus. By dynamic properties we mean properties specifically related to the behavior over time of a term. We also examine the relation between the type system with recursive types $\mathbf{Ob}_{1<:\mu}$ for the ς-calculus and the modal μ-calculus [Koz83]. It turns out that there are close correspondences between the type system and a fragment of the μ-calculus using only maximal fixed-points. In particular, there is a sound and complete translation from types to logical formulae preserving typability and the subtype ordering, when we interpret subtyping as containment. Phrased differently, the translation establishes a Curry-Howard-style result in that it allows us to view ς-calculus types as realizers of certain μ-calculus formulae.

8.1 A logic for objects

In the following we shall consider the modal μ-calculus [Koz83] interpreted over the labelled transition system defined in the Section 7.1. Later we shall show that one of its sublogics corresponds to the $\mathbf{Ob}_{1<:\mu}$ type system in a precise sense.

The modal μ-calculus was introduced by Kozen in [Koz83]. Taking actions as propositions, the modal μ-calculus corresponds to the logic introduced by Hennessy and Milner and [HM85] extended with recursive definitions [Lar90].

8.1.1 Syntax and informal semantics

The set of μ-calculus formulae **Form** is given in Table 8.1

$$\begin{array}{rcl} F & ::= & F_1 \vee F_2 \mid F_1 \wedge F_2 \mid \langle\alpha\rangle F \mid [\alpha]F \mid X \mid \nu X.F \mid \mu X.F \\ & \mid & \mathit{tt} \mid \mathit{ff} \\ \alpha & ::= & \mathrm{l} \mid \mathsf{unfold} \mid \mathrm{l}{\Leftarrow}\varsigma(x)b \end{array}$$

Table 8.1: The syntax of μ-calculus formulae.

Here X ranges over the set of formula recursion variables **FormVar**. tt and ff are atomic formulae, not to be confused with the boolean values of the ς-calculus. The modalities of the logic are indexed by the observations from the labelled transition semantics. Intuitively, a formula $\langle\alpha\rangle F$ is true for an object a_A if a_A allows *some* observation α such that F is true for the resulting object. Similarly, a formula $[\alpha]F$ is true for the object a if *all* observations of α will result in an object for which F is true.

8.1.2 Semantics of the μ-calculus

We interpret our logic over the labelled transition system of the previous section. If a formula F is true for an object a_A, we say that a_A *satisfies* F and write that $a_A \models F$.

The denotation of a formula F is the set of proved programs (pairs of a closed object and its type) that satisfy F. As formulae may contain free variables, the denotation of a formula is seen relative to an environment $\sigma : \mathbf{FormVar} \hookrightarrow \mathcal{P}(\mathbf{Prog})$ which for a given variable, returns the set of proved programs which satisfies the formula bound to that variable.

We can extend operations and predicates on sets of objects to environments. For any two environments σ_1, σ_2 we write $\sigma_1 \subseteq \sigma_2$ iff for all variables we have that $\sigma_1(X) \subseteq \sigma_2(X)$. If S is a set of objects, we write $\sigma \subseteq S$ if for all X we have that $\sigma(X) \subseteq S$. Similarly, $\sigma_1 \cup \sigma_2$ is the environment σ such that $\sigma(X) = \sigma_1(X) \cup \sigma_2(X)$.

The semantics of formulae not using the recursion connectives can then be defined as shown in Table 8.2.

The operators $\nu X.F$ and $\mu X.F$ are recursion operators. For any recursive formula $\nu X.F$ or $\mu X.F$ we define a *declaration function*

$$\mathcal{D}_F : (\mathbf{FormVar} \hookrightarrow \mathcal{P}(\mathbf{Prog})) \to \mathcal{P}(\mathbf{Prog}) \text{ by } \mathcal{D}_F(\sigma) = [\![F]\!]\sigma$$

$$
\begin{array}{rcl}
[\![tt]\!]\sigma & = & \mathbf{Prog} \\
[\![ff]\!]\sigma & = & \emptyset \\
[\![F_1 \vee F_2]\!]\sigma & = & [\![F_1]\!]\sigma \cup [\![F_2]\!]\sigma \\
[\![F_1 \wedge F_2]\!]\sigma & = & [\![F_1]\!]\sigma \cap [\![F_2]\!]\sigma \\
[\![X]\!]\sigma & = & \sigma(X) \\
[\![\langle\alpha\rangle F]\!]\sigma & = & \{a_A \mid \exists b_B \text{ s.t. } a_A \xrightarrow{\alpha} b_B \text{ and } b_B \in [\![F]\!]\sigma\} \\
[\![[\alpha] F]\!]\sigma & = & \{a_A \mid \forall b_B \text{ with } a_A \xrightarrow{\alpha} b_B : b_B \in [\![F]\!]\sigma\}
\end{array}
$$

Table 8.2: The semantics of formulae not using recursion.

Both recursion operators denote a solution to the equation $X = F$, that is, we want an environment σ such that $[\![X]\!]\sigma = [\![F]\!]\sigma$. A σ with this property is called a *model*.

One may easily show that $(\mathbf{FormVar} \hookrightarrow \mathcal{P}(\mathbf{Prog}), \subseteq)$, the set of environments ordered under inclusion, constitutes a complete lattice and that the function $\mathcal{D}_F$ is a monotonic function for any recursive formula $\nu X.F$ or $\mu X.F$. Consequently, Tarski's fixed point theorem [Tar55] for complete lattices and monotonic functions, guarantees that models always exist for any recursive formula.

Theorem 8.1 (Maximal and minimal model) *Given a recursive formula F of the form $\nu X.F$ or $\mu X.F$, there exist models σ_{max} and σ_{min} given by:*

$$\sigma_{max} = \bigcup \{\sigma \mid \sigma \subseteq \mathcal{D}_F(\sigma)\}$$

$$\sigma_{min} = \bigcap \{\sigma \mid \mathcal{D}_F(\sigma) \subseteq \sigma\}$$

σ_{max} is the maximal model w.r.t. $\subseteq$ and σ_{min} is the minimal model w.r.t. $\subseteq$.

The ν-connective is taken to indicate that we want the model σ_{max}, whereas the μ-connective is taken to indicate that we want the model σ_{min} and thus the semantics of the recursion operators is

$$[\![\nu X.F]\!]\sigma = \sigma_{max}(X)$$
$$[\![\mu X.F]\!]\sigma = \sigma_{min}(X)$$

From Theorem 8.1 we obtain the following definition of pre- and post-models.

Definition 8.2 (Pre- and post-models) *Given a formula $\mu X.F$ or $\nu X.F$, an environment σ is a* pre-model *if $\sigma \in \{\sigma \mid \mathcal{D}_F(\sigma) \subseteq \sigma\}$. σ is a* post-model *if $\sigma \in \{\sigma \mid \sigma \subseteq \mathcal{D}_F(\sigma)\}$.*

Consequently, Theorem 8.1 says that the semantics of a ν-formula is the union of all its post-models and that the semantics of a μ-formula is the intersection of all its pre-models.

This logic fully characterizes Gordon-Rees bisimulation; that is two objects are bisimilar if and only if they satisfy the same logical formulae.

Theorem 8.3 (Characterization) *Assume that $a_A, b_A \in \mathbf{Prog}$. Let $\mathbf{Form}^-$ denote the set of formulae generated from the grammar in Table 8.1 without the use of box-modalities. Then,*

$$\begin{array}{l} a_A \sim b_A \\ \Updownarrow \\ \forall F \in \mathbf{Form}^- : a_A \models F \Leftrightarrow b_A \models F \end{array}$$

The reason why we can disregard the box-modalities is that the labelled transitions system is deterministic, the only reason why we have included then in the logic is because they allow easier specifications of properties. The proof of Theorem 8.3, which is lengthy but standard, is omitted; it can be found in [AP96]. Observe, that the theorem also holds in the full logic **Form**.

8.2 Specifying objects

The modal μ-calculus is very powerful when used as a temporal logic of labelled transition systems. It is well-known that the temporal modalities of the propositional branching time temporal logic CTL [Eme94] are expressible within the alternation free fragment of the μ-calculus (in fact, it can be shown that all of CTL* [Eme94] can be expressed within the μ-calculus).

In this short section we shall describe how properties of ς-calculus can be described this way. We let the set **Act** denote the set of possible observations; we write $[\mathbf{Act}]F$ as an abbreviation of $\bigwedge_{\alpha \in \mathbf{Act}} [\alpha]F$ and similarly, we write $\langle \mathbf{Act} \rangle F$ as an abbreviation of $\bigvee_{\alpha \in \mathbf{Act}} \langle \alpha \rangle F$.

The CTL temporal modalities can be defined as

$$
\begin{aligned}
\mathsf{AG}F &= \nu X.(F \wedge [\mathbf{Act}]X) \\
\mathsf{EG}F &= \nu X.(F \wedge ([\mathbf{Act}]ff \vee \langle \mathbf{Act} \rangle X)) \\
\mathsf{EF}F &= \mu X.(F \vee \langle \mathbf{Act} \rangle X) \\
\mathsf{AF}F &= \mu X.(F \vee (\langle \mathbf{Act} \rangle tt \wedge [\mathbf{Act}]X)) \\
\mathsf{U}^s(F,G) &= \mu X.(G \vee (F \wedge \langle \mathbf{Act} \rangle tt \wedge [\mathbf{Act}]X)) \\
\mathsf{U}^w(F,G) &= \nu X.(G \vee (F \wedge [\mathbf{Act}]X))
\end{aligned}
$$

The intuitive interpretation of these modalities is

$\mathsf{AG}F$**:** An object a satisfies $\mathsf{AG}F$, if F is satisfied by all states of any transition path of a.

$\mathsf{EG}F$**:** $\mathsf{EG}F$ is satisfied by an object a if there exists some transition path of a such that F is satisfied by all states of the path.

$\mathsf{EF}F$**:** The dual of $\mathsf{AG}F$. An object a satisfies $\mathsf{EF}F$ if F is satisfied by some state along some transition path.

$\mathsf{AF}F$**:** Guarantees that F will sooner or later be true along any transition path.

$\mathsf{U}^s(F,G), \mathsf{U}^w(F,G)$**:** The meanings of $\mathsf{U}^s(F,G)$ and $\mathsf{U}^w(F,G)$ are almost identical. The idea is that an object a satisfies $\mathsf{U}^s(F,G)$ or $\mathsf{U}^w(F,G)$ if F is satisfied by a until G at some point is satisfied. The difference is that $\mathsf{U}^w(F,G)$ does not guarantee that G will ever be satisfied. $\mathsf{U}^s(F,G)$ is called *strong until*, and $\mathsf{U}^w(F,G)$ *weak until*.

Notice that invariance properties are expressed using maximal fixed-points, whereas eventuality properties are expressed using minimal fixed-points.

The following example illustrates how an object can be specified using the CTL modalities:

Example 8.1 In [AC96] a ***calculator*** object is described. The behavior of the calculator can informally be stated as follows: Either one of the methods enter, add or sub can be invoked, which will result in a new calculator, or the method equals can be invoked resulting in termination. In the following let **Act** be defined as $\mathbf{Act} = \{\text{enter}, \text{add}, \text{sub}\}$.

The formula F shown below specifies the observations that must be allowed by a ***calculator*** object of recursive type. The formula G describes

what must always hold about the methods enter, add and sub of the ***calculator*** object:

$$\begin{aligned} F &= \langle \mathsf{unfold} \rangle tt \wedge [\mathbf{Act}] ff, \\ G &= \langle \mathrm{add} \rangle tt \vee \langle \mathrm{sub} \rangle tt \vee \langle \mathrm{enter} \rangle tt \wedge [\mathsf{unfold}] ff. \end{aligned}$$

To express that it holds that F is satisfied until G at some point will be satisfied we use the connective "strong until" and write:

$$\mathsf{U}^s(F, G).$$

We write that $\mathsf{U}^s(F, G)$ must always hold as

$$\mathsf{AGU}^s(F, G)$$

The specification of the ***calculator*** object is a combination of $\mathsf{AGU}^s(F, G)$ and the specification for the equals method:

$$F_{eq} = \langle \mathsf{unfold} \rangle \langle \mathrm{equals} \rangle tt$$

The final correctness specification looks as follows:

$$\mathsf{AGU}^s(F, G) \vee F_{eq}.$$

□

8.3 Types as logical formulae

In this section we shall show an intimate correspondence between the types of $\mathbf{Ob}_{1<:\mu}$ and formulae of a subset of our modal logic, in that we interpret $\mathbf{Ob}_{1<:\mu}$ types as formulae. Inhabiting a type then corresponds to satisfying the corresponding formula in modal logic, and subtyping corresponds to implication between modal properties.

8.3.1 The ν-calculus and its semantics

From the point of view of temporal logic, types are certain invariance properties. When relating types and logical properties, we are therefore interested in *maximal* models. The sublogic that we shall consider therefore only has the ν-connective.

$$
\begin{array}{rcl}
F & ::= & F_1 \wedge F_2 \mid \langle\alpha\rangle F \mid X \mid \nu X.F \\
 & \mid & \langle \mathsf{true}\rangle \mathit{tt} \mid \langle \mathsf{false}\rangle \mathit{tt} \mid \mathit{tt} \mid \mathit{ff} \mid \mathsf{isBool} \\
\alpha & ::= & \mathrm{l} \mid \mathsf{unfold} \mid \mathrm{l}{\Leftarrow}\varsigma(x)b
\end{array}
$$

In order to be able to express the typings of objects in our modal logic we have introduced the atomic predicate isBool, which is the logical formula corresponding to Bool. The predicate isBool is satisfied by an object a if a has type Bool.

The reason for introducing this predicate is that the μ-calculus connectives alone cannot express that an object of type Bool will allow precisely the observations true or false – in order to achieve this, we would need infinite conjunctions. Further, the fact that an object of type Bool can diverge in the reduction semantics cannot be expressed. In other words, it seems reasonable that the base types should be modelled by atomic formulae.

Moreover, we now only consider closed formulae as the recursion operator on types is interpreted using the ν-connective. In what follows, this allows us to formulate and employ an alternative definition of the semantics, namely a coinductive definition in terms of satisfaction relations.

Definition 8.4 *A* satisfaction relation *is a relation* $\mathcal{S} \subseteq \mathbf{Prog} \times \mathbf{Forms}^{-}$ *for which it holds that*

$a_A \mathcal{S}\ \mathit{tt}$		*for any* $a_A \in \mathbf{Prog}$
$a_A \mathcal{S}$ isBool		*for any* a:Bool
$a_A \mathcal{S}\ F_1 \wedge F_2$	*implies*	$a_A \mathcal{S}\ F_1$ *and* $a_A \mathcal{S}\ F_2$
$a_A \mathcal{S}\ F_1 \vee F_2$	*implies*	$a_A \mathcal{S}\ F_1$ *or* $a_A \mathcal{S}\ F_2$
$a_A \mathcal{S}\ \langle\alpha\rangle F$	*implies*	$\exists b_B$ *s.t.* $a_A \xrightarrow{\alpha} b_B$ *and* $b_B \mathcal{S}\ F$
$a_A \mathcal{S}\ \nu X.F$	*implies*	$a_A \mathcal{S}\ F\{{}^{\nu X.F}/_{X}\}$

and no pairs on the form (a_A, ff) *are in* $\mathcal{S}$.

We now have the following result.

Proposition 8.5 *If* F *is a closed* ν*-calculus formula, then* $a_A \models F$ *iff there exists a satisfaction relation* $\mathcal{S}$ *such that* $a_A \mathcal{S}\ F$.

Proof. The *only if*-part of the proposition follows from the fact that $\models$ is a satisfaction relation. For the *if*-part, one proceeds by induction on the structure of F. The only interesting case is that of recursive formulae $\nu X.F_1$.

Here notice that if we let $S_X = \{a_A \mid a_A \mathcal{S} \nu X.F_1\}$, then $S_X \subseteq [\![F]\!]_{[X \mapsto S_X]}$; in other words, the environment $[X \mapsto S_X]$ is a post-model. □

According to the above proposition, if we wish to show that $a_A \models F$, then it is enough to determine a satisfaction relation $\mathcal{S}$ for which $a_A \mathcal{S} F$.

8.3.2 An interpretation of types in the ν-calculus

The ν-calculus is, in fact, too large – only certain formula can occur as the translations of $\mathbf{Ob}_{1<:\mu}$ types. We shall call such formulae *type formulae*. The abstract syntax of type formulae is given by the grammar

$$F \ ::= \ \# \mid \bigwedge_{i \in I} \langle \mathrm{l}_i \rangle F_i \mid \mathsf{isBool} \mid X \mid \nu X.(\langle \mathsf{unfold} \rangle F)$$

We denote the set of type formulae by $\mathbf{Form}_t$.

We are now able to introduce the translation $\mathcal{T} : \mathbf{Type} \to \mathbf{Form}_t$ from types to type formulae as follows:

$$\begin{aligned}
\mathcal{T}(\mathsf{Top}) &= \# \\
\mathcal{T}(\mathsf{Bool}) &= \mathsf{isBool} \\
\mathcal{T}([\mathrm{l}_i{:}A_i \ ^{i \in I}]) &= \bigwedge_{i \in I} \langle \mathrm{l}_i \rangle \mathcal{T}(A_i) \\
\mathcal{T}(\mu(X)A) &= \nu X.(\langle \mathsf{unfold} \rangle \mathcal{T}(A)) \\
\mathcal{T}(X) &= X
\end{aligned}$$

Not surprisingly, the type Top is assigned the formulae $\#$, since all typable objects have type (or supertype) Top. The type Bool is as a special case translated to the atomic formula isBool. The μ-calculus translation of an object type reflects the possible method invocations that can be performed with respect to objects of the object type. The specification for a recursive object type is an invariance property. It states that it must *always* be possible to perform an unfold-transition leading to the specification for an object type. As recursive types are interpreted as invariants, we have to use the maximal fixed point operator. In other words, the μ becomes a ν when passing from types to formulae.

The translation defined by $\mathcal{T}$ is similar to the notion of *characteristic formula* [IS94] for the typings of objects. It is not the case, though, that these characteristic formulae express all possible behaviors of objects. In particular, it is not possible to prove that the logic for object types fully characterizes the bisimulation of Gordon and Rees, as the types and thus the

type formulae say nothing about the possibility of method overrides. (This is exemplified in Example 8.2 towards the end of this section.)

8.3.3 Soundness and completeness of the translation

The translation presented here is correct in a very precise sense. In order to express this, we need to formulate the logical counterparts of the typability notions. Definition 8.6 expresses the notion of subtyping with respect to the modal formulae for types.

Definition 8.6 *Let $A, B \in \mathbf{Type}$ and Γ some well formed environment. We say that Γ models the subtyping relation $A <: B$ written*

$$\Gamma \models A <: B$$

if for some post-model σ it holds that $[\![\mathcal{T}(A)]\!]\sigma \subseteq [\![\mathcal{T}(B)]\!]\sigma$ and $\sigma(X) \subseteq [\![\mathcal{T}(C)]\!]\sigma$ for all $(X <: C) \in \Gamma$.

Theorem 8.7 relates the standard subtype judgments, which are of the form $\Gamma \vdash A <: B$, to the inclusion relation between logical properties.

Theorem 8.7 *Let $A, B \in \mathbf{Type}$, Γ some well formed environment and $\Gamma \vdash A <: B$. Then $\Gamma \models A <: B$.*

Proof. By induction on the proof tree of $\Gamma \vdash A <: B$, that is, by inspecting the rules for subtyping (c.f. Table 3.8).

(Sub Refl): It follows directly that $[\![\mathcal{T}(A)]\!]\sigma \subseteq [\![\mathcal{T}(A)]\!]\sigma$.

(Sub Top): Since $[\![\mathcal{T}(\mathsf{Top})]\!]\sigma = \mathbf{Prog}$ we have $[\![\mathcal{T}(A)]\!]\sigma \subseteq [\![\mathcal{T}(\mathsf{Top})]\!]\sigma$ for all types $A \in \mathbf{Type}$.

(Sub Object): Let $[\mathrm{l}_i{:}A_i\ ^{i\in I}] <: [\mathrm{l}_i{:}A_i\ ^{i\in J}]$ for some indexing sets I, J for which it holds that $J \subseteq I$. By definition of $\mathcal{T}$, $\mathcal{T}([\mathrm{l}_i{:}A_i\ ^{i\in I}])$ must impose at least the same restrictions on objects as $\mathcal{T}([\mathrm{l}_i{:}A_i\ ^{i\in J}])$. It follows that $[\![\mathcal{T}([\mathrm{l}_i{:}A_i\ ^{i\in I}])]\!]\sigma \subseteq [\![\mathcal{T}([\mathrm{l}_i{:}A_i\ ^{i\in J}])]\!]\sigma$.

(Sub X): Follows from the well formedness of Γ.

(Sub Rec): Assume there is a post-model σ such that $\sigma(X) \subseteq [\![\mathcal{T}(C)]\!]\sigma$ for all $(X <: C) \in \Gamma$, $\sigma(X_1) \subseteq \sigma(X_2)$ and $[\![\mathcal{T}(A)]\!]\sigma \subseteq [\![\mathcal{T}(B)]\!]\sigma$. We must find a post-model σ' such that $\sigma'(X) \subseteq [\![\mathcal{T}(C)]\!]\sigma'$ for all $(X <: C) \in \Gamma$ and $[\![\mathcal{T}(\mu(X_1)A)]\!]\sigma' \subseteq [\![\mathcal{T}(\mu(X_2)B)]\!]\sigma'$. Take σ' defined by

$$\sigma'(X) = \begin{cases} \sigma(X_1) \cup [\![\mathcal{T}(A)]\!]\sigma & \text{if } X = X_1 \\ \sigma(X_2) \cup [\![\mathcal{T}(B)]\!]\sigma & \text{if } X = X_2 \\ \sigma(X) & \text{otherwise} \end{cases}$$

By the assumptions $[\![\mathcal{T}(\mu(X_1)A)]\!]\sigma' \subseteq [\![\mathcal{T}(\mu(X_2)B)]\!]\sigma'$. Thus, we only need to check that σ' is a post-model, i.e. that

$$\sigma'(X_1) \subseteq [\![\mathcal{T}(A)]\!]\sigma' \text{ and } \sigma'(X_2) \subseteq [\![\mathcal{T}(B)]\!]\sigma'$$

By definition of σ', $\sigma(X) \subseteq \sigma'(X)$ for all X. Then, since $[\![\cdot]\!]$ is monotonic on σ it must hold that $[\![F]\!]\sigma \subseteq [\![F]\!]\sigma'$ for all $F \in \mathbf{Form}_t$.

□

We now show that our translation is sound and complete with respect to the usual typing of objects.

Soundness is stated by the following theorem:

Theorem 8.8 (Soundness) *Let $a_A \in \mathbf{Prog}$ then $a_A \models \mathcal{T}(A)$.*

Proof. The relation

$$\begin{aligned}\mathcal{S}_T &= \{(a_A, \mathcal{T}(A)) \mid a_A \in \mathbf{Prog}\}\\ &\cup \{(a_A, \langle \mathrm{l}_i\rangle \mathcal{T}(A_i)) \mid A = [\mathrm{l}_i{:}A_i\ ^{i\in I}]\}\\ &\cup \{(a_A, \langle \mathsf{unfold}\rangle \mathcal{T}(B\{^{\mu(X)B}/_X\})) \mid A = \mu(X)B\}\\ &\cup \{(a_A, \mathsf{isBool}) \mid a{:}\mathsf{Bool}\}\end{aligned}$$

is a satisfaction relation. □

Completeness here means that if a typable object in $\mathbf{Ob}_{1<:\mu}$ satisfies a type formula, then it also has the corresponding type:

Theorem 8.9 (Completeness) *Let $a_B \in \mathbf{Prog}$ and $F \in \mathbf{Form}_t$. If $a \in [\![F]\!]\sigma$ then there exists a type A such that $\emptyset \vdash a{:}A$, where $\mathcal{T}(A) = F$.*

Proof. By induction on the structure of F.

(Top): If $F = \mathit{tt}$ then it must hold that $a{:}\mathsf{Top}$, where $\mathcal{T}(\mathsf{Top}) = \mathit{tt}$.

(Bool): If $F = \mathsf{isBool}$ then, obviously, $a{:}\mathsf{Bool}$, where $\mathcal{T}(\mathsf{Bool}) = \mathsf{isBool}$.

(Object): Assume that

$$F = \bigwedge_{i\in I} \langle \mathrm{l}_i\rangle \mathcal{T}(A_i)$$

It must be the case that a can perform the transitions $a_B \xrightarrow{\mathrm{l}_i} (a.\mathrm{l}_i)_{A_i}$, where $(a.\mathrm{l}_i)_{A_i} \in [\![\mathcal{T}(A_i)]\!]\sigma$, for all $i \in I$. That is, a must at least have

the type $[\mathrm{l}_i{:}A_i\ ^{i\in I}]$. We do not know if a may have further possibilities for transitions, $a_B \xrightarrow{\mathrm{l}_j} (a.\mathrm{l}_j)_{A_j}$ for $j \in J$, $I \cap J = \emptyset$, which results in the typing

$$[\mathrm{l}_i{:}A_i, \mathrm{l}_j{:}A_j\ ^{i\in I, j\in J}]$$

Now, the result follows using the subtype rule (Sub Obj) for objects:

$$[\mathrm{l}_i{:}A_i, \mathrm{l}_j{:}A_j\ ^{i\in I, j\in J}] <: [\mathrm{l}_i{:}A_i\ ^{i\in I}]$$

(Unfold): Assume that

$$F = \nu X.(\langle \mathsf{unfold} \rangle \mathcal{T}(A))$$

Then, obviously, $a{:}\mu(X)A$.

(Var): Trivial.

□

As mentioned earlier, the characteristic formulae for object types do not fully characterize the bisimulation equivalence of objects due to Gordon and Rees. This is due to the fact that the typing of an object does not take into account the possible method overrides that can be performed on the object. In fact it is possible, with respect to an object, to do infinitely many method overrides. That is, the ς-calculus is not finite branching. This lack of information about the possible behavior of an object carries through to the type formulae.

Example 8.2 Consider the two objects:

$$\begin{array}{rcl} a & = & [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2 = \mathsf{true}] \\ b & = & [\mathrm{l}_1{=}\mathsf{true},\ \mathrm{l}_2{=}\varsigma(x{:}A)x.\mathrm{l}_1] \end{array}$$

Both have the type

$$A \ = \ [\mathrm{l}_1{:}\mathsf{Bool}, \mathrm{l}_2{:}\mathsf{Bool}],$$

and supertypes B and C:

$$\begin{array}{rcl} B & = & [\mathrm{l}_1{:}\mathsf{Bool}], \\ C & = & [\mathrm{l}_2{:}\mathsf{Bool}] \end{array}$$

It can be shown that $a \sim_B b$ and $a \sim_C b$ but $a \not\sim_A b$, since after the method overrides $a.\mathrm{l}_1 \Leftarrow \varsigma(x)x.\mathrm{l}_2$ and $b.\mathrm{l}_1 \Leftarrow \varsigma(x)x.\mathrm{l}_2$, $a.\mathrm{l}_2$ converges while $b.\mathrm{l}_2$ diverges.

The encodings of A,B and C look as follows

$$\begin{aligned}
\mathcal{T}(A) &= \langle \mathrm{l}_1 \rangle \mathcal{T}(\mathsf{Bool}) \wedge \langle \mathrm{l}_2 \rangle \mathcal{T}(\mathsf{Bool}) \\
&= \langle \mathrm{l}_1 \rangle \mathsf{isBool} \wedge \langle \mathrm{l}_2 \rangle \mathsf{isBool} \\
\mathcal{T}(B) &= \langle \mathrm{l}_1 \rangle \mathcal{T}(\mathsf{Bool}) \\
&= \langle \mathrm{l}_1 \rangle \mathsf{isBool} \\
\mathcal{T}(C) &= \langle \mathrm{l}_2 \rangle \mathcal{T}(\mathsf{Bool}) \\
&= \langle \mathrm{l}_2 \rangle \mathsf{isBool}
\end{aligned}$$

If the type formulae should characterize Gordon-Rees bisimulation, then it should be the case that $a_B, b_B \in [\![\mathcal{T}(B)]\!]\sigma$ and $a_C, b_C \in [\![\mathcal{T}(C)]\!]\sigma$, but $a_A, b_A \notin [\![\mathcal{T}(A)]\!]\sigma$. But this does not hold, since $a_A, b_A \in [\![\mathcal{T}(A)]\!]\sigma$. □

8.4 Final comments

In this chapter we have described how certain properties of ς-calculus terms can be described within the modal μ-calculus. The work presented here first appeared in the Master thesis of Andersen and Petersen [AP96] and later appeared in [AHKP97]. Unfortunately both papers contained some errors, especially Theorem 8.8 and its proof was wrong. The major difference between the papers mentioned and this chapter is the introduction of satisfaction relations (Definition 8.4) and the use of them to prove soundness of the translation of types to logical formulae (Theorem 8.8). Another difference is that we in this chapter only use box-modalities in Example 8.1, whereas [AP96] and [AHKP97] include them also in the translation of types (which is unnecessary since the ς-calculus is deterministic).

8.4.1 Extensions

A natural question is: How much further can we go in this direction? We can easily deal with arrow types, if we introduce a simple notion of intuitionistic implication into our logic. Let us say that F implies G for some object a if a is an object abstraction which, whenever given an argument satisfying the property F becomes an object satisfying the property G.

$$[\![F \Rightarrow G]\!]\sigma = \{a_{A \to B} \mid \forall\, b_A \in [\![F]\!]\sigma : (ab)_B \in [\![G]\!]\sigma\}$$

We then get

$$\mathcal{T}(A \to B) = \mathcal{T}(A) \Rightarrow \mathcal{T}(B)$$

It is straightforward to see that this extended interpretation is sound w.r.t. the subtyping rules for arrow types, and in particular for the rule

$$(\textsc{Sub Arrow}) \quad \frac{\Gamma \vdash A' <: A \quad \Gamma \vdash B <: B'}{\Gamma \vdash A \to B <: A' \to B'}$$

which states that the arrow type is contravariant in its first argument and covariant in its second argument. (Observe that any recursion variable must occur in covariant position.)

This leads straight to the question of the possibility of dealing with the *variance annotations* suggested by Abadi and Cardelli in [AC96]. This could probably be done by changing the labelled transition system, such that a term a_A, where A is an object type containing variance annotations, has transitions labelled l_j for methods having co- and invariace annotations and transitions labelled $\mathrm{l}_j \Leftarrow \varsigma(x)b$ for methods with contra- and invariance annotations.

Chapter 9

Conclusions

The work presented in this thesis represents an attempt to answer the question:

> *Can one successfully adapt and use process calculus techniques to reason about object-oriented languages?*

As usual, it is difficult to give an affirmative answer to a question like this, as it would require a common agreement on what the answer is. So, the conclusion presented in the following only represents my own personal opinion, and is not necessarily the opinion of any of the coauthors of the papers behind this thesis.

Since the work presented here can be divided into two different approaches towards the initial question, the conclusion is likewise divided into two parts — with Section 9.1 discussing the translational approach used in Chapters 5 and 6, and Section 9.2 discussing adaptations of process calculi ideas directly to the ς-calculus found in Chapters 7 and 8. Finally in Section 9.3 we try to answer the initial question.

9.1 Use of process calculi

The major part of the work presented here has been on the use of the π-calculus as a language for giving semantics to object-oriented languages. As Chapter 5 shows, it can be a non-trivial task to get the semantics right. Most of the work on translations of the Functional ς-calculus into the π-calculus presented in Chapter 5 was on finding a translation that was correct, and on showing that the final one was indeed correct. But as we discussed in

Section 5.4 the translation we end up with, although correct, does not seem to support reasoning very well.

The translation of the Imperative ς-calculus presented in Chapter 6 looks more promising. Again a lot of work was required to get a correspondence between the translation in the π-calculus and the operational semantics for the ς-calculus. But when the correspondence was established we were able to show some non-trivial laws for the Imperative ς-calculus. The examples of proofs of properties for the Imperative ς-calculus indeed show the π-calculus as a promising tool for reasoning about properties for typed object-oriented languages. In particular, the fact that proofs can be finitary and that the π-calculus provides us with coinductive techniques incline us to believe that the π-calculus can be put to use here.

Whereas the techniques used in Chapter 5 to show operational adequacy were pretty standard, the proof of operational adequacy for the translation of the Imperative ς-calculus introduces some novel techniques which we believe are well-suited for reasoning about imperative languages in general. The idea in the factorization of the translation is to isolate the imperative parts of the translation, such that it is possible to use well-known techniques for functional names and processes. It also sheds some light on the exact difference between the Imperative and Functional ς-calculus. Our work introduces ready simulation to the π-calculus and puts it to new use, namely to prove the relation between the factorized and the original encoding.

9.1.1 Related work

An interpretation of the Imperative ς-calculus into a form of imperative polymorphic λ-calculus with subtyping, recursive types and records has been found by Abadi, Cardelli and Viswanathan [ACV96]. This interpretation has been used to validate the subtyping and typing judgments of the Imperative ς-calculus. However, it would be difficult to prove behavioral properties of the Imperative ς-calculus from this interpretation, since very little is known of the theory of the target imperative λ-calculus. The translation in [ACV96] also differs form the one presented here, in that the types are translated into a second order type system, whereas we translate first-order types to first-order types. Viswanathan [Vis97] has also developed a translation of the Imperative ς-calculus to a functional language with records and references where the translation of types is first-order to first-order. Interestingly enough, the type translation is *exactly* the same as the one we have exposed.

The only work on behavioral equivalences for the Imperative ς-calculus that we are aware of is Gordon, Hankin and Lassen's [GHL97]. In this work, however, the Imperative ς-calculus is *untyped*. Gordon, Hankin and

Lassen study contextual equivalence for this untyped Imperative ς-calculus, prove that it coincides with a variant of Mason and Talcott's CIU ("Closed Instances of Use") equivalence [MT91], and use the latter to validate some basic laws of the calculus. We briefly discuss the difference between their and our work. The two most notable differences are:

- Gordon, Hankin, and Lassen do not have a coinductive technique. They show that contextual equivalence for the Imperative ς-calculus coincides with the CIU equivalence of Mason and Talcott [MT91], which reduces the number of contexts that one has to consider. But, still, the definition of CIU equivalence uses universal quantification over an infinite number of contexts.

  Instead, our translations provide us with the proof techniques of the π-calculus. This gives us the possibility of using equational laws and coinductive techniques of the π-calculus to reason about the Imperative ς-calculus. As shown in Section 6.7, the bisimulations required to validate non-trivial laws can even be finite.

- Gordon, Hankin, and Lassen only study the untyped Imperative ς-calculus. This means that it is only possible to prove properties that do not rely on the type of objects. This can be a serious drawback, as a lot of interesting transformations for object-oriented languages rely on type information.

  Since our translation respects typing, we can use the π-calculus to reason about properties for the Imperative ς-calculus that rely on types (c.f. (Eq Sub Obj) in Section 6.7).

The closest related work on translations of object-oriented languages to the π-calculus is that of the typed translation of the Functional ς-calculus into the π-calculus [San98]. As discussed in Section 6.6, our translation of the Imperative ς-calculus can be seen as an extension of the translation of the Functional ς-calculus. The major differences are the very different proofs of operational correctness for the translation and our work on proving properties for the Imperative ς-calculus.

Related is also the work by Liu and Walker [LW99], Walker [Wal95, Wal91, Wal93] and Jones [Jon93] on translations of POOL-like class-based object-oriented languages into the π-calculus.

9.1.2 Further work

An interesting topic for further study is to continue the investigation of the difference between the Functional and Imperative ς-calculus in the common

framework that the translations provide us with. For instance, one can further investigate the examples of the difference between contextual equivalence for the Functional and Imperative ς-calculus.

The extensibility of the interpretation of imperative objects, and its resemblance to the interpretation of functional objects, suggests that the representation of objects into the π-calculus is a robust one, and that it could be used for giving semantics to, and proving properties of, a wide range of object-oriented languages, possibly combining imperative, functional and concurrent features.

An interesting and challenging question is to find a direct characterization of the equivalence induced on terms of the Imperative ς-calculus by the encoding into the π-calculus (where two terms are equated if their process translations are behaviorally equivalent), as has been done for the lazy λ-calculus [San94b, San99a].

At the time of writing, an investigation of Cardelli's experimental distributed programming language Obliq [Car95] using an extension of the translation of the Imperative ς-calculus is being carried out by Merro, Nestmann, Hüttel, Sangiorgi and the author [HKNS98, Nes99, HKMN99c, HKMN99a, HKMN99b, KMN00].

The results from this work are very promising. The π-calculus has been used to give semantics to a subset of Obliq, which so far has only had an informal semantics (apart from unpublished work by Talcott [Tal96]). The π-calculus semantics for the subset (named Øjeblik[1]) has been used in an attempt to show that object migration in Obliq as proposed by Cardelli is transparent. While attempting to do this, we discovered that object migration could not be transparent, and we were (based on the failure of the proof) able to give simple examples showing this [HKMN99a]. The failure led us to propose a modified semantics for Obliq for which we have been able to show transparency of object migration [KMN00].

Another extension of the Imperative ς-calculus for which no use of the π-calculus has been attempted is the Concurrent ς-calculus of Gordon and Hankin [GH98]. Their calculus is a simple concurrent extension of the Imperative ς-calculus, and it should be pretty straightforward to extend the translation from Chapter 6 to cover this case.

The ultimate test of the translational approach would be to deal with programming languages that are used commercially. Most of those, like JAVA and C++, are class-based imperative languages, and one way to pursue this line of work would be to work on a stripped-down version of such a language.

[1] A Danish word with a phonetic resemblance to Obliq.

9.2 Adaptation of process calculus techniques

The work on adaptations of process calculi directly to the semantics of the ς-calculus is, by comparison to the use of the π-calculus, of a more foundational nature.

In Chapter 7, we have shown that the denotational model proposed by Abadi and Cardelli [AC96] is correct, but not fully abstract with respect to the operational semantics, and in Chapter 8, we have shown a correspondence between the type system $\mathbf{Ob}_{1<:\mu}$ for the ς-calculus and the modal μ-calculus that captures both type assignment and subtyping. These results are primarily of theoretical interest.

However in both cases, if we did not have a correspondence between the operational and denotational semantics, or between types and modal logics respectively, it would certainly be an indication that something was not the way it ought to be.

The only real attempt to use the modal logic of Chapter 8 is Example 8.1, where the logic is used to describe the behavior of a calculator.

9.2.1 Related work

To our knowledge the only work on the relation between the operational and denotational semantics for the ς-calculus is the work presented in this thesis. Related is work on relations between operational and denotational semantics for λ-calculi (such as [AO93]) and denotational models generated directly from a structural operational semantics (such as [AI96]).

Other studies of logics for the ς-calculus has been carried out in the setting of the Imperative ς-calculus (c.f. Section 6.1). In [AL97], Abadi and Leino study a Hoare style logic for a restricted version of the Imperative ς-calculus where only fields (that is methods that do not use self) can be updated and with a first order type system without recursion. In their work the logic is proven sound with respect to the operational semantics and a translation of types to logical formulae is provided. In [Lei97], Leino extends the previous work to handle a kind of recursive types. In [NN98], Nielson and Nielson study a flow logic for an untyped version of the Imperative ς-calculus, proving correctness of the logic in a way similar to the one we use in the proof of Theorem 8.8. In [Boe99] de Boer present a Hoare style logic for a sequential version of POOL.

Rasmussen and Hüttel [Ras99, HR99] have developed a labelled transition semantics for the untyped Imperative ς-calculus and showed, similarly to what Gordon and Rees have done for the typed Functional ς-calculus, that bisimulation equivalence defined on their labelled transition system coincides

with contextual equivalence.

9.2.2 Further work

Connection between operational and denotational semantics Here, the work presented is just a first step, and much remains to be done.

It is no surprise that the equational theory is sound, but incomplete in the untyped case. As we can express all computable functions within the ς-calculus, we can express the complement of the halting problem for any given object a by the equation $a \leftrightarrow \Omega$ where Ω is the divergent object. The set of such equations is clearly not recursively enumerable. However, the set of provable equalities is a recursively enumerable set, so if the model can adequately capture simple nontermination properties, some equalities will not be provable. However, one would like a systematic approach that will shed more light on the model under consideration.

A topic of further research is to show the incompleteness of certain equational theories by establishing a stronger result on soundness, namely that Abadi and Cardelli's equational theory is 'sound in all models'. In order to achieve such a result, we need to make precise the notion of an object model along the lines of the familiar notion of a model for the λ-calculus [Bar84]. In particular, we would need an interpretation of types.

As an important by-product, the notion of a model of the ς-calculus would let us compare various interpretations already in existence. Ideally, the translation of the untyped ς-calculus into the π-calculus from Chapter 5 should provide us with another example of a ς-model, just as Sangiorgi [San95] has shown that a translation of the λ-calculus into the π-calculus gives rise to a λ-model. Whether this is indeed the case, is a topic for future investigation. A starting point for this effort might be the fully abstract model for $\mathbf{Ob}_{1<:\mu}$ of Viswanathan [Vis98].

Modal logic For the use of modal logics, a natural next step is to investigate how one can use the μ-calculus to verify interesting properties of objects. The notion of model-checking, that is, algorithmically checking whether a term satisfies a given modal formula [SW89], is already well-understood in the context of process calculi. It remains to be seen how far we can proceed within the ς-calculus. At the moment this line of work looks quite hard, as the labelled transition system is both infinite state and infinitely branching. One could of course remove the problem of infinite branching by restricting method override to field override, and develop a symbolic semantics, as has been done for value-passing calculi [HL95] and the π-calculus [BD96]. As for

the problem of the transition system being infinite state, one should try to find a reasonably large subset whose symbolic semantics is finite.

A related question is whether one can capture the polymorphic type systems of the ς-calculus proposed in [AC96] within a higher-order modal predicate logic.

One might also want to investigate the relation between the logic and the denotational model of Abadi and Cardelli. A first guess might be that each type formula F gives rise to a PER $\mathcal{R}_F$ over **Obj**

$$\mathcal{R}_F = \{(a, b) \mid a \models F,\ b \models F\}$$

Finally, the interpretation of types as modal formulae suggests an alternative account of the semantics of types to that presented in [AC96]. One might consider determining whether the translation of $\mathbf{Ob}_{1<:\mu}$ types into the modal μ-calculus together with a suitably quotiented term model gives rise to a typed ς-model.

9.3 General conclusions

It is my strong opinion, that *process calculus techniques can be used successfully to support reasoning about object-oriented languages*. Having said this, I must also say that it is neither an easy task to do, nor something I would claim every programmer/developer should learn to do. In my opinion, one has to distinguish between showing properties of the programming language itself and showing properties of programs written in the language.

Properties of programming languages As programming languages become more and more complex, the interplay between different language constructs also becomes complex. Here is a thorough, perhaps even formal, analysis of certain features of the programming language advantageous (although it is often not performed). Such an analysis should be used to show that the language obeys certain properties and to help in developing equational theories that can be used by programmers to reason about programs written in the language. Most (if not all) of the results about the ς-calculus realized by using process calculus techniques in this thesis have been of the aforementioned kind.

Properties of programs In most cases, the programmers intuitive understanding of the semantics of the programming languages suffices in order to reason about the programs she/he writes. Exceptions of course exist, such as

mission-critical systems, complex datastructures and so on. In these cases, formal reasoning is be needed to ensure the correctness of programs.

In the present thesis, the only case that can be considered to have a vague resemblance with reasoning about programs is Example 8.1, but it is not really something that gives any insight, as to which one can use process calculus techniques to reason about programs.

Fortunately, others have used process calculus techniques to reason about programs. Sangiorgi [San99c] uses the π-calculus to prove a program transformation of a sequential object-oriented program into a concurrent version correct. Philippou and Walker [PW97] prove the correctness of concurrent operations on B-trees using a π-calculus. What these examples show is that it is indeed possible to use translations of programs into the π-calculus to support reasoning.

However the work mentioned has been performed by researchers who are specialists in using the π-calculus. I do not believe that these techniques can come to widespread use, unless people are trained in reasoning using the π-calculus and some kind of tool support (like theorem provers and model checkers for the π-calculus) is provided.

The reason why other peoples work on adaptations of process calculus techniques directly to the semantics of object-oriented languages is not mentioned, is not because it does not exist, but because of my ignorance of their work.

Since the answer to our initiating question is positive, one final question arises: *Should one prefer the translational or the adaptive approach?* By looking at the work presented here and the work done by others, it looks as if the most promising path is to use the translational approach. The advantage of using the translational approach is that the "only" thing one needs to do before starting using the translation is to come up with a suitable target language, a translation and establish a correspondence between the original semantics (if one exists) of the language in question and the translation. Although this can be tedious and error-prone, it is a well-defined task. In order to use an adaptation of process calculus techniques one needs to develop a labelled transition system, equivalences (one is usually not enough), to show the correctness of the labelled transition system, to develop laws using the equivalences, and so on. And this needs to be done for every language. Another advantage of using a translation into a common calculus is that it allows one to compare and contrast properties of different languages on a common platform, just as we did when comparing the Imperative and Functional ς-calculus in Section 6.6.

Based on this, but also because I might be biased towards translations

which I happen to like working with, the approach I would recommend if someone asked me is therefore to use a translation.

Appendix A

Theoretical developments on the typed π-calculus

A.1 Ready simulation

In this section we develop some theory for ready simulation. To ease the reading we omit type annotations since ready simulation is an untyped relation.

Definition A.1 (Ready Simulation (also Definition 6.16)) *A relation $\mathcal{R}$ is a (weak)* ready simulation *if $P\ \mathcal{R}\ Q$ implies, for any μ (with bound names of μ not free in P, Q):*

i. If $P \xrightarrow{\mu} P'$, then there exists a Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$.

ii. If $Q \xrightarrow{\mu} Q'$, then there exist a P' s.t. $P \stackrel{\mu}{\Longrightarrow} P'$

We say that Q ready simulates *P, written $P \prec Q$, if $P\ \mathcal{R}\ Q$ for some ready simulation $\mathcal{R}$.*

Theorem A.2 *Ready simulation is a precongruence with respect to parallel composition and restriction.*

Proof.

Let $\mathcal{R}$ be a ready simulation and let

$$\mathcal{S} = \left\{ \Big((\nu\tilde{x})(P|R), (\nu\tilde{x})(Q|R) \Big) \;\middle|\; \forall (P,Q) \in \mathcal{R},\ \tilde{x} \in \mathcal{P}^{\mathbf{Names}},\ R \in \mathbf{Proc} \right\}$$

We now claim that $\mathcal{S}$ is a ready simulation.

If $(\nu\tilde{x})(P|R) \xrightarrow{\mu} S$ this must be inferred using either (Res) or (Open) as the last rule.

Let us first consider the case where (Res) is the rule used, that is $P|R \xrightarrow{\mu} S'$ with $S = (\nu\tilde{x})S'$. This transition can be inferred using the following rules:

(Comp-R): $R \xrightarrow{\mu} R'$ and $S' = P|R'$ which $Q|R$ can match by doing the same, and clearly $(\nu\tilde{x})(P|R') \mathcal{S} (\nu\tilde{x})(Q|R')$.

(Comp-L): $P \xrightarrow{\mu} P'$ and $S' = P'|R$. Since $P \mathcal{R} Q$ there exist a Q' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ with $P' \mathcal{R} Q'$. Now using (Comp-L) we have $Q|R \stackrel{\mu}{\Longrightarrow} Q'|R$ and clearly by definition of $\mathcal{S}$ we have $(\nu\tilde{x})(P'|R) \mathcal{S} (\nu\tilde{x})(Q'|R)$.

(Sync-L): We must have $\mu = \tau$, $P \xrightarrow{(\nu\tilde{n})\overline{y}v} P'$, $R \xrightarrow{yv} R'$ and $S' = (\nu\tilde{n})(P'|R')$. Since $P \mathcal{R} Q$ there exist a Q' such that $Q \stackrel{(\nu\tilde{n})\overline{y}v}{\Longrightarrow} Q'$ with $P' \mathcal{R} Q'$. By (Sync-L) we infer $Q|R \stackrel{\tau}{\Longrightarrow} (\nu\tilde{n})(Q'|R')$ and again $(\nu\tilde{x},\tilde{n})(P'|R') \mathcal{S} (\nu\tilde{x},\tilde{n})(Q'|R')$.

(Sync-R): Same argument as in the previous case.

The other possibility is that (Open) is used as the last rule in the inference of $(\nu\tilde{x})(P|R) \xrightarrow{\mu} S$, then $\mu = (\nu\tilde{n})\overline{y}v$ with $y \notin \tilde{x}$, and $P|Q \xrightarrow{(\nu\tilde{n})\overline{y}v} S'$ with $S = (\nu\tilde{x}')S'$ for some $\tilde{x}' \subseteq \tilde{x}$. The transition $P|Q \xrightarrow{(\nu\tilde{n})\overline{y}v} S'$ can be inferred using the following rules:

(Comp-L): We have $P \xrightarrow{(\nu\tilde{n})\overline{y}v} P'$ and $S' = P'|S$. Because $P \mathcal{R} Q$ then $Q \stackrel{(\nu\tilde{n})\overline{y}v}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$ for some Q'. By (Comp-L) and (Open) we infer $(\nu\tilde{x})(Q|R) \stackrel{(\nu\tilde{n})\overline{y}v}{\Longrightarrow} (\nu\tilde{x}')(Q'|R)$ and by construction of $\mathcal{S}$ we have $(\nu\tilde{x}')(P'|R) \mathcal{S} (\nu\tilde{x}')(Q'|R)$.

(Comp-R): We have $R \xrightarrow{(\nu\tilde{n})\overline{y}v} R'$ and $S' = P|R'$. Obviously $Q|R \xrightarrow{(\nu\tilde{n})\overline{y}v} Q|R'$ and $(\nu\tilde{x}')(P|R') \mathcal{S} (\nu\tilde{x}')(Q|R')$.

The other way around; if $(\nu\tilde{x})(Q|R) \xrightarrow{\mu}$ then also $(\nu\tilde{x})(Q|R) \xrightarrow{\mu}$ is handled in a similar manner. □

Definition A.3 (Ready Simulation up to Context) *A relation $\mathcal{R}$ is a* ready simulation up to context *if $P \mathcal{R} Q$ implies:*

i. If $P \xrightarrow{\mu} P'$; then there exist processes $Q', \tilde{P}, \tilde{Q}, R$, names $\tilde{x}$, such that $Q \stackrel{\mu}{\Longrightarrow} Q'$, $P' \sim (\nu\tilde{x})(R|\tilde{P})$, $Q' \sim (\nu\tilde{x})(R|\tilde{Q})$ and $\tilde{P} \mathcal{R} \tilde{Q}$.

ii. If $Q \xrightarrow{\mu}$*; then there exist a* P' *s.t.* $P \stackrel{\mu}{\Longrightarrow}$

Theorem A.4 *If* $P \mathcal{R} Q$ *for some ready simulation up to context then* $P \prec Q$*.*

Proof. Let $\mathcal{R}$ be a ready simulation up to context and let

$$\mathcal{S}= \left\{ \Big((\nu\tilde{x})(R|\tilde{P}), (\nu\tilde{x})(R|\tilde{Q}) \Big) \;\middle|\; \forall R \in \mathbf{Proc},\ \tilde{x} \in \mathcal{P}^{\mathbf{Names}},\ \tilde{P}\ \mathcal{R}\ \tilde{Q} \right\}$$

We now claim that $\sim \mathcal{S} \sim$ is a ready simulation.

The proof of this goes by a lengthy analysis of the possible transitions. We shall only consider one of the cases, as the rest follows by similar reasoning.

Let $P^* \sim (\nu\tilde{x})(R|\tilde{P})\ \mathcal{S}\ (\nu\tilde{x})(R|\tilde{Q}) \sim Q^*$. If $P^* \xrightarrow{\mu} P^{**}$ then there exists a S s.t. $(\nu\tilde{x})(R|\tilde{P}) \xrightarrow{\mu} S$ and $P^{**} \sim S$.

If $(\nu\tilde{x})(R|\tilde{P}) \xrightarrow{\mu} S$ it must be inferred using either (Res) or (Open) as the last rule used. If is was (Res) that was the last rule used, then we must have $R|\tilde{P} \xrightarrow{\mu} S'$ with $S = (\nu\tilde{x})S'$. This transition must be the result of, either a transition by one of the components, or by an interaction between two of the components.

We consider the case when the transition is caused by an interaction between two components of $\tilde{P}$ and inferred using (Sync-L) as the last rule. That is, we have i, j s.t. $i \neq j$ and $P_i \xrightarrow{(\nu\tilde{n})\overline{y}v} P_i'$ and $P_j \xrightarrow{yv} P_j'$ and

$$S = (\nu\tilde{x})(R|P_1|\ldots(\nu\tilde{n})(P_i'\ldots P_j')\ldots P_n)$$

Since $\tilde{P}\ \mathcal{R}\ \tilde{Q}$, there exist processes $R_i, R_j, \tilde{Q}_i, \tilde{Q}_j, \tilde{P}_i, \tilde{P}_j$ and names $\tilde{y}_i, \tilde{y}_j$ such that $Q_i \stackrel{(\nu\tilde{n})\overline{y}v}{\Longrightarrow} Q_i'$ and $Q_j \stackrel{yv}{\Longrightarrow} Q_j'$ with

$$P_k' \sim (\nu\tilde{y}_k)(R_k|\tilde{P}_k)\ \mathcal{S}\ (\nu\tilde{y}_k)(R_k|\tilde{Q}_k) \sim Q_k' \text{ for } k \in \{i, j\}$$

Let $T = (\nu\tilde{x})(R|Q_1|\ldots(\nu\tilde{n})(Q_i'\ldots Q_j')\ldots Q_n)$ Using (Comp-L), (Comp-R) and (Sync-L) and (Res) we can infer $(\nu\tilde{x})(R|\tilde{Q}) \stackrel{\tau}{\Longrightarrow} T$ and then also that $Q^* \stackrel{\tau}{\Longrightarrow} Q^{**}$ with $T \sim Q^{**}$.

Now $P^{**} \sim S \sim ((\nu\tilde{x}, \tilde{y}_i, \tilde{y}_j, \tilde{n})(R|R_i|R_j|P_1|\ldots\tilde{P}_i'\ldots\tilde{P}_j'\ldots P_n)$ and $Q^{**} \sim T \sim (\nu\tilde{x}, \tilde{y}_i, \tilde{y}_j, \tilde{n})(R|R_i|R_j|Q_1|\ldots\tilde{Q}_i'\ldots\tilde{Q}_j'\ldots Q_n)$, so we have $P^{**} \sim \mathcal{S} \sim Q^{**}$.

The second clause in the definition of ready simulation is handled in the same manner, it is just simpler since we do not have to relate the result of the transitions. □

A.2 Typed bisimulation

In [BS98] Boreale and Sangiorgi present a labelled typed bisimulation for the monadic π-calculus, and show that their definition of typed bisimulation coincides with typed barbed congruence. In this section, we present a simplified form of Boreale and Sangiorgi's typed bisimulation. The simplification makes the definition of typed bisimulation simpler, but we loose the characterization of typed barbed congruence. Instead, we only achieve that type bisimulation implies type barbed congruence. The proofs are omitted, as most of them are quite tedious and just extend Boreale and Sangiorgi's results to the π-calculus of Section 6.2.

The difference between typed and untyped labelled bisimulation is, that in a typed bisimulation we restrict the actions that we need to match in order for two processes to be bisimilar. The restrictions are based on the knowledge about the type of name that the environment knows. For instance, assume that $a : T^{\mathsf{w}^{\mathsf{w}}}$ is an assumption in Γ and that $\Gamma \vdash P,\ Q$. If we want to show that $P \sim_b^\Gamma Q$ directly, we have to consider how P and Q behave in all contexts that has a hole respecting Γ. One such context is $C[\cdot] = a(x).\overline{x}v \mid [\cdot]$ having a type environment Δ such that $\Delta \vdash v{:}T,\ a{:}T^{\mathsf{w}^{\mathsf{b}}}$. What this context does, is that it recieves some name on a together with the write capability, and then transmit a value on that name. P and Q may (if they have the read capability for the name transmitted on a) recieve v.

If we were to show this in a bisimulation-like manner, it would have to be something like: If $P \xrightarrow{\mu} P'$ with a as subject, then if μ is an output action Q must match this by doing a similar action $Q \xrightarrow{\mu} Q'$. If μ was an input action on a we do not have to consider it, because the environment is not able to observe such actions. The name we recieve on a, say b, is a name that the environment can use for output, so this implies that we must now match an input action on b by P', with a similar input actions by Q'. This illustrates that a typed bisimulation should consist of triples (Γ, P, Q), where Γ contain information about the environments knowledge of names.

Since values are structured, a transition can contain more than one name. Below, typeof is a partial function that maps a pair (v, S) of a π-calculus value and a type to a set of pairs $\{(x_i, S_i)\}_{i\in I}$ of names and types. Intuitively, typeof extracts the names contained in v together with their type, as derived from S. That is, $\mathsf{typeof}(v, S) = \{(x_i, S_i)\}_{i\in I}$ implies $\{(x_i, S_i)\}_{i\in I} \vdash v : S$.

Definition A.5 $\mathsf{typeof}(v, S)$ *is a function that associate the names in* v *with the type they are given in the type* S *defined by:*

$$\mathsf{typeof}(x, S) \triangleq \{(x, S)\}$$

$$\begin{array}{rcl} \mathsf{typeof}(\langle v_1 \ldots v_n \rangle, \langle T_1 \ldots T_n \rangle) & \triangleq & \cup_{i=1}^{n} \mathsf{typeof}(v_i, T_i) \\ \mathsf{typeof}(\mathrm{l}_i\_v, [\,\mathrm{l}_1\_T_1..\mathrm{l}_n\_T_n\,]) & \triangleq & \mathsf{typeof}(v, T_i) \text{ if } i \in 1..n \end{array}$$

We use Λ to range over *multiset type environments*. These are defined like type environments, but a name may appear in Λ more than once. We define $\Lambda \vdash p : T$ if there a subset Γ of Λ s.t. Γ is a type environment and $\Gamma \vdash p : T$.

To reflect update of the environments knowledge, we define an operational semantics for multiset type environments:

Definition A.6

i. $\Lambda \xrightarrow{\tau} \Lambda$

ii. $$\frac{\Lambda \vdash p{:}S^{\mathsf{r}} \quad \tilde{n} \cap \mathsf{n}(\Lambda) = \emptyset}{\Lambda \xrightarrow{(\nu \tilde{n}:)\overline{p}v} \Lambda, \mathsf{typeof}(v, S)}$$

iii. $$\frac{\Lambda \vdash p{:}S^{\mathsf{w}} \quad \tilde{n} \cap \mathsf{n}(\Lambda) = \emptyset \quad \tilde{n} \subseteq \mathsf{n}(v) \quad \Lambda, \tilde{n}{:}\tilde{T} \vdash v{:}S}{\Lambda \xrightarrow{pv} \Lambda, \tilde{n}{:}\tilde{T}}$$

If μ is an output action $(\nu \tilde{n}{:}\tilde{T})\overline{p}v$, then $|\mu|$ is $(\nu \tilde{n}{:})\overline{p}v$; if μ is an input or a tau action, then $|\mu| = \mu$.

A typed relation in a set of pairs of the form (Λ, P, Q).

Definition A.7 (Typed bisimulation) *A symmetric typed relation $\mathcal{R}$ is a* typed bisimulation *if $(\Lambda, P, Q) \in \mathcal{R}$ implies:*

i. if $P \xrightarrow{\mu} P'$, then there are Λ' and Q' such that

- $\Lambda \xrightarrow{|\mu|} \Lambda'$,
- $Q \xrightarrow{\mu'} Q'$ *with* $|\mu'| = |\mu|$,
- *and* $(\Lambda', P', Q') \in \mathcal{R}$.

If $(\Lambda, P, Q) \in \mathcal{R}$ for some typed bisimulation, we say that P and Q are type bisimilar under Λ, written $P \simeq_{\Lambda} Q$.

In the above definition, we may replace (Λ', P', Q') with (Λ'', P', Q'), for $\Lambda'' = \Lambda' \cap \mathsf{fn}(P', Q')$.

Example A.1 As an example of the advantage of typed bisimulation consider the following two agents:

$$P = \overline{a}v \quad Q = \overline{a}v \mid b(x)$$

Obviously P and Q are not bisimilar, but in the type environment $\Gamma = a^{\mathsf{w}}$ they are type bisimilar because we cannot observe the input on b. □

Definition A.8 *A substitution* σ respects a type environment Δ *if* $\mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\Delta)$ *and for all* $x \in \mathsf{dom}(\sigma)$ *there exists a type* T *s.t. both* $\Delta \vdash x : T$ *and* $\Delta \vdash \sigma(x) : T$ *hold.*

The typed bisimulation is preserved by all contexts respecting the type assumptions. That is:

Theorem A.9 *Suppose that* Γ *and* Δ *agree on common names (that is, for all* x *s.t.* $x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)$*, it holds that* $\Gamma(x) = \Delta(x)$*.*

If for all extensions Δ' *of* Δ *and substitution* σ *that respect* Δ'*, it holds that* $P\sigma \simeq_{\Delta'} Q\sigma$*, then for all* Γ/Δ*-context* C *it holds that* $C[P] \simeq_{\Gamma} C[Q]$*.*

In particular, we have:

Corollary A.10 *If* $P \simeq_{\Delta} Q$*, and* $\Delta \vdash R$*, then* $P|R \simeq_{\Delta} Q|R$*.*

Stating the relation between typed bisimulation and typed barbed congruence has minor complications. Firstly, if two processes P and Q are typed barbed congruent under the assumptions Δ, then $\Delta \vdash P, Q$, implies that Δ contains assumptions about *all* free names in P and Q. We do not have such a requirement for the type assumptions in the environment used in typed bisimulation — the environment does not need to contain all free names (as in Example A.1) and in can contain assumptions that are not consistent with the use of names in P and Q. Furthermore, the type assumptions in the environment represent the assumptions under which a (virtual) context should be typeable, not the processes being compared. This means that the assumptions are "complementary" to the ones of the processes. For example, if $a{:}T^{\mathsf{r}}$ is an assumption in the environment, it means that the environment is willing to receive a value transmitted on the name a; this, of course, requires the collaboration by the process, that must be willing to transmit a value (that is the process must have the write-capability on a, and therefore by typable under the assumption $a{:}T^{\mathsf{w}}$).

Theorem A.11 *Let* $\Delta\updownarrow$ *denote the type environment* Δ *with the outermost tags reversed (*$T^{\mathsf{w}}\updownarrow = T^{\mathsf{r}}$*,* $T^{\mathsf{r}}\updownarrow = T^{\mathsf{w}}$ *and* $T^{\mathsf{b}}\updownarrow = T^{\mathsf{b}}$*). If for all substitutions* σ *such that* $\Delta\sigma \vdash P\sigma, Q\sigma$ *and* $P\sigma \simeq_{\Delta\sigma\updownarrow} Q\sigma$*, then* $P \sim_b^{\Delta} Q$*.*

The proof techniques of the ordinary untyped bisimulations, can be adapted to the typed bisimulation. Here, we show a version of the up-to-parallel-composition technique.

If $\mathcal{R}$ is a typed relation, then we write $(\Lambda, P, Q) \in \widetilde{\mathcal{R}}$ if if the following conditions hold:

i. for some n, $P = \prod_{i \in 1..n} P_i$ and $Q = \prod_{i \in 1..n} Q_i$;

ii. for all i: either $P_i = Q_i$ or $(\Lambda, P_i, Q_i) \in \mathcal{R}$

iii. there is no name x s.t. x is free in both P_h and P_k ($h \neq k$, $1 \leq h, k \leq n$) and x is free in one of the two processes in input subject position;

iv. all P_i's are well typed under a type environment where the r tag does not occur.

v. the same two conditions above for the Q_i's.

Definition A.12 (Typed bisimulation up-to parallel composition) *A symmetric typed relation $\mathcal{R}$ is a* typed bisimulation up-to parallel composition *if $(\Lambda, P, Q) \in \mathcal{R}$ implies:*

i. if $P \xrightarrow{\mu} P'$, then there are Λ' and Q' s.t.

- $\Lambda \xrightarrow{|\mu|} \Lambda'$,
- $Q \xrightarrow{\mu'} Q'$ *with* $|\mu'| = |\mu|$,
- *there are* $P'' \sim P'$ *and* $Q'' \sim Q'$ *s.t.* $(\Lambda', P'', Q'') \in \widetilde{\mathcal{R}}$.

Theorem A.13 *If $\mathcal{R}$ is a typed bisimulation up-to parallel composition and $(\Lambda, P, Q) \in \mathcal{R}$ then $P \simeq_\Lambda Q$.*

Another useful property which we shall need, is that under special circumstances we can add restrictions with different type annotations and still keep typed bisimulation between terms.

Lemma A.14 *If*

i. $P \simeq_\Gamma Q$ and $a \notin \mathsf{dom}(\Gamma)$;

ii. a is only used in subject position;

iii. $(\nu a{:}S)P$ and $(\nu a{:}S')Q$ are well-typed.

then $(\nu a{:}S)P \simeq_\Gamma (\nu a{:}S')Q$.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[ACS98] Roberto M. Amadio, Ilaria Castellani and Davide Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[ACV96] Martín Abadi, Luca Cardelli and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In POPL'96 [POP96], pages 396–409.

[AHIK97] Luca Aceto, Hans Hüttel, Anna Ingólfsdóttir and Josva Kleist. Relating Semantic Models for the Object Calculus. In *Proceedings of Express 97 Workshop*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1997.

[AHKP97] Dan S. Andersen, Hans Hüttel, Josva Kleist and Lars H. Pedersen. Objects, Types and Modal Logics. In *Proceedings of 4th Workshop on Foundations of Object Oriented Languages - FOOL4*, 1997.

[AI96] Luca Aceto and Anna Ingólfsdóttir. CPO Models for Compact GSOS Languages. *Information and Computation*, 129(2):107–141, September 1996.

[AL97] Martín Abadi and K. Rustan M. Leino. A Logic of Object-Oriented Programs. In Michel Bidoit and Max Dauchet, editors, *Proceedings of TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696, Lille, France, April 1997. Springer-Verlag.

[AO93] Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Information and Computation*, 105(2):159–267, August 1993.

[AP96] Dan S. Andersen and Lars H. Pedersen. An Operational Approach to the ς-Calculus. Master's thesis, Aalborg University, September 1996. Available as technical report R-96-2034.

[Apo82] Tom M. Apostol. *Mathematical Analysis*. Addison-Wesley, 1982.

[Bar84] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[BD96] Michele Boreale and Rocco De Nicola. A Symbolic Semantics for the π-Calculus. *Information and Computation*, 126(1):34–52, 1996. Available as Report SI 94 RR 04, Università "La Sapienza" di Roma; an extended abstract appeared in *Proceedings of CONCUR '94*, pages 299–314, LNCS 836.

[Bes93] Eike Best, editor. *CONCUR '93: Fourth International Conference on Concurrency Theory (Hildesheim, Germany)*, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BIM95] Bard Bloom, Sorin Istrail and Albert Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995. Earlier version appeared in POPL '88: 229–239.

[Boe99] Frank de Boer. A WP-calculus for OO. In *Proceedings of FOSSACS'99*, 1999.

[Bor98] Michele Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2):205–226, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 163–178.

[BS98] Michele Boreale and Davide Sangiorgi. Bisimulation in name-passing calculi without matching. In LICS'98 [LIC98].

[Car95] Luca Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Proceedings of POPL '95*. A preliminary version appeared as Report 122, Digital Systems Research, June 1994.

[CDM92] Pierre Cointe, Christophe Dony and Jacques Malenfant. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of OOPSLA'92*, 1992.

[CHC90] William R. Cook, Walter L. Hill and Peter S. Canning. Inheritance is not Subtyping. In *Seventeenth Annual Symposium on Principles of Programming Languages (POPL) (San Fancisco, CA)*, pages 125–135. ACM, January 1990. Also in the collection [GM94].

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection Types and Multiple Inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title *Multiple Inheritance via Intersection Types*.

[Eme94] E.A. Emerson. *Handbook of Theoretical Computer Science*, chapter Temporal and Modal Logic, pages 995–1072. Elsevier, 1994.

[FF86] Martin Felleisen and Daniel P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[FG96] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In POPL'96 [POP96], pages 372–385.

[Fis96] Kathleen Fisher. *Type Systems for Object-Oriented Languages*. PhD thesis, Stanford University, August 1996.

[GH98] Andrew D. Gordon and Paul D. Hankin. A Concurrent Object Calculus: Reduction and Typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[GHL97] Andrew D. Gordon, Paul D. Hankin and S. B. Lassen. Compilation and Equivalence of Imperative Objects. In S. Ramesh

and G. Sivakumar, editors, *Proceedings of FSTTCS '97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, December 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.

[GJS96] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. The JAVA Series. Addison-Wesley, 1996.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics, and Language Design*. Foundations of Computing Series. MIT Press, 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implemementation*. Addison-Wesley, Reading, MA, 1983.

[GR96] Andrew D. Gordon and Gareth D. Reeth. Bisimilarity for a First-Order Calculus of Objects with Subtyping. In POPL'96 [POP96], pages 386–395. Full version available as Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

[HHK95] Martin Hansen, Hans Hüttel and Josva Kleist. Bisimulations for asynchronous mobile processes. In Insup Lee and Scott A. Smolka, editors, *Proceedings of CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[HK96] Hans Hüttel and Josva Kleist. Objects as Mobile Processes. Research Series RS-96-38, BRICS, October 1996. Presented at MFPS '96.

[HKMN99a] Hans Hüttel, Josva Kleist, Massimo Merro and Uwe Nestmann. Aliasing Models for Object Migration. In *Proceedings of EURO-PAR'99*, volume 1685 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. Distinguished Paper.

[HKMN99b] Hans Hüttel, Josva Kleist, Massimo Merro and Uwe Nestmann. Aliasing Models for Mobile Objects. To appear in Journal of Information & Computation, 1999.

[HKMN99c] Hans Hüttel, Josva Kleist, Massimo Merro and Uwe Nestmann. Migration = Cloning ; Aliasing (preliminary version). In *Informal Proceedings of FOOL 6, January 23, 1999*, 1999. Available in the BRICS Report Series.

[HKNS98] Hans Hüttel, Josva Kleist, Uwe Nestmann and Davide Sangiorgi. Surrogates in Øjeblik: Towards Migration in Obliq. In Hans Hüttel and Uwe Nestmann, editors, *Proceedings of SOAP '98*, volume NS-98-5 of *BRICS Notes Series*, pages 43–50. BRICS - Basic Research in Computer Science, Denmark, 1998.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):315–352, February 1995.

[HM85] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[Hon96] Kohei Honda. Composing Processes. In POPL'96 [POP96], pages 344–357.

[HP94] Martin Hofmann and Benjamin Pierce. A Unifying Type-Theoretic Framework for Objects. In P. Enjalbert, E. W. Mayr and K. W. Wagner, editors, *Proceedings of STACS '94*, volume 775 of *Lecture Notes in Computer Science*, pages 251–262. Springer-Verlag, 1994.

[HR99] Hans Hüttel and René Rasmussen. Bisimulation in the untyped imperative object calculus. Available from `http://www.cs.auc.dk/~hans`, 1999.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In P[ièerre] America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[IS94] Anna Ingólfsdóttir and Bernhard Steffen. Characteristic Formulae for Processes with Divergence. *Information and Computation*, 110:149–163, April 1994.

[Jac83] Michael Jackson. *System Development*. Prentice Hall, 1983.

[Jon93] Cliff Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In Best [Bes93], pages 158–172.

[KG89] Sonya Keene and Dan Gerson. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, Massachusetts, 1989.

[KMN00] Josva Kleist, Massimo Merro and Uwe Nestmann. Local π-Calculus at Work: Mobile Objects as Mobile Processes. To appear in Proceedings of TCS'2000, 2000.

[Koz83] Dexter Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce and David N. Turner. Linearity and the Pi-Calculus. In POPL'96 [POP96], pages 358–371.

[KS98] Josva Kleist and Davide Sangiorgi. Imperative Objects and Mobile Processes. In David Gries and Willem-Paul de Roever, editors, *Proceedings of PROCOMET '98*, pages 285–303. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

[Lar90] Kim G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theoretical Computer Science*, 72:265–288, 1990.

[Lei97] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. SRC Technical Note 1997-025, Digital SRC, October 1997.

[LIC98] IEEE. *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. Computer Society Press, July 1998.

[LMMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, 1993.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1–28, September 1991.

[LW99] Xinxin Liu and David Walker. Concurrent Objects as Mobile Processes. In Plotkin et al. [PST99]. To appear.

[Mey88] A.R. Meyer. Semantical Paradigms: Notes for an Invited Lecture. In *Proceedings* 3^{th} *Annual Symposium on Logic in Computer Science,* Edinburgh, pages 236–242. IEEE Computer Society Press, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mic] Microsoft. The DCOM webpage. `http://www.microsoft.com/com/default.htm`.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil92] Robin Milner. Functions as Processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[Mil93] Robin Milner. The Polyadic π-Calculus: A Tutorial. In Friedrich L. Bauer, Wilfried Brauer and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F: Computer and System Sciences*. NATO Advanced Study Institute, Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[MMS94] Tom Mens, Kim Mens and Patrick Steyart. OPUS: A Formal Approach to Object-Orientation. Techreport vub-tinf-tr-94-02, Vrije Universiteit Brussel, 1994.

[Mor68] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, December 1968.

[MPW92] Robin Milner, Joachim Parrow and David Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, September 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed Bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[MS98] Massimo Merro and Davide Sangiorgi. On Asynchrony in Name-Passing Calculi. In Kim G. Larsen, Sven Skyum and

Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, July 1998.

[MT91] I. A. Mason and C. L. Talkott. Equivalence in Functional Languages with Effects. *Journal of Functional Programming*, 1:26–47, 1991.

[Nes99] Uwe Nestmann. Mobile Objects (a project overview). In *Proceedings of FBT'99*, pages 155–164. Herbert Utz Verlag, 1999.

[Nie92] Oscar Nierstrasz. Towards an Object Calculus. In M[ario] Tokoro, O[scar] Nierstrasz and P[eter] Wegner, editors, *Object-Based Concurrent Computing 1991*, volume 612 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1992.

[NN98] F. Nielson and H. R. Nielson. The Flow Logic of Imperative Objects. In *Proc. MFCS'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding Choice Encodings. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 1996. Latest full version as report BRICS-RS-99-42, Universities of Aalborg and Århus, Denmark, 1999. To appear in *Journal of Information and Computation*.

[Ode95] Martin Odersky. Polarized Name Passing. In Pazhamaneri S. Thiagarajan, editor, *Proceedings of FSTTCS '95*, volume 1026 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, 1995.

[OMG] The Object Managment Group. The CORBA webpage. `http://www.omg.org`.

[Pal97] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM, January 1997.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Pit94] A. M. Pitts. Computational Adequacy via 'Mixed' Inductive Definitions. In *Mathematical Foundations of Programming Semantics, Proc. 9th Int. Conf., New Orleans, LA, USA, April 1993*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, 1994.

[Plo77] Gordon Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo90] Gordon Plotkin. Generic Notes on Domains. Notes for Lectures from several Terms at the University of Edinburgh, 1989-90.

[POP96] ACM. *23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)*, January 1996.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proceedings of LICS '93*: 376–385.

[PST99] Gordon Plotkin, Colin Stirling and Mads Tofte, editors. *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999. To appear.

[PT93] Benjamin Pierce and David Turner. Object-Oriented Programming Without Recursive Types. In *Proceedings of POPL '93*. ACM, January 1993.

[PT99] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In Plotkin et al. [PST99]. To appear.

[PW97] Anna Philippou and David Walker. A Rigorous Analysis of Concurrent Operations on B-Trees. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1997.

[Ras99] René Rasmussen. Bisimulering og ko-induktive principper i varianter af ς-kalkylen. Master's thesis, Aalborg University, 1999. In Danish.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).

[San94a] Davide Sangiorgi. An investigation into functions as processes. In Steve Brookes, Michael Main, Austin Melton and David Schmidt, editors, *Proceedings of MFPS '93*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer-Verlag, 1994.

[San94b] Davide Sangiorgi. The Lazy Lambda Calculus in a Concurrency Scenario. *Information and Computation*, 111(1):120–131, 1994. An extended abstract appeared in *Proceedings of LICS '92*.

[San95] Davide Sangiorgi. Lazy functions and mobile processes. Rapport de Recherche RR-2515, INRIA Sophia-Antipolis, 1995.

[San96] Davide Sangiorgi. A Theory of Bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[San98] Davide Sangiorgi. An Interpretation of Typed Objects into Typed π-Calculus. *Information and Computation*, 143(1):34–73, 1998. Earlier version published as Rapport de Recherche RR-3000, INRIA Sophia-Antipolis, August 1996.

[San99a] Davide Sangiorgi. Lazy functions and mobile processes. In Gordon Plotkin, Colin Stirling and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. Massachusetts Institute of Technology, 1999.

[San99b] Davide Sangiorgi. The Name Discipline of Uniform Receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256, pages 303–313.

[San99c] Davide Sangiorgi. The Typed π-Calculus at work: A Proof of Jones's Parallelisation Theorem on Concurrent Objects. *Theory and Practice of Object-Oriented Systems*, 5(1), 1999. An early version was included in the *Informal proceedings of FOOL 4*, January 1997.

[SM92] Davide Sangiorgi and Robin Milner. The Problem of "Weak Bisimulation up to". In Rance Cleaveland, editor, *Proceedings of CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[Sto88] A. Stoughton. *Fully abstract models of programming languages*. Research Notes in Theoretical Computer Science. Pitman, London, 1988.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991.

[SW89] Colin Stirling and David Walker. Local Model Checking in the Modal Mu-Calculus. In *LNCS 351*, pages 369–383. Springer-Verlag, 1989.

[Tal96] Carolyn L. Talcott. Obliq semantics notes. Unpublished note. Available from `mailto:clt@cs.stanford.edu`, January 1996.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint and Its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[US91] D. Ungar and R.B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

[Vas94] Vasco Vasconcelos. Typed Concurrent Objects. In M[ario] Tokoro and R[emo] Pareschi, editors, *Proceedings of ECOOP '94*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.

[VH93] Vasco Thudichum Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In Best [Bes93], pages 524–538.

[Vis97] Ramesh Viswanathan. A first order translation of the Imperative ς-calculus to a first order functional language. Personal email correspondance, November 1997.

[Vis98] Ramesh Viswanathan. Full Abstraction for First-Order Objects with Recursive Types and Subtyping. In LICS'98 [LIC98]. Submitted.

[VT93] Vasco Thudichum Vasconcelos and Mario Tokoro. A Typing System for a Calculus of Objects. In Shojiro Nishio and Akinori Yonezawa, editors, *First JSSST International Symposion on Object Technologies for Advances Software (Kanazawa, Japan)*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, 1993.

[Wal91] David Walker. π-calculus Semantics of Object-Oriented Programming Languages. In Takayasu Ito and Albert Meyer, editors, *Proceedings of TACS '91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991. Available as Report ECS-LFCS-90-122, University of Edinburgh.

[Wal93] David Walker. Process Calculus and Parallel Object-Oriented Programming Languages. In P. Tvrdik, T. Casavant and F. Plasil, editors, *Parallel Computers: Theory and Practice*, pages 369–390. Computer Society Press, 1993. Available as Research Report CS-RR-242, Department of Computer Science, University of Warwick.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

[Yos96] Nobuko Yoshida. Graph Types for Monadic Mobile Processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.