Forms, Agents and Channels

Defining Composition Abstraction with Style

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Franz Achermann

von Buochs, NW

Leiter der Arbeit: Prof. Dr. O. Nierstrasz

Institut für Informatik und angewandte Mathematik

Forms, Agents and Channels

Defining Composition Abstraction with Style

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Franz Achermann

von Buochs, NW

Leiter der Arbeit: Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 29. Januar 2002

Der Dekan:

Prof. Dr. P. Bochsler

Abstract

Object-oriented technology and design is not the final answer to the recurrent problem of making systems, on one hand, more open and flexible and, on the other hand, more robust, safe, and fast. While object-oriented languages have a lot of success in implementing components, they have rather limited support for expressing composition abstractions. As such, the component-based software principle is only partially supported by the object-oriented approach.

Component-based software development breaks down an application into stable parts, i.e., the components, and high-level abstractions for composing the components. Flexibility is provided by the possibility to recompose. How can we design a composition language to support this metaphor? What mechanisms are needed to encapsulate components, to make their contextual assumptions explicit, and to define composition abstractions in a compact way?

We argue that we should seek the minimal kernel of mechanisms that allows us to define composition abstractions, instead of adding additional language constructs to the object-oriented paradigm. This is necessary in order to reason about these abstractions and derive properties of the composed application. In this thesis we propose *Forms, Agents and Channels* as this minimal set of abstractions. Forms are extensible records unified with services. They are primitive objects, act as explicit namespaces, and encapsulate arguments to invoke services. Agents are autonomous entities that exchange forms along channels. We show that this simple model is expressive enough to define high-level composition abstractions while being small enough to be mathematically tractable.

We present the formal model of forms, agents and channels in terms of a composition calculus. We encode the composition calculus into the asynchronous π-calculus and show the soundness of this encoding. We define the composition language Piccola on top of the composition calculus by adding some syntactic sugar and by defining a bridge to access external components. The usefulness of Piccola is demonstrated by defining composition abstractions and reasoning about them at the level of the language. We present several kinds of composition abstractions: wrappers to adapt components, connectors to implement composition styles, and coordination abstractions that cross-cut the functional decomposition of a system. We also demonstrate how to reason about composition and how to use glue code to detect and fix compositional mismatches.

Acknowledgements

First of all, I would like to express my gratitude to Oscar Nierstrasz for giving me the possibility to work in the SCG and for his continuous support for my PhD work. I learned a lot from him, not only about software composition and scientific writing, but also how to target problems that at first seem unmanageable for me. Thank you a lot.

I also want to thank Martin Odersky for writing the Koreferat and for being on the PhD committee.

Many thanks to Uwe Nestmann who helped me to improve the presentation of the formal material in this thesis. Thanks also to Andrew Black who read parts of this work. His comments helped me to better express my ideas. I also thank Thomas Studer for his comments on some earlier drafts.

I also thank my colleagues involved with Piccola. Many thanks to Nathanael Schärli. We had a lot of interesting and fun discussions about Piccola and other important aspects. I also want to thank Stefan Kneubühl, Michael Locher, Andreas Schlapbach (he kindly helped me to improve my English sentences), Cristina Gheorghiu, and finally Jean-Guy Schneider and Markus Lumpe with whom the early idea of forms was born out of intensive discussions.

I thank the members of the SCG for their careful reading and commenting on earlier drafts of my writings, for interesting discussions over lunch, for coffee-drinking and cake-eating together, and for the good time we had: Gabriela Arévalo, Juan Carlos Cruz, Serge Demeyer, Stéphane Ducasse, Isabelle Huber, Michele Lanza, Robb Nebbe, Tamar Richner, Matthias Rieger, Therese Schmid, Sander Tichelaar, and Roel Wuyts.

Last but not least, I thank my parents for their continuous support for me and my ideas. It is good to know that they stand behind me. I also thank my family and friends, in particular Cornelia for being here and for all the love she has for me. Beside God's love for me, being married with Cornelia is the best that could have happend to me. I also thank our son Cyrill for being here.

Franz Achermann
January, 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Component-based software development appears as the dominant solution to one of the hardest and recurring problems we face in software engineering: How can we effectively evolve software in the face of changing requirements? How can we master the complexity that is inherent in big systems? Software composition breaks an application down into manageable pieces, *components*, and specifies an application as the *composition* thereof. Components encapsulate what is stable behind well defined interfaces — flexibility is provided by the possibility to reconfigure, re-compose, or replace components.

The idea of composing a system out of building blocks is not new — in fact it was envisioned as a solution to the first software crisis in the late sixties [McI69]. The goal was to *reuse* components for different applications to leverage development cost and to establish a market for software components. In the last years this idea increased in popularity by the emergence of component models like COM [Rog97] or EJB [MH00] and their use in industry.

1.1 The Problem

How can we design components in such a way that they can cooperate with others that are independently developed? As software engineers we are, more often than we like, faced with the fact that merely assembling components with the expected functionality does not immediately yield the desired system. Components make assumptions about the environment in which they are deployed. If those assumptions are not met, compositional or architectural mismatch [GAO95] occurs and the system will not work as expected. In order to be able to detect and fix mismatches as early as possible we need a way to rigorously express these assumptions.

The object-oriented paradigm builds on the ideas of information-hiding and decomposing problems. While object-oriented languages are good for implementing software components, they have not yet shined in the construction of component-based applications. This is mainly due to the restricted set of composition mechanisms that is available in object-oriented languages, namely message send and inheritance. Object-oriented development principles are not in line with some of the goals of component based software development:

- Instead of focusing on *reuse*, object-oriented analysis and design is domain driven. This leads to designs based on domain objects and non-standard architectures.

- Instead of adhering to *small, restricted* and *plug-compatible protocols*, inheritance leads to rich object interfaces. This makes it hard to isolate the objects from their environments and use them in other contexts.

- Instead of exposing the *object interaction*, the source code exposes the inheritance hierarchy. How objects are connected is typically distributed among the objects themselves. This requires detailed analysis to adapt an application or to effectively use object-oriented frameworks.

- Instead of providing reusable abstractions for object collaborations, object composition is often implemented by following *design patterns*. While we can (and should) reuse design, we often cannot reuse the actual code.

These problems have not only to do with the way how object-oriented languages are used. They are also a symptom that object-oriented abstraction mechanisms are not adequate to express and implement certain composition abstractions as reusable code.

1.2 Approach and Contributions

Instead of adding additional features to object-oriented languages to better support the definition of composition abstractions, we propose a different approach. We advocate a composition language that builds on a minimal set of abstraction mechanisms as a tool to study and express composition abstractions. In this thesis we claim that

> *Extensible composition abstractions can be defined and implemented on a foundation of* forms, agents *and* channels. *A set of plug-compatible components is captured by a* composition style.

In order to validate this claim we must be precise about what it means:

- An *architectural style* defines a family of systems in terms of a pattern of structural organization. It defines a vocabulary of components and connector types, and a set of rules that constrain composition [SG96]. The style determines the plugs each components must have so that they can be connected. Scripts define a specific connection. *Implementing and defining composition abstractions* means to implement connectors as first-class abstraction in a programming language.

- *Forms* are extensible records unified with services. A record contains bindings from labels to nested forms. The service of a form can be invoked. Forms are immutable.

  Forms are composed by form extension and label hiding. Form extension denotes a new form where the bindings and the service either can be overwritten or added as defaults. We can inspect forms and find out what labels they contain. Bindings of a form associate names with services and nested forms — forms represent structure.

- *Agents* are autonomous entities that communicate with each other. The only observable effect of a running agent is communicating with its environment. Agents do something — they represent behaviour.

- *Channels* are the locations along which agents exchange forms. Channels are asynchronous. We can consider a channel as a bag or multiset. We can always add a form to a bag, i.e., send the form to the channel. Receiving from an empty channel blocks unless there is at least one form available. If the channel is not empty, an arbitrary form that was sent to it and that is not yet received gets returned. Channels can store values — they represent state.

To validate our thesis, we present the composition language *Piccola*. A *composition language* is a generic scripting language. A scripting language offers a high-level view of services implemented in more lower-level languages. The language supports composition following this view. While a scripting language offers a fixed paradigm for composing, a composition language supports the implementation of different composition styles tuned with respect to the application domain. A composition language allows the programmer to specify applications as a high-level composition of components.

Components are either external entities that are accessed via a bridging interface or they are scripted from simpler components. Piccola is a *pure* composition language as all computation is eventually performed by external components. A component is accessible through a form as its interface. Piccola defines composition abstractions by composing the forms. Glue abstractions are implemented by using channels and agents.

The language has the expressive power to model composition abstractions. It can embed composition styles as domain specific languages. Moreover, it is based on a simple semantic model that facilitates reasoning about composition. The technical contributions of this thesis are as follows:

- We define the Piccola calculus based on the core concepts of forms, agents and channels. We present an operational semantics of the calculus and define an equivalence relation based on barbed congruence [MS92].

- The calculus is a higher-order variant of the well known π-calculus introduced by Milner *et al.* [MPW89]. We demonstrate that forms add the needed expressive power to express safe and extensible composition abstractions. We give an encoding of the Piccola calculus into the π calculus and show that the encoding is sound.

- We define the composition language Piccola in terms of the calculus. The language has a minimal syntax and supports scripting of external components. We define a language bridge for accessing host components from Piccola. The bridge does not require adaption of these host components in the host language. The components can be adapted inside Piccola and composed seamlessly with other host services. The necessary adaption is transparent to Piccola and the host language. Piccola is a pure composition language as eventually every computation is performed by external components. Components can be adapted to particular composition styles by adapters written within Piccola.

- We present a partial evaluation algorithm for Piccola. This algorithm separates side-effects from referentially transparent computation. Side-effects are executed in the right order and referentially transparent services are inlined. This optimisation eliminates the performance costs of generic glue wrappers. We prove correctness of the algorithm with respect to the Piccola calculus.

- We capture a set of components and connectors into a composition style. The style supports a high-level and declarative way to compose. We present the embedding of a nontrivial object-oriented framework into Piccola and wrap the framework as a composition style. As an example we chose a framework whose composition mechanism are procedural or wiring-based. Wires are the gotos of component based software development. We demonstrate that plugging an application by using the style simplifies composition. It helps to understand the code as it is more declarative and makes the architecture explicit in the code.

- We show that channels, agents and the generalization of forms into namespaces allows us to define control and coordination abstractions as reusable abstractions. In conventional programming languages such abstractions are often defined as language primitives or are implemented by mechanisms such as meta-programming, macros or continuation-passing. The approach of using agents and channels is more direct.

- We present how to derive the states and transitions a Piccola program can have directly from the code. The resulting model is simple enough so that we can detect composition mismatches. We use model checking to derive algebraic properties of composition abstractions.

- We present an approach that uses wrappers to formalize the assumptions a component makes about its environment. If we can assert that these assumptions hold, the wrapper is not needed. Otherwise, we apply the wrapper to the component in order to protect the component from bad usage and to overcome compositional mismatch [GAO95]. We can decide whether a composition with a wrapped component has the same behaviour as the composition without the wrapper. If this is the case the environment fulfills the assumptions of the component.

1.3 Thesis Outline

This thesis is structured as follows.

- In Chapter 2 we present the requirements for a composition language, discuss related work and motivate our approach. We introduce the necessary terms used throughout this work.

- Chapter 3 is devoted to the development of the Piccola calculus. The calculus is a higher-order variant of Milner's π-calculus where forms are the communicated values. The calculus has explicit namespaces that are reified as forms. It is a higher-order calculus that communicates abstractions as first-class values. Since values play such a central part we define the value, i.e., the result of a parallel composition. As a consequence parallel composition is not commutative as in traditional process algebras. The parallel composition operator has the flavour of spawning off a parallel agent from the main thread of control.

  We present the syntax and reduction semantics of the Piccola calculus and define a notion of equivalence based on barbed congruence.

- In Chapter 4 we introduce the π-calculus, more specifically a restricted variant of it where agents may only send but not receive values along channels that they have not created themselves. They can only read from local channels. This localized π-calculus has recently been proposed by Sangiorgi and Merro [Mer00, MS98]. It turns out that the $L\pi$-calculus is powerful enough to embed the Piccola calculus. We present an encoding of Piccola into the $L\pi$-calculus and prove soundness of the encoding. This property says that whenever two translated agents are equivalent in the $L\pi$-calculus, then they are also equivalent in the Piccola calculus. This allows us to use proof techniques from the π-calculus for reasoning about Piccola agents.

- The language Piccola is defined by adding some syntactic sugar on top of our composition calculus. We present the language in Chapter 5 and give a denotational semantics in terms of the calculus. In order to keep the language small we adhere to the principle that everything is a form. The language contains the notion of the dynamic context and simplifies the definition of recursive forms. An explicit handle on the dynamic context turns out to be useful to model context dependent abstractions. Recursive forms are essential for defining recursive services and for modeling the self reference of objects. The semantics of recursive forms is given by channels and we show that recursive services can equally be defined by a lazy fixed-point combinator from the λ-calculus.

  In this chapter, we also present the language bridge from Piccola to external components by describing the Java-Piccola bridge. The bridge allows us to seamlessly integrate components and services composed from Piccola and external components and services, respectively.

- Chapter 6 presents a partial evaluation algorithm for Piccola. We focus on the functional subset of Piccola and treat channel creation and spawning of new agents as external services that contain a side-effect. The chapter proves correctness of the algorithm. Although it is feasible to run a partially evaluated expression in a standard Piccola interpreter, a specialized expression has properties that allow for much faster execution. Partial evaluation removes much of the overhead introduced by generic glue wrappers. It inlines all indirection caused by renaming or the specification of default values.

- In Chapter 7 we demonstrate the expressive power of Piccola to define extensible composition abstractions. First, we present reusable connectors to declaratively script components in a composition style. Second, we present control and coordination abstractions used to implement these styles. Third, we present extensible wrappers that add the connectors to bare components which offer the functionality so that they fit the style.

  As a running example we develop a scripting language for composing GUI dialogs embedded into Piccola. We present the difference between procedural wiring and declarative plugging using the familiar example of a pipes-and-filters style. We evolve this style so that events are the data elements processed and we combine this style with a style for doing GUI layout. The layout style consists of external components from the Java AWT framework that are adapted in Piccola.

- In Chapter 8 we show how to reason about composition abstractions at the Piccola language level. We show how to derive the states and their possible transitions to do model checking almost directly on the code. This allows us to detect and overcome compositional mismatches. As examples, we introduce generic glue code wrappers that bridge composition styles.

It turns out that these wrappers also help formalizing the assumptions a component makes about its run-time environment. If these assumptions hold for a particular composition, the glue code wrapper is not necessary. Otherwise, we apply the wrapper so that it protects the component from illegal use.

- Chapter 9 concludes this thesis and discusses future work.

If the reader is mainly interested in seeing Piccola at work, he should browse Chapter 5 to get an overview of the language. Chapter 7 demonstrates the implementation of composition styles and Chapter 8 shows how to safely glue styles together. The formal foundation of Piccola is defined in Chapter 3. The calculus is embedded into the π-calculus in Chapter 4 and used in Chapter 6.

Chapter 2

Software Composition

A lot of work has been done regarding the composition of software components. In this chapter we give an overview of this work and motivate our approach using Piccola. We first define the notion of a software component. We state our requirements for a composition language and review how these requirement are met in existing approaches and explain our approach.

2.1 Components and their Environment

A software component cannot be differentiated from any other software element only by the programming language that implements the component. A piece of software becomes a component when it can be deployed and used in component framework. Heinemann *et al.* [HC01] define the terms component, component model, and infrastructure as follows:

> "A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
>
> A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.
>
> A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications."

This definition makes it clear that a component cannot be considered in isolation, but that it only makes sense if there is a supporting environment to host it. To put it more compactly: *A component is designed to be composed* [ND95]. *Composition* is the combination of two or more components yielding a new component behaviour. The characteristics of the composite are determined by its parts and the way they are combined.

There are other aspects of components as well. Szyperski argues that they have no state and are binary deployable [Szy98]. The distinction between stateless component factories and stateful component instances is blurred. The confusion is caused as components, like buttons or windows, often do have state. We are thus not working with the abstract component classes, but with their instances. It is therefore useful to distinguish between design-time and run-time of a component. At design-time the developer chooses the properties that are set when the component is instantiated. The requirement for binary packaging of components does not mean that a component must be delivered in some compiled format. It specifies that a programmer can readily access the component without the need to manually adapt either the component or its environment.

2.2 Requirements and Related Work

Open applications are characterized by three key requirements [NM95]: applications are inherently concurrent and distributed, they run on a variety of software platforms, and applications requirements evolve during the lifetime of a software. A composition language addresses these requirements by supporting a view in which applications are built by scripting components. A composition language must therefore address the following requirements:

1. *Expressiveness:* A composition language must support the definition of components and abstractions to compose them as first-class citizens. This makes the relations between the components and the architecture explicit in the code.

2. *Formal model:* Formal reasoning techniques validate that applications ensure specified properties. A formal model is also needed to bridge between different component models.

3. *External components:* Applications are composed from components that may be implemented in different languages and run on a variety of systems.

In the rest of this section we give an overview of related work that addresses all or some of the above requirements. We consider the following areas: In the first subsection we look at how to implement component *frameworks*. We consider the expressiveness of object-oriented languages to support the definition of such frameworks. Second, we consider how the *architecture* of an application can be made explicit and how we can reason about properties at this level. Third, we have a brief look at *domain-specific* and *scripting languages* and define the notion of *embedding* such languages into host languages. Scripting languages give a high-level view to components and the services they offer. Finally, for addressing our last requirement we take *heterogeneous* systems into account.

2.2.1 Frameworks

As mentioned, a piece of software only becomes a component if it may be hosted by a component integration framework. It is the framework that establishes and ensures the assumptions required by the component.

A *framework* is a structure or skeleton for a project. In the following we consider object-oriented languages since these languages are the latest incarnation of software engineering principles like encapsulation and separation of concern [Cle95]. In the context of object-oriented languages, a framework is a set of cooperating classes that makes a reusable design [FS97, JF88]. Some of the classes may be abstract. They must be subclassed by the application developer.

A white-box framework supports specialisation by subclassing. The hot-spots are the locations where application specific code can be provided [Pre95]. The framework code calls the application code. This principle is known as the Hollywood principle, i.e., don't call us, we'll call you [Vli96]. Frameworks must evolve: from a set of a few examples to a white-box framework, to pluggable objects, and finally to a black-box or component-oriented framework [RJ97]. At this most mature stage the domain knowledge is sufficiently stable to merit a domain-specific language to script the framework. Examples include GUI frameworks where the application code is generated by 4GL GUI-builders. We will consider this point again in Section 2.2.3.

The component framework consists of sets of components together with an architecture that defines the interfaces of the components. The framework also defines rules governing composition [SG96].

The problem with object-oriented frameworks is that the code often does not make the architecture or the composition rules explicit. This is caused by the fact that the framework uses certain composition abstractions which are not expressed as first-class entities. As a result, these abstractions are split throughout the framework code. Such abstractions are typically recorded as design patterns. *Design patterns* are used to record experience of successful object-oriented designs and architectures. A pattern represents a composition abstraction implemented in an object-oriented language [GHJV95].

The fact that design patterns and composition abstractions in general cannot be implemented as first-class citizens in many object oriented languages stimulated research to improve the expressiveness of those languages. In the following we list these extensions and techniques.

- Implementations of design patterns as reusable code go beyond the classical paradigm of object-oriented languages. Such implementations use reflection [Duc97], macros [Sou95], or generators to produce the pattern specific code [BFVY96].

  Reflection is the ability of a program to manipulate the state of the running program as data. There are two aspects of reflection: *Introspection* allows a program to observe and read its own executing state whereas *intercession* is the ability to change its own state as well [BGW93]. Intercession changes the semantics of certain language constructs.

- *Mixins* [Bra92, BPS99] specify behaviour orthogonal to the inheritance relation. Mixins are abstract subclasses, i.e., classes with an unspecified parent class. Mixins allow much more flexibility than static subclassing. Ernst [Ern99] has demonstrated that mixins with a richer set of combinators have equal expressiveness as AOP or SOP (see next items) and are statically type-checkable.

- In the last years, *Aspect-Oriented Programming* (AOP) has emerged as a methodology for overcoming reduced expressiveness of (object-oriented) programming languages. An aspect is a feature of a system that cannot be isolated and implemented at a single point in the source code [KLM+97]. Aspects are implemented by weaving them into the source code. The join-points are the locations where an aspect is coordinated with the base components. Examples of join-points are the enter or exit of a method. An AOP-language specifies the functionality of an aspect and join-points at which the aspect should be merged with the class. The join-points allow finer control than inheritance which only supports overriding of methods.

A recent proposal is to make join-points dependent on dynamic context information. For instance, depending on what the sender will do with result, we can associate different behaviour to a join-point [CKFS01]. AspectJ [KHH+01] is the aspect weaver for the Java language.

- *Subject-Oriented Programming* (SOP) [HO93, OKH+95, TOHS99] is similar to AOP. In SOP, disparate class hierarchies, i.e. subjects, representing different concerns are composed. Subject composition operators have richer semantics than just overwriting methods has. SOP supports slicing concerns and recomposing them.

- In the work on *Composition Filters*, Aksit *et al.* [AWB+94, BAW93] propose a layered object model in which the layers control message passing. The object itself encapsulates only its state. In the layered model, the programmer can control message send at the caller side as well as at the client side. Bosch illustrates how this model allows the implementation of many design patterns as first-class abstractions [Bos97].

- In his work on the *Demeter methods*, Lieberherr proposes the use of adaptive techniques to make object-oriented programs more reusable [LSLX94]. In this approach, class structures are allowed to change by separating the relationship of a class with other classes out of the classes themselves. Only constraints on the class structure are defined [SPL98]. The final program is generated by applying propagation patterns that describe how to carry out a specific task while traversing the object structure.

- *Nested classes* remove the restriction of traditional object-oriented languages that only allow methods and members inside classes. This allows the programmer to encapsulate whole frameworks as a single class and to have several instantiations of the same framework inside a single application. This has several advantages. For each framework instantiation of the same application, the global variables can be instantiated differently. Under the classical (flat) class regime, global framework variables have a unique instantiation for the framework. Furthermore, the framework is not encapsulated as a whole leading to framework composition problems [MB97]. An example where specializing a framework several times is useful is given in the class `Simulation` from Simula [DMN70]. This class consists of a framework for discrete simulation processes and multiple different processes are normally active in one application.

  Batory *et al.* use Java's inner classes to define mixin layers. Such mixins layers specify a refinement for product-line architectures (see Section 2.2.3) [BO92, SB98]. Seiter *et al.* use the enhanced expressiveness of inner classes to support dynamic framework instantiation [SML99].

  Anonymous classes support the definition of a class with an immediate instantiation of an object of this class. Anonymous classes can be used to create singleton objects, i.e., classes from which there is only one instance in an application. Hedin [HK99] demonstrates the usefulness of anonymous classes for adapting frameworks and for the specification of pluggable objects [KP88].

- *Generalized inheritance* allows subclasses to specialize parts of methods. The language BETA [MMPN93] provides the *inner* construct for this purpose. With outer inheritance we have to use the template method pattern to define and specialize sub-methods, leading to a proliferation of the messages understood by a class. Using inner inheritance, we can specialize the same method several times with different signatures inside the same subclass. For instance, this allows us to implement a generic monitor in BETA [MMPN93] which is not possible in languages that use outer inheritance like Smalltalk or Java. In Java we have to describe a monitor as a pattern [Lea99]. The problem with outer inheritance is that we have to fix the signature up front and cannot reuse the same synchronization schema for different methods in the same class. BETA's inner construct is similar to the *call-next-method* construct for *around* methods in CLOS [Kee89]. Agerbo *et al.* demonstrate that many of the classical design patterns can be implemented as reusable, first-class abstractions due to BETA's generalized block structure and inheritance [AC97].

A formal semantics is needed in order to combine components written in these different models and to support reasoning about these models.

2.2.2 Software Architecture

The architecture and the rules defined by a component framework should be explicit. An *architecture* describes a software system in terms of computational elements and interactions among these element [BCK98, PW92, SG96]. The goal is to make the process of architecting more rigorous and to build systems with a more reliable and reusable architecture. In the field of software architecture this can be achieved by using (formal) specification languages, called *architectural description languages* (ADL) and by using corresponding analysis techniques. The associated theory influences the ADL's ability for modeling particular kinds of systems.

Architectures can be classified into *architectural styles*. A style is an abstraction over a set of related software architectures. It defines a vocabulary of component and connector types and constraints how they can be composed. We can classify architectures and associate properties with different styles [SC96].

Components embody assumptions about the environment in which they operate. At composition time, the engineer is faced with the problem of uncovering and avoiding *architectural* or *compositional mismatch* [GAO95, Sam97]. A mismatch occurs whenever it is impossible to successfully interconnect components with existing connectors. Compositional mismatch is fixed by replacing or rewriting the components or applying glue code. *Glue code* adapts components from one component model to another one and therefore overcomes compositional mismatches [DW99]. Wrappers [BW00, Höl93], Bridges, Proxies and Mediators [GHJV95] are some techniques to implement glue code.

Mismatches arise from conflicts at two levels of interaction. One problem is the compatibility of the data exchanged among the components. Such mismatches are usually captured by the type of information present in the interfaces. Type analysis is used to detect those mismatches and data-conversion is used to overcome them. The other, more difficult compatibility problem is the dynamic interaction and communication behaviour between the components [CIW99, GAO95].

An ADL can focus on different aspects: While some ADL's (like CHAM, Wright or Rapide) focus on formal analysis methods, others focus on formally specifying the architecture to make it explicit.

- Inverardi and Wolf use the *chemical machine metaphor* (CHAM) to specify and analyse architectures [IW95]. The CHAM formalism was developed by Berry and Boudol [BB92] as a general computational framework. It works by defining molecules as terms of a syntactic algebra and solutions, i.e., multisets of molecules. The state of a CHAM are its solutions. The machine specification consists of a set of transformation rules of one multiset into another. The transformation rules define the behaviour, i.e., how state evolves. Analysis is done by model checking.

- The *Wright* specification language [AG94, All97] specifies the behaviour of connectors in terms of a subset of CSP [Hoa85]. As long as the specification of the connectors is satisfied, the connections between the components can be reconfigured. Formal analysis of Wright focuses on two properties: the first is deadlock freedom which is extensively studied in the context of CSP. The second guarantees that the ports of the connectors and the roles played by components match. This is specified by requiring identical protocols on both sides or one being a *refinement* of the other.

- *Rapide* [LKA+95] is based on specifying the order in which events occur in a system. An event is a very flexible and abstract notion that allows for arbitrary detailed specification, depending on the particular events of interest. A simulation is used to check consistency of interfaces and connections and to verify the systems overall communication structure. Event patterns are specified as partially ordered event sets. Analysis in Rapide amounts to checking for proper orderings of events and causality among them.

- In *Darwin* [EP93, MDEK95, RE94] the interface of a component is described as a collection of provided and required services. Configurations are developed by component instantiation declarations and bindings between required and provided services. Darwin provides semantics for its structural aspects through the π-calculus and supports the definition of an architectural style through the description of a parameterized configuration. The connector type of Darwin is asymmetric due to the underlying differentiation of provided and required services.

- The language *UniCon* [SDK+95] provides a richer set of connectors than Darwin. Each connector has a collection of roles that define what a participant can expect in a given interaction.

- Le Métayer [Mét96, Mét98] proposes the use of *graphs* to model software architecture. The approach makes a distinction between the specification of a single component and the overall structure. A graph represents the architecture by interpreting nodes as components and the edges as connectors. A style is expressed as a graph grammar. The evolution of a system is governed by a coordinator component who performs graph rewritings. In this approach it is possible to check if a coordinator follows the style.

Other examples of ADL's are given by Abowd *et al.* who uses Z [Spi89] for specifying architectural styles [AAG93] or the language LILEANNA [Tra93] that is based on the algebraic formalism OBJ [DF98]. For a broader classification of ADL's we refer to the survey of Medvidovic [MT97].

Each ADL addresses a different set of properties that can be detected. It is necessary to combine the findings of the different tools. A step in that direction is done by Garlan *et al.*

with their work on Acme. Acme is a generic ADL that serves as a common representation of architectures and that permits the integration of architectural analysis tools [GMW00].

2.2.3 Domain-specific Scripting Languages

Program families are collections of software elements related by their commonalities [Par76]. Different family members are differentiated by their variations. A *product-line architecture* is a design for a program family or product-line that identifies the underlying components. It enables the composition of these components to instantiate different applications of the same domain [LHB01].

A *domain-specific language* is a small, usually declarative, language specially designed for a particular domain. Domain-specific or *little languages*, as they are also called, enhance quality, flexibility and timely delivery of software because they take advantage of specific properties of the domain [vDKV00].

A domain-specific language packs the engineering effort that went into the development of the product-line architecture and the implementation of the components into a high-level language. This language allows casual users to script their own applications. A *scripting language* is a high-level language to create, customize and assemble components into a predefined architecture [SN99]. A scripting language typically embodies a specific composition style together with features for general programming. For instance, Perl [WCO00] provides regular expressions to work on a number of string buffers whereas the Unix shell languages are designed around the pipe-and-filter style.

2.2.4 Embedding Languages

There are two approaches to the implementation of a language. One approach is to implement a language from scratch, involving tasks like defining a syntax and semantics and implementing the required software. The other approach is to embed the little language by extending another language with just the constructs needed. Implementing from scratch uses but hides the available technology; embedding shares and reuses technology from the host language [CGKF01]. Embedding has the advantage that the programmer can focus on the domain specific aspects, whereas general programming elements like variables and loops are handled by the host language. Embedding leverages the learning effort needed by new users: they only have to know the base language and the additional domain specific abstractions.

Hudak introduced the notion of embedding a little language into a (functional) host language [Hud96]. A composition language is a generic scripting language. It should be possible to wrap component frameworks as little languages embedded into the composition language. The component frameworks may be written within the composition language or may be external to it.

In functional languages, a function is the generalized abstraction mechanism [ASS91]. Some functional languages like Haskell support infix operators as higher-order functions. Such operators make it possible to embed little language as grammars, similar to arithmetic expressions. A particular advantage when embedding a little language into statically typed languages like Haskell or ML is the reuse of the strong type system for the domain to be modeled [JML98]. This form of composition is widely applied in functional languages like Haskell or ML [CH98, Fai87, Hud98, MSC99].

However, certain wiring abstractions cannot be easily expressed in a functional setting. Examples are abstractions that introduce variable bindings or affect the flow of control in a non-standard way. Kamin *et al.* have embedded FPIC, a variant of PIC [Ben86] which is a language to represent two-dimensional pictures into ML [KH97]. They illustrate the problem of name bindings. When one picture refers to points defined in another picture and the two pictures are composed, then the user has to use strings instead of variables to refer to those external points.

To overcome these restrictions, macros and monads extend the expressiveness of the (functional) language. Shivers embedded AWK into Scheme using Scheme's macro facilities and reports that his system is more powerful than the original using one tenth of the originals size and was implemented in a few days [Shi96]. Other examples of little languages embedded into Scheme can be found in [CGKF01, WR99].

Monads [Mog89, Wad95] are used to model state in a language without side-effects. A monad encapsulates and hides the global state and allows functions to modify the state in a disciplined way. Monads can be used to model variables, exception handling, output and non-determinism. Liang *et al.* have demonstrated how monad transformers can be used to build components and connectors for the construction of a modular compiler [LHJ95].

2.2.5 Heterogeneous Systems

Middleware is the software that enables different languages to cooperate. In the area of component based software development, component models are most important. The most commonly used component models are COM [Rog97] with several incarnations (OLE, ActiveX, COM, DCOM, COM+, .NET), Java (Enterprise) Beans [Mor97, MH00], and the Corba Component Model that is part of Corba 3.0 [CCM]. We include Java and the older Microsoft variants in this list, since there exists standard mappings for these components to make them cross-platform. A detailed overview and comparison of these models is beyond the scope of this work. We refer the reader to the literature [Lon01].

The basic idea of these component models is to make the location of a component transparent. The location may be on a different platform on a different system. A component is described by an interface which specifies the services offered by the component. Furthermore, the interface may describe what events the component can generate.

IBM's System Object Model (SOM) [Lau94] allows programmers to access objects written in separate languages. Object are distributed in binary form and, in addition, can be subclassed across different languages. As such, SOM is based on an advanced object model and an object-oriented runtime engine that supports this model.

In the CHAIMS project [SBMW99, KMF01] services have additional parameters like their reliability or performance. Composite services can call different services with different parameters and, for instance, terminate pending requests when the first service has returned. Such composition mechanisms are also studied in the context of web-scripting [CD99].

2.3 Forms, Agents and Channels

We have collected a number of requirements a composition language must fulfill: It must be expressive enough to implement components and high-level composition abstractions, it

Applications	Components + Scripts
Composition styles	Streams, GUI composition, ...
Standard libraries	Coordination abstractions, control structures, ...
Piccola language	Host components, user-defined operators, dynamic namespace
Composition primitives	forms, agents and channels

Table 2.1: Piccola Composition Layers

must support reasoning at the language level, and it must support the accessing of external components.

Coplien [Cop99] argues that a pure object-oriented paradigm does not offer enough flexibility to define composition abstractions. He claims that other language features as found in C++ like template methods, overloaded operators, or procedural programming are needed to have enough expressive power to implement composition abstractions.

Instead of making the union of paradigms — we claim — we should seek the minimal kernel that allows us to embed composition styles, while not giving up a simple semantic model for the language. In this way we will be able to reason at the language level. Our thesis is that a foundation for a composition language should be given in terms of forms, agents and channels. To validate this claim we designed the composition language Piccola that builds on the ideas of forms, agents and channels. We use forms as unifying concept for component interfaces, abstractions, arguments, and namespaces.

Similar approaches are taken in the languages Funnel and Oz. Funnel [Ode00] is a programming language based on the generalization idea of functional nets. A functional net is the combination of functional languages and join-patterns from the join calculus [FG96]. The language Oz [Smo95] subsumes functional, object-oriented, and logic programming aspects into a coherent system.

2.3.1 Design Guidelines for Piccola

We design Piccola according to the following guidelines: A layered approach is used to bridge the gap between the foundation of forms, agents and channels and high-level scripts; A simple semantic model supports formal reasoning techniques; The idea of generalization ensures expressiveness.

Piccola composition layers. The gap between a high-level script in Piccola and the foundations given by forms, agents and channels is closed by a set of layers. These layers are given in Table 2.1. Using the composition primitives of forms, agents and channels we define the Piccola language. The language has additional syntactical sugar and defines basic data-types as host components. In the language we define control and coordination abstractions. Examples of such abstractions are the if-then-else or the try-catch abstractions or generic adapters for wrapping forms.

Using these abstractions we define *composition styles*. A style defines component algebras where a component is of a particular sort, and composition corresponds to operations in the algebra. Using these styles, we propose a compositional approach for building applications. Scripts are expressions of the underlying style(s) that can easily be re-composed. The layers are transparent since we embed the styles on top of Piccola instead of building isolated domain-specific languages. Thus the programmer can combine styles or extend them inside Piccola.

Simple Model. Forms are much simpler than objects. Their primary difference is that they do not have an identity and that they are immutable. The composition operators for forms are extension and label hiding. We can inspect a forms and learn about its services at runtime. This allows us to define generic wrappers to adapt forms.

External components are embedded into Piccola as forms. In our current implementation we use reflection of the host language (Java or Squeak) to wrap host objects as a form. We argue that other component models can equally be integrated. For instance, a Unix process can be embedded as a component that requires standard input and output streams and provides services to send signals. External components must seamlessly integrate with the composition language. This means that we want to uniformly manipulate components whether they are written in Piccola or in external languages. Expressions that include services written in Piccola and external services should transparently wrap or unwrap external components. By using nested forms, this requirement can be achieved almost for free as we will show in Section 5.8.

Apart from forms Piccola uses agents and channels. The π-calculus has considerably firmed our understanding of parallel composition and of scope extrusion to generate and use private resources. Channels and agents can be used to define abstractions that modify the flow of control and they can bridge compositional mismatches.

Generalizing. We can learn from functional languages or the language BETA that much expressive power is gained when certain concepts are generalized and applied uniformly. One of the cornerstones of Piccola's expressive power is the ubiquitous use of forms.

- Forms are *interfaces to components*. Components may be scripted inside Piccola or they may be adapted, external components.
- *Abstractions* over forms are unified with forms. This leads to higher-order forms and services.
- Forms are *arguments* in applications. This supports keyword based arguments passing and default arguments.
- The *environment* or scope at each point in a program is explicitly available as a form that can be read or modified. This allows a script to learn about new services at runtime. Agents can receive scripts and execute them inside a sandbox. A script is just an abstraction that considers its argument to be the environment, i.e., the argument must contain bindings for all free variables in the script.

In the rest of this work we present the model of forms, agents and channels in detail, show that it has the required expressive power and supports reasoning at the language level.

Chapter 3

A Composition Calculus

In this chapter we present the Piccola calculus. The calculus defines the notion of forms, agents and channels. These composition primitives are at the heart of Piccola's composition layers. The calculus combines ideas from the asynchronous π-calculus and from lambda-calculi with explicit substitution. We first present the ingredients of the calculus in Section 3.1. In Section 3.2 we introduce the Piccola calculus and define syntax and semantics of it. In Section 3.3 we demonstrate how to specify recursive services in the Piccola calculus. Section 3.4 shows a few example reductions in the calculus to get a feeling for its expressiveness. In Section 3.5 we define an equivalence relation for agents. Section 3.6 discusses how to avoid erroneous agents in Piccola. In Section 3.7 we define a canonical form for agent expressions and in Section 3.8 we prove that the beta reduction is a valid law. In Section 3.9 we explain the differences between the Piccola calculus and the Form- and the $\pi\mathcal{L}$-calculus. Section 3.10 summarizes and concludes the chapter.

3.1 Requirements and Related Work

The Piccola calculus is a combination of the asynchronous π-calculus with higher-order abstractions. It uses asymmetric parallel composition and contains records as first-class environments.

π-calculus. The π-calculus is a calculus of communicating systems in which one can naturally express processes with a changing structure. Communication between neighbors can carry arbitrary information including how to change the topology [MPW89, Mil91]. The π-calculus can be seen as the reference calculus for concurrent computation. Many important properties of concurrent and distributed systems can be studied within this framework. Its theory has been thoroughly studied and many results relate other formalisms or implementations to it (e.g. [Mil92, Smo94, Wal95, SW01]).

Many different variants of π-calculi have been introduced. Some of the differences are minor choices of notation and style, other are important choices that are driven by a particular application. There are two different areas where the π-calculus is applied: *Modeling* and *Programming*. The difference is manifested by different operators that are defined in the calculus. For modeling or specification we need a richer set of operators. An example is the non-deterministic or internal choice operator to specify that a component may exhibit different behaviour depending on its internal state. From a programming and implementation point of view it is preferred to omit such operators to make the language simpler. The Pict experiment has shown that the π-calculus is a suitable basis for programming many high-level construct by encodings [PT00]. For example, Pict does not contain primitives for choice or matching.

Piccola adopts from the π-calculus the concept of lexical scopes for channels. These scopes are widened implicitly on communication, a principle called scope extrusion. The term $(\nu a.P) \mid Q$ denotes two processes P and Q in parallel and P has a local channel a. The scope of channel a is P. If Q does not contain a free we can widen the scope of the channel to include Q. The parallel process is the same as $\nu a.(P \mid Q)$, i.e., where the scope of a includes process Q. This is what happens when process P sends a to Q.

Asynchronous channels. The amount of synchronization provided by the communication primitives is an important aspect to distinguish between different paradigms of message passing. The basic issue is whether communication is synchronous or asynchronous. If it is synchronous, the sender of a message knows that the message has been consumed. Sending messages blocks. In an asynchronous world, the sender has no implicit knowledge whether a message sent is already consumed by a receiver. For programming and implementation purposes, synchronous communication seems uncommon and can generally be encoded by using explicit acknowledgments (c.f. [HT91, Bou92, Pal97]). Moreover, asynchronous communication has a closer correspondence to distributed computing [Woj00].

Furthermore, in the π-calculus the asynchronous variant has the pleasant property that equivalences are simpler than for the synchronous case [FG98]. Input-guarded choice can be encoded and is fully abstract [NP96]. For these reasons we adopt asynchronous channels in Piccola.

Higher-order abstractions. Programming directly in the π-calculus is often considered like programming a concurrent assembler. When comparing *programs* written in the π-calculus with the lambda-calculus it seems like lambda abstractions scale up, whereas sending and receiving messages does not scale well. There are two possible solutions proposed to this problem: We can change the metaphor of communication or we can introduce abstractions as first-class values.

The first approach is advocated by the Join-calculus [FGL$^+$96, Fou98]. Communication does not happen between a sender and a receiver, instead a join pattern triggers a process on consumption of several pending messages. Such join patterns have a close similarity with an encoding of lambda terms using a continuation-passing style. The resulting encoding of the lambda calculus is simpler than encodings into the π-calculus [FG96]. Another example of changing the channel metaphor is the Update calculus of Parrow *et al.* [PV97]. In this calculus, the asymmetry of non-blocking send and blocking receive is resolved and both primitives are non-blocking. In Odersky's applied π-calculus, processes return a channel name [Ode95]. The Blue calculus of Boudol [Bou97] changes the receive primitive into a definition which is defined for a scope. By that change, the Blue calculus is more closely related to functions and provides a better notion for higher-order abstraction. Boudol calls it a continuation-passing calculus.

The other idea is used by Sangiorgi in the HOπ-calculus. Instead of communicating channels or tuples of channels, processes can be communicated as well. Surprisingly, the higher-order variant has the same expressive power as the first-order version [San93, San01].

In Piccola we take the second approach and reuse existing encodings of functions into the π-calculus as in Pict. The motivation for this comes from the fact that the HOπ-calculus itself can be encoded in the first-order calculus.

Asymmetric parallel composition. The classical way to describe concurrent systems is by giving a sequence of configurations or by prescribing how a configuration may evolve. This model uses a formalism based on processes, i.e., mere computations that do not return a result. An expression, on the other side is a term that returns a value as its result. Models of objects and functions in the π-calculus like in any process calculus translate an expression into a process that sends a message on a result channel using continuations. Such encodings are important to show the expressiveness of the calculus. However, such encodings also obscure the understanding of a program. We argue that expressions with a result are such a fundamental aspect of composition that they deserve a semantics in their own right. We directly describe the semantics of Piccola expressions rather than giving them via an encoding.

When expressions play such a central role we abandon the concept of a symmetric parallel composition operator. The parallel composition $A \mid B$ of two expressions A and B is an expression that runs A and B in parallel. Any result returned by B is returned as the result of the whole expression. The result returned by A is discarded. As a consequence, unlike in the π-calculus, the effects of $A \mid B$ and $B \mid A$ are different. Running $A \mid B$ can be implemented by forking off A as a new thread and then running B. The semantics of asynchronous parallel composition is used in the concurrent object calculus of Gordon and Hankin [GH98] or the (asymmetric) Blue calculus studied by Dal-Zilio [DZ99]. The same concept is also present by Ferreira *et al.* [FHJ96] to give a compositional semantics to CML [Rep91].

In the higher-order π-calculus the evaluation order is orthogonal to the communication semantics [San01]. In Piccola, evaluation strategy interferes with communication, therefore we have to fix one for meaningful terms. For Piccola, we define strict evaluation which seems appropriate and more common for concurrent computing.

Record calculus. When working with components and interfaces, a record based approach is the obvious choice. We use *Forms* [Lum99, LAN00] as an explicit notion for extensible records. Record calculi are studied in more detail for example in [CM93, Rém94]. A record has fields which are accessed by projection. Extensible records support a concatenation operation. Records can be modeled in two different ways. On one hand, we can compile the records away and use offsets and arrays and model records as syntactic sugar. On the other hand, we can keep the records as explicit runtime objects. For efficiency reasons one might prefer the first approach, for highest flexibility one might prefer the latter. However, the dictionary approach normally makes reasoning very hard, since a program may arbitrarily change its methods.

The forms used in Piccola are somewhat in between. They support some introspection but not intercession. It is possible to inspect a form. Form inspection returns arbitrary label bound in the form. A label is reified as a form with three services: one to project, one to hide, and one to bind a new form using the encoded label. Such an operation is necessary to define agents that learn new capabilities at runtime. Furthermore, this allows us to iterate over the labels of a form and specify generic wrappers.

In the λ-calculus with names of Dami [Dam94] arguments to functions are named. The resulting system supports records as arguments instead of tuples as in the classical calculus. The λN-calculus was one of the main inspiration for our work on forms without introspection.

An issue omitted in our approach is record typing. It is not clear how far record types with subtyping and runtime acquisition of new names can be combined. An overview of record typing and the problems involved can be found for example in [CM93, Car93].

Explicit environments. An environment is a set of variable-value pairs. In most languages, the environment gets compiled away and is not available for inspection or modification at runtime. However, many scripting languages like Perl or Python provide a built-in service `eval` which evaluates dynamically created code. This is often referred to as the eval-feature [Ous98]. It gives the programmer the expressive power needed to express what he could not — or not so compactly — program otherwise. For example this is needed to acquaint new components at run-time and plug them in correctly. From a foundational standpoint however, such an abstraction is rather ad-hoc and lacks a well-defined semantics [SN99].

An explicit environment generalizes the concept of explicit substitution [ACCL91] by using a record like structure for the environment. We claim that many of the uses of an eval-feature can be expressed by explicit environments. In the environment calculus of Nishizaki, there is an operation to get the current environment as a record and an operator to evaluate an expression using a record as environment [Nis00, SSB99]. Projection of a label x in a record R then corresponds to evaluating the script x in an environment denoted by R. The reader may note that explicit environments subsume records. This is the reason why we call them forms in Piccola instead of just records.

Handling the environment as a first-class entity allows us to define concepts like modules, interfaces and implementation for programming in the large within the framework. To our knowledge, the language Pebble of Burstall and Lampson was the first to formally show how to build modules, interfaces and implementation, abstract data types and generics on a typed lambda calculus with bindings, declarations and types as first-class values [BL84]. Postscript [Ado90] is another language that includes first-class environments.

Ferrari *et al.* show a π-calculus with explicit substitution [FMQ96]. However, they do not use it for explicit namespaces, but to define a structured operational semantics (SOS) [Plo81] for the π-calculus. The explicit approach makes it possible to use different name instantiation schemas for early and late semantics.

3.2 The Piccola Calculus

Before defining the syntax, we have a look at an example that demonstrates the explicit environments of Piccola. The Piccola calculus provides a notation for specifying *agent expressions* A and forms F. The reduction relation $A \rightarrow B$ means that the agent A reduces in one atomic step to agent B.

Agent expressions can reduce to forms, which are a subset of agents. Intuitively, forms are collections of bindings and services. The following form F contains two bindings $x{\mapsto}F_1$ and $z{\mapsto}F_2$ that are concatenated by extension:

$$F = x{\mapsto}F_1 \cdot z{\mapsto}F_2$$

Abstractions are written as lambda abstractions like $\lambda y.(\mathbf{R} \cdot y; x)$. A sandbox expression $A; B$ denotes an agent B which is evaluated in the environment denoted by A. If B is a variable we say that the sandbox expression is a projection. The body of the abstraction is thus a projection on x in the term $\mathbf{R} \cdot y$ which denotes the current root context $\mathbf{R}$ extended with the formal argument y. An abstraction put into a sandbox where the environment is a form denotes a closure which can be invoked. The environment of the following closure S consists of the form $x{\mapsto}\epsilon$:

$$S = x{\mapsto}\epsilon; \lambda y.(\mathbf{R} \cdot y; x)$$

Applying service S to F denotes the following agent:

$$SF = (x{\mapsto}\epsilon; \lambda y.(\mathbf{R} \cdot y; x))F$$

An application of a closure to a form reduces to a sandbox expression, where the environment is extended with a binding $y{\mapsto}F$, i.e., a binding containing the actual argument. Now SF reduces to:

$$\rightarrow x{\mapsto}\epsilon \cdot y{\mapsto}F; \mathbf{R} \cdot y; x$$

Next, we evaluate $\mathbf{R} \cdot y$ in its environment. We distribute the environment and replace $\mathbf{R}$ with it. The projection $x{\mapsto}\epsilon \cdot y{\mapsto}F; y$ reduces to F. This gives:

$$\rightarrow x{\mapsto}\epsilon \cdot y{\mapsto}F \cdot (x{\mapsto}F_1 \cdot z{\mapsto}F_2); x$$

Finally, the value of x is looked up in the environment. Extension $\cdot$ overwrites bindings with the same labels, thus projection for x finds the binding $x{\mapsto}F_1$:

$$\rightarrow F_1$$

The agent expression SF has reduced to F_1, written $SF \Rightarrow F_1$. We can say that the service S projects on the label x. If, however, the argument to S does not contain a binding for x, then the empty form ϵ is returned as default. Formally this is expressed by $S\epsilon \Rightarrow \epsilon$.

The example we have presented demonstrates how forms act as explicit environments. The other aspect of Piccola are agents and channels for which more examples are given in Section 3.4.

3.2.1 Design Rationale

The semantics of Piccola we will give is based on a structural congruence $\equiv$ and a reduction relation $\rightarrow$ between agents. *Structural rules* are reversible. They define syntactical rearrangements on agent terms. They do not correspond to an actual computation. We have not used structural rules in our introduction example. They define, for instance, how we distribute the environment and reorder forms so that projections can look up values. *Reduction rules* rewrite (and simplify) terms. They correspond to the basic computation steps and are not reversible. This style of giving semantics was used by Milner [Mil91] and is inspired by the Chemical Abstract Machine (CHAM) of Berry and Boudol [BB92].

One of the main difference between Piccola and the π-calculus is the big number of structural rules. The main reason for this is that we not only have parallel composition, but also application, extension, and sandbox expressions in order give a direct semantics for composition abstractions.

When defining the semantics of forms we can either relate expressions by structural congruence or by reduction. As an example, we want to say that extension overwrites bindings with equal labels. This can be specified by the axiom $x{\mapsto}F_1 \cdot x{\mapsto}F_2 \equiv x{\mapsto}F_2$ or by the reduction rule $x{\mapsto}F_1 \cdot x{\mapsto}F_2 \rightarrow x{\mapsto}F_2$. We have chosen to give the semantics of forms as an equational theory where the equations are structural rules.

Our guidelines for using congruence rules and reduction relations are as follows:

- We want the set of forms to be closed under the congruence rules. The congruences define the semantics of a form independent of any reduction semantics and equivalence predicate.

- We want that the congruence rules allow us to rewrite any agent into a canonical form. This canonical form (see Definition 3.11) makes it explicit whether the agent is a barb or not, and how it can reduce.

Once we have defined the syntax and semantics we will give some more examples how computation works in our calculus. Then we will explain what it means to say that two agents are equivalent: two agents are equivalent when no context can ever detect a difference between the two. While this definition is intuitive it is hard to prove that two agents are equivalent. The problem is that the definition includes a quantification over all contexts. In the rest of the chapter we will therefore establish a more technical characterization of how agents may reduce and how this reduction capabilities are preserved when the agent is put into a context. For this characterization we write agents in a canonical representation so that we can formulate a decision process how the agent may reduce. This allow us to characterize stuck agents, i.e., agents that do not reduce further but are not forms. We will also see that beta-reduction is a valid law, i.e., that a beta-reduction step is not a visible reduction.

3.2.2 Syntax

The Piccola calculus is given by agents A, B, C that range over the set of agents $\mathcal{A}$ in Table 3.1. There are two categories of identifiers: labels and channels. The set of labels $\mathcal{L}$ is ranged over by x, y, z. We often use the term variables and labels interchangeably. Specific labels are also written in the *italic* text font. Channels are denoted by $a, b, c, d \in \mathcal{N}$. Labels are bound with bindings and λ-abstractions, and channels are bound by ν-restrictions.

Agent expressions are reduced to static form values or simply *forms*. Forms are ranged over by F, G, H, see Table 3.1. Notice that the set of forms is a subset of all agents. Forms are the first-class citizens of the Piccola calculus, i.e., they are the values that get communicated between agents and are used to invoke services. Forms are collections of bindings and services. The set of forms is denoted by $\mathcal{F}$. Certain forms play the role of services. We use S to range over services. User-defined services are closures. Primitive services are inspect, the bind and hide primitive, and the output service.

In the following, we give an informal description of the different agent expressions and how they reduce.

A, B, C	$::=$	ϵ	*empty form*	$\mid$	$\mathbf{R}$	*current root*
	$\mid$	$A; B$	*sandbox*	$\mid$	x	*variable*
	$\mid$	$x\mapsto$	*bind*	$\mid$	$hide_x$	*hide*
	$\mid$	$\mathbf{L}$	*inspect*	$\mid$	$A \cdot B$	*extension*
	$\mid$	$\lambda x.A$	*abstraction*	$\mid$	AB	*application*
	$\mid$	$\nu c.A$	*restriction*	$\mid$	$A \mid B$	*parallel*
	$\mid$	$c?$	*input*	$\mid$	c	*output*
F, G, H	$::=$	ϵ	*empty form*	$\mid$	S	*service*
	$\mid$	$x\mapsto F$	*binding*	$\mid$	$F \cdot G$	*extension*
S	$::=$	$F; \lambda x.A$	*closure*	$\mid$	$\mathbf{L}$	*inspect*
	$\mid$	$x\mapsto$	*bind*	$\mid$	$hide_x$	*hide*
	$\mid$	c	*output*			

Table 3.1: Agents and Forms

- The *empty form* does not reduce further. It denotes a form without any binding.
- The *current root* agent denotes the current lexical scope. The lexical scope is explicitly set using a sandbox.
- A *sandbox* $A; B$ evaluates the agent B in the root context given by A. Agent B is evaluated in a controlled environment. The sandbox provides all resources needed by B. If B is a variable x, we say that $A; x$ is a *projection* on x in A.
- A *variable* denotes the value bound by the variable in the current root context.
- The primitive service *bind* creates bindings. If A evaluates to F, then $x\mapsto A$ evaluates to the *binding* $x\mapsto F$. The bind primitive $x\mapsto$ is equivalent to the closure $\epsilon; \lambda y.x\mapsto y$, i.e., a service that takes a form y and returns the binding $x\mapsto y$. We have taken $x\mapsto$ as a primitive service instead of the primitive expression $x\mapsto A$ in the grammar for agents. This reduces the number of congruence rules in the next section.
- The primitive service $hide_x$ hides the visibility of bindings. If A evaluates to F with no binding for the label x in F, then $hide_x A$ evaluates to F. If A evaluates to a form with a binding for x, then $hide_x A$ evaluates to a form where this binding is hidden.
- The *inspect* service is the primitive service for inspecting forms. Applying inspect to a form that contains some bindings returns a first-class encoding of some arbitrary label in the form. Inspect is also used to check for the empty form or if a form consists only of a service.
- The values of two agents are concatenated by *extension*. In the value of $A \cdot B$ the bindings of B override those for the same label in A. Extension was originally introduced as polymorphic form extension [Lum99, LAN00] distinguished from binding extension. In the Piccola calculus, both extensions are unified into a single construct.
- An *abstraction* $\lambda x.A$ abstracts x in A.

- *Application* AB denotes the result of applying A to B. Piccola uses a call-by-value reduction order. In order to reduce AB, A must evaluate to a service and B to a form. If A evaluates to the closure $F; \lambda x.A'$ and B evaluates to G, then AB evaluates to the sandbox expression $F \cdot x \mapsto G; A'$, i.e., the agent A' is evaluated in the root context that consists of the form F extended with a binding $x \mapsto G$. If A evaluates to a bind primitive then AB evaluates to a binding.

- The expression $\nu c.A$ *restricts* the visibility of the channel name c to the agent expression A. A is the scope of channel c.

- The *parallel* agent $A \mid B$ evaluates A and B concurrently. The value of $A \mid B$ is the value of agent B. The parallel composition operator is not commutative (c.f. Section 3.1). Evaluating $A \mid B$ spawns off the agent A asynchronously and yields the value of B.

- The agent $c?$ *inputs* a form from channel c and reduces to that value. The reader familiar with the π-calculus will notice a difference with the input prefix. Since we have explicit substitution in our calculus it is simpler to specify the input by $c?$ and use the context to bind the received value instead of defining a prefix syntax $c(X).A$ as in the π-calculus.

- The channel c is a primitive *output* service. If A evaluates to F, then cA evaluates to the *message* cF. The value of a message is the empty form ϵ.

Observe that a single channel c is an output service, i.e., a form. On the other hand, input $c?$ is not a form. This reflects the fact that a channel stands for a primitive sending service. Receiving from a channel is an agent that evaluates to any form sent along the channel.

It should be noted that forms may contain free channel names. This is necessary so that agents can communicate channels. An agent may create a local channel and communicate this channel. One can think of this communication like giving the address of the channel. An agent that receives the address can communicate using this channel.

We define the null agent $\mathbf{0}$ to be the agent $\nu c.c?$. We will see that $\mathbf{0} \not\equiv \epsilon$. The null agent creates a restricted channel c and tries to read from it. Since the channel is restricted no other agent can ever write to it and the null agent is blocked forever.

The *free channels* $fc(A)$ of an agent A are inductively defined in Table 3.2. α-conversion (of channels) is defined in the usual sense. We identify agents expressions up to α-conversion. We allow ourselves some freedom in the use of α-conversion on channels. We furthermore assume that the substitution of channels does not affect bound channels. In a statement, we say that a channel name is *fresh* to mean that it is different from any other name which occurs in the statement.

We omit a definition of *free variables*. Since Piccola is a calculus with explicit environments, we cannot easily define α-conversion on variables. Such a definition would have to include the special nature of $\mathbf{R}$. Instead, we define a *closed agent* where all variables, root expressions, and abstractions occur beneath a sandbox.

Definition 3.1 *The following agents A are closed:*

- $\epsilon, x \mapsto, hide_x, \mathbf{L}, c$ *and* $c?$ *are closed agents.*

- *If* A *and* B *are closed then also* $A \cdot B, AB, A \mid B$ *and* $\nu c.A$.

- *If* A *is closed, then also* $A; B$ *for any agent* B.

$fc(\epsilon) = \emptyset$	$fc(\mathbf{R}) = \emptyset$
$fc(x) = \emptyset$	$fc(\mathbf{L}) = \emptyset$
$fc(x\mapsto) = \emptyset$	$fc(hide_x) = \emptyset$
$fc(A;B) = fc(A) \cup fc(B)$	$fc(A \cdot B) = fc(A) \cup fc(B)$
$fc(\lambda x.A) = fc(A)$	$fc(AB) = fc(A) \cup fc(B)$
$fc(\nu c.A) = fc(A) \backslash \{c\}$	$fc(A \mid B) = fc(A) \cup fc(B)$
$fc(c?) = \{c\}$	$fc(c) = \{c\}$

Table 3.2: Free Channels

Observe that any form F is closed by the above definition. An agent is open if it is not closed. Open agents are $\mathbf{R}$, variables x, abstractions $\lambda x.A$ and compositions thereof. Any agent can be closed when we put it into a sandbox with a closed context. Sandbox agents are closed if the root context is closed. In Lemma 3.4 we show that the property of being closed is maintained by reduction.

The following defines the set of labels of a form.

Definition 3.2 *For each form F, the set of $labels(F) \subset \mathcal{L}$ is given by:*

$$labels(\epsilon) = \emptyset \qquad labels(S) = \emptyset$$
$$labels(x\mapsto G) = \{x\} \qquad labels(F \cdot G) = labels(F) \cup labels(G)$$

3.2.3 Syntactic Conventions

The operators of agent expressions have the following precedence: Application is stronger than extension and associates to the left. Extension is stronger than ν- and λ-abstraction, which are stronger than sandbox. Sandbox is stronger than parallel composition. Extension and sandbox operators are associative thus we omit parentheses. For example:

$$\begin{aligned} x\mapsto y \cdot A; \mathbf{R} \mid B &\text{ is } ((((x\mapsto)y) \cdot A); \mathbf{R}) \mid B \\ \nu c.A \mid B &\text{ is } (\nu c.A) \mid B \\ y\mapsto \lambda x.B; C &\text{ is } (y\mapsto(\lambda x.B)); C \end{aligned}$$

We use a single ν or λ binder to scope several channels or define higher-order abstractions. For instance

$$\nu cd.A \text{ is } \nu c.(\nu d.A)$$

We often write the empty form as () when it is the argument in an application thus we write $F()$ instead of $F\epsilon$.

3.2.4 Structural Congruence

We now define a *structural congruence* $\equiv$ on agents expressions. This approach builds on the Chemical Abstract Machine Ideas of Berry and Boudol [BB92], and the π semantics of Milner [Mil91, Mil92]. The idea is that two agents can be considered equivalent even if they are syntactically different. In the π-calculus this allows us to rewrite expressions to bring communicating partners in juxtaposition.

We extend this idea and define the semantics of operators like sandbox or hide by structural congruence. This corresponds to the algebraic specification approach of using conditional equations [Wir90]. For instance the rule *ext service commute* $S \cdot x{\mapsto}F \equiv x{\mapsto}F \cdot S$ is read as: If there are agents A, B such that A is a service and B is a form then it holds that $A \cdot x{\mapsto}B \equiv x{\mapsto}B \cdot A$.

A congruence relation $\equiv$ obeys the following laws:

$$\begin{aligned}
& & A \equiv A & \qquad \text{(reflexive)}\\
A \equiv B & \quad \text{implies} & B \equiv A & \qquad \text{(symmetric)}\\
A \equiv B \text{ and } B \equiv C & \quad \text{implies} & A \equiv C & \qquad \text{(transitive)}\\
A \equiv B & \quad \text{implies} & \mathcal{C}[A] \equiv \mathcal{C}[B] & \qquad \text{(congruence)}
\end{aligned}$$

We write $\mathcal{C}$ for a context. A context is an agent expression with one or several holes. Filling these holes with an agent A is written $\mathcal{C}[A]$. The rule *congruence* thus says that we can replace any sub-agent with a congruent one.

We present the congruence rules in three groups. The first group deals with congruence on forms. It specifies that extension is idempotent and associative on forms. The second group defines *preforms*. Those are agents expressions that are congruent to a form. The rules in this group define the semantics of certain sandbox expressions and of the extension and hide operators. The last group defines the semantics of parallel composition and of communication, those are the rules for communicating agents.

The set of preforms $\mathcal{F}^{\equiv}$ is a subset of the set of agent expressions $\mathcal{A}$. Some agent expressions are congruent to a form using the $\equiv$ relation. The intention is that the agent A can be restructured to a form F without actually performing a computation. For instance, the agent $hide_x\epsilon$ is equivalent to the empty form ϵ. The set of all preforms is defined by:

$$\mathcal{F}^{\equiv} = \{A \mid \exists F \in \mathcal{F} \text{ with } F \equiv A\} \tag{3.1}$$

Clearly, all forms are preforms.

Form Congruences

This set of congruence rules specify the properties of the extension operator when used to compose forms.

$$\begin{aligned}
& & F \cdot \epsilon &\equiv F & \text{(ext empty right)}\\
& & \epsilon \cdot F &\equiv F & \text{(ext empty left)}\\
& & (F \cdot G) \cdot H &\equiv F \cdot (G \cdot H) & \text{(ext assoc)}\\
& & S \cdot x{\mapsto}F &\equiv x{\mapsto}F \cdot S & \text{(ext service commute)}\\
x \neq y & \quad \text{implies} & x{\mapsto}F \cdot y{\mapsto}G &\equiv y{\mapsto}G \cdot x{\mapsto}F & \text{(ext bind commute)}\\
& & x{\mapsto}F \cdot x{\mapsto}G &\equiv x{\mapsto}G & \text{(single binding)}\\
& & S \cdot S' &\equiv S' & \text{(single service)}
\end{aligned}$$

The first three rules require $\cdot$ to be an associative operator with the empty form as neutral element. The rules *ext service commute* and *ext bind commute* allow us to rearrange the bindings in a form. We use these rules to bring the required binding or service at the end of a form for the reduction rules.

The rules *single service* and *single binding* specify that extension overwrites services and bindings with the same label. Using these form congruences, we can rewrite any form F into one of the following three cases:

$$\begin{aligned} &F \equiv \epsilon \\ &F \equiv S \\ &F \equiv F' \cdot x \mapsto G && \text{where } x \notin \mathit{labels}(F') \end{aligned}$$

This is proven by structural induction over forms. It formalizes our idea that forms are extensible records unified with services. A form has at most one binding for a given label.

Preform Congruences

A sandbox expression $A; B$ denotes an agent B being evaluated in the environment given by the value of A. The following set of rules rewrite sandbox expressions when the environment is evaluated to a form F.

$$\begin{aligned} F; A \cdot B &\equiv (F; A) \cdot (F; B) && \text{(sandbox ext)} \\ F; AB &\equiv (F; A)(F; B) && \text{(sandbox app)} \\ A; (B; C) &\equiv (A; B); C && \text{(sandbox assoc)} \\ F; G &\equiv G && \text{(sandbox value)} \\ F; \mathbf{R} &\equiv F && \text{(sandbox root)} \end{aligned}$$

The sandbox operator is associative and distributive over extension and application. A sandbox expression over a form value is the same as the form value. A sandbox over root yields the sandbox form. Note that these last two rules convert a preform into a form.

The rules distribute the environment. The subagents may be variables which are looked up in F. The reader may note that the rules build on the fact that the evaluation strategy is fixed from left to right. The environment is evaluated to a form F and will thus not enjoy any further reductions. Had we specified:

$$C; A \cdot B \overset{?}{\equiv} (C; A) \cdot (C; B)$$

we could rewrite $S(); A \cdot B$ into $(S(); A) \cdot (S(); B)$ which would mean that the application $S()$ which might cause some side-effects would get invoked twice. With the above rules, such duplications are not possible.

The next set of congruence rules define the semantics of label hiding. Applications of $\mathit{hide}_x F$ are recursively defined over F.

$$\begin{aligned} && \mathit{hide}_x(F \cdot x \mapsto G) &\equiv \mathit{hide}_x F && \text{(hide select)} \\ x \neq y \quad \text{implies} && \mathit{hide}_y(F \cdot x \mapsto G) &\equiv \mathit{hide}_y F \cdot x \mapsto G && \text{(hide over)} \\ && \mathit{hide}_x \epsilon &\equiv \epsilon && \text{(hide empty)} \\ && \mathit{hide}_x S &\equiv S && \text{(hide service)} \end{aligned}$$

Observe that label hiding does not allow us to drop one binding and fetch an "older" binding. Forms do not have a history. The label hiding semantics is the same as in the form calculus [Sch99].

The following rule specifies that the important part of a form is the service when used as a functor. We can drop additional bindings F in this case.

$$(F \cdot S)G \equiv SG \qquad \text{(use service)}$$

Agent Operators

We intend that the result of a concurrent expression be returned from the right-hand side of its topmost parallel composition. This topmost, right-hand side agent is called the *main agent*. Therefore, in contexts expecting a result, parallel composition is not commutative. On the other hand, in contexts immediately to the left of a parallel composition, parallel composition is commutative since its result is discarded. The main agent C can spawn off sub-agents during its lifetime. The final *value* of C is available even when sub-agents A or B are still alive. This semantics has a close relation to concurrent programming languages where a thread may spawn off sub-threads. The rules

$$\begin{aligned} (A \mid B) \mid C &\equiv A \mid (B \mid C) && \text{(par assoc)} \\ (A \mid B) \mid C &\equiv (B \mid A) \mid C && \text{(par left commute)} \end{aligned}$$

state that parallel composition is associative and commutative to the left. This allows us to arbitrarily exchange concurrent agents as long as the main agent is not changed. The rules are similar to the rules given in the calculus $\text{conc}_{\varsigma m}$ of Gordon *et al.* [GH98].

There are a number of structural rules that allow the left component of a parallel composition to migrate freely in the soup of agents. The ability to do so reflects the fact that the value of a parallel composition is specified by its right component. Consider for instance the two rules for extension:

$$\begin{aligned} (A \mid B) \cdot C &\equiv A \mid B \cdot C && \text{(par ext left)} \\ F \cdot (A \mid B) &\equiv A \mid F \cdot B && \text{(par ext right)} \end{aligned}$$

The rule *par ext left* says that the communication capabilities of $(A \mid B) \cdot C$ are that of A and B, no matter the structure of C. However, the symmetric rule *par ext right* has the additional constraint that its left part is evaluated to a form value F. This guarantees left to right evaluation. For example in the term $S() \cdot S'()$ the application $S()$ gets evaluated strictly before $S'()$.

The other *par* rules are:

$$\begin{aligned} (A \mid B)C &\equiv A \mid BC && \text{(par app left)} \\ F(A \mid B) &\equiv A \mid FB && \text{(par app right)} \\ (A \mid B); C &\equiv A \mid (B; C) && \text{(par sandbox left)} \\ F; (A \mid B) &\equiv (F; A) \mid (F; B) && \text{(par sandbox right)} \end{aligned}$$

The rules for distributing application and parallel composition follow the same patterns as the rules for extension. The rule *par sandbox left* exploits the fact that the value of $A \mid B$ is the value of B for sandbox expressions. The rule *par sandbox right* distributes the environment F for both agents A and B. This rule is similar to the rules *sandbox ext* and *sandbox app* we presented in the previous paragraph.

The next rule says that the left part of a parallel composition can be discarded once it is evaluated to a form.

$$F \mid A \;\equiv\; A \tag{discard zombie}$$

This rule is motivated by the fact that a form is a fully evaluated agent. The form F will neither reduce nor communicate anymore.

An output service c applied to a form F is a message. The following rule allows us to rewrite a message as the parallel composition of the message and the empty form as its value:

$$cF \;\equiv\; cF \mid \epsilon \tag{emit}$$

The specification of the empty form as the value of a message (its main agent) is somewhat arbitrary. We could also define the value to be F without changing much.

Rule emit enables messages to move around freely. For instance in

$$\begin{aligned} x{\mapsto}c() &\;\equiv\; x{\mapsto}(c() \mid \epsilon) && \text{by rule } \textit{emit} \\ &\;\equiv\; c() \mid x{\mapsto}\epsilon && \text{by rule } \textit{par ext right} \end{aligned}$$

the message $c()$ escapes the binding.

It is helpful to compare the structure of the rules *discard zombie* and *emit*. Both rules eliminate or introduce a parallel composition. In fact, rule *discard zombie* neutralizes unnecessary applications of rule *emit*. Consider

$$\begin{aligned} cF &\;\equiv\; cF \mid \epsilon && \text{by rule } \textit{emit} \\ &\;\equiv\; (cF \mid \epsilon) \mid \epsilon && \text{by rule } \textit{emit} \\ &\;\equiv\; (\epsilon \mid cF) \mid \epsilon && \text{by rule } \textit{par left commute} \\ &\;\equiv\; cF \mid \epsilon && \text{by rule } \textit{discard zombie} \end{aligned}$$

The rule *discard zombie* eliminates the parallel composition introduced by the second application of rule *emit*.

The rule *commute channels* specifies that the order of channel declarations does not matter. This rule is a classical rule known from the π-calculus.

$$\nu cd.A \;\equiv\; \nu dc.A \tag{commute channels}$$

We also adopt from the π-calculus the *scope extrusion* rules. These rules are necessary in order to communicate channels. The scope extrusion rules have a side condition which ensures that no free channel names are captured when the agent enters a channel scope. For instance in

$$c \notin fc(A) \quad \text{implies} \quad A \mid \nu c.B \;\equiv\; \nu c.(A \mid B) \tag{scope par left}$$

the agent A migrates in and out of the channel scope νc. The side condition ensures that A does not contain free channel names c. This means that A will not use the channel c. Note that we may apply α-conversion and rename c in B to a name not free in A.

Since there are many (asymmetric) operators we also need many scope extrusion rules. For simplicity, we omit the precondition $c \notin fc(A)$ in the following rules:

$$
\begin{aligned}
(\nu c.B) \mid A &\equiv \nu c.(B \mid A) && \text{(scope par right)}\\
(\nu c.B) \cdot A &\equiv \nu c.(B \cdot A) && \text{(scope ext left)}\\
A \cdot \nu c.B &\equiv \nu c.(A \cdot B) && \text{(scope ext right)}\\
A; \nu c.B &\equiv \nu c.(A; B) && \text{(scope sandbox right)}\\
(\nu c.B); A &\equiv \nu c.(B; A) && \text{(scope sandbox left)}\\
(\nu c.B)A &\equiv \nu c.BA && \text{(scope app left)}\\
A(\nu c.B) &\equiv \nu c.AB && \text{(scope app right)}
\end{aligned}
$$

3.2.5 Reduction Relation

We define the reduction relation $\rightarrow$ on agent expressions to reduce applications, communication and projections.

The rule *reduce beta* defines beta reduction. The rule does not substitute G for x in the agent A as in the classical λ-calculus. Instead, it extends the environment in which A is evaluated. This is the beta-reduction rule found in calculi for explicit substitution [ACCL91, Nis00]:

$$(F; \lambda x.A)G \rightarrow F \cdot x \mapsto G; A \qquad \text{(reduce beta)}$$

This rule defines how to reduce an application of the closure $F; \lambda x.A$ with an argument G. The application reduces to a sandbox expression where the agent A is evaluated in the environment $F \cdot x \mapsto G$. The binding $x \mapsto G$ ensures the variable x in A yields G.

The rule *reduce comm* defines communication over a channel:

$$cF \mid c? \rightarrow F \qquad \text{(reduce comm)}$$

The agent $c?$ reduces to the value F if there is a message cF available. Note that the message gets consumed by this reduction.

There is no need for a symmetric variant of the above rule, even though parallel composition is not commutative. Using rule *emit*, rule *par left commute*, and rule *discard zombie* it holds:

$$c? \mid cF \equiv c? \mid cF \mid \epsilon \equiv cF \mid c? \mid \epsilon \rightarrow F \mid \epsilon \equiv \epsilon$$

The value of the parallel composition will be the value of the message cF which is the empty form. The main agent is cF whose value is the empty form.

The following rule removes a projection and replaces it by the value of binding:

$$F \cdot x \mapsto G; x \rightarrow G \qquad \text{(reduce project)}$$

The reader should note that rule *ext bind commute* may be used to bring the necessary binding at the end of the context form in order to trigger rule *reduce project*. For instance:

$$x \mapsto F \cdot y \mapsto G; x \equiv y \mapsto G \cdot x \mapsto F; x \rightarrow F$$

The following are the rules for reducing applications of inspect:

$$\begin{aligned} \mathbf{L}\epsilon &\rightarrow \epsilon; \lambda x.(x; \mathit{isEmpty})\epsilon && \text{(reduce inspect empty)} \\ \mathbf{L}S &\rightarrow \epsilon; \lambda x.(x; \mathit{isService})\epsilon && \text{(reduce inspect service)} \\ \mathbf{L}(F \cdot x{\mapsto}G) &\rightarrow \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_x && \text{(reduce inspect label)} \end{aligned}$$

where $\mathit{label}_x = \mathit{project}{\mapsto}(\epsilon; \lambda x.(x; x)) \cdot \mathit{hide}{\mapsto}\mathit{hide}_x \cdot \mathit{bind}{\mapsto}(x{\mapsto})$.

We use $\mathbf{L}$ to inspect a form F. If F is the empty form, rule *reduce inspect empty* applies. If F consists of a service and has no bindings rule *reduce inspect service* applies. Otherwise F contains at least a binding $x{\mapsto}G$ for x and G appropriate. The value form label_x is a first-class encoding of the label x. A label has three capabilities: to be projected in a form, to hide the visibility, and to create a new binding with this label. These three capabilities are reified by the two primitive services hide_x and $x{\mapsto}$ and the projection service.

$\mathbf{L}$ is not deterministic if the inspected form contains several labels. Let $F = x{\mapsto}\epsilon \cdot y{\mapsto}\epsilon$. It holds $\mathbf{L}F \rightarrow \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_x$ as well as $\mathbf{L}F \rightarrow \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_y$ due to the structural rule *ext bind commute* that allow reordering of bindings. Applications of $\mathbf{L}$ can be found in Section 3.4.4.

In order to reduce subexpressions we use evaluation contexts. An evaluation context $\mathcal{E}$ has a single hole, written $[]$. Filling this hole with an agent expression A denotes the expression $\mathcal{E}[A]$. The evaluation context states where reduction may happen. The evaluation context defines strict evaluation strategy:

Definition 3.3 *An evaluation context is generated by the following grammar:*

$$\begin{array}{rcl|l} \mathcal{E} & ::= & [] & \nu c.\mathcal{E} \\ & | & \mathcal{E} \cdot A & F \cdot \mathcal{E} \\ & | & \mathcal{E}; A & F; \mathcal{E} \\ & | & \mathcal{E} A & F\mathcal{E} \\ & | & A|\mathcal{E} & \mathcal{E}|A \end{array}$$

Notice that two symmetric variants for extension are not the same: $\mathcal{E} \cdot A$ says that we can reduce a term in the left side of an extension no matter the structure of the right-hand side. The variant $F \cdot \mathcal{E}$ states that we only reduce the right-hand side if the left side is a form value, i.e., it is fully reduced. The same applies for sandbox expressions and applications. Reduction is always possible under a parallel composition and channel restriction operator. Note also that $\lambda x.\mathcal{E}$ is not an evaluation context. The body of abstractions does not reduce.

The following rules combine structural congruence and evaluation contexts with reduction. They allow us to reduce subterms, provided they are in a valid evaluation context:

$$\frac{A \equiv A' \quad A' \rightarrow B' \quad B' \equiv B}{A \rightarrow B} \qquad \text{(reduce struct)}$$

$$\frac{A \rightarrow B}{\mathcal{E}[A] \rightarrow \mathcal{E}[B]} \qquad \text{(reduce propagate)}$$

We denote by $\Rightarrow$ the reflexive and transitive closure of $\rightarrow$. This concludes the reduction semantics of the Piccola calculus. The congruence and reduction rules are summarized in Appendix A.

The property of being closed is respected by reduction:

Lemma 3.4 *If A is a closed agent and $A \rightarrow B$ or $A \equiv B$ then B is closed as well.*

Proof. Easily checked by induction over the formal proof for $A \rightarrow B$. □

3.3 Recursive Services

Since Piccola contains lambda abstraction it contains fixed point combinators as they are known from the classical lambda calculus [Bar84]. In order to use a fixed-point combinator to specify mutually recursive services, we have to keep in mind that that reduction is eager. Consider the fixed point combinator **Y**:

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

This combinator has the property that it finds a fixed point for every lambda term f, formally: $\mathbf{Y}f = f(\mathbf{Y}f)$. Since reduction in Piccola is strict, we cannot directly use $\epsilon; \mathbf{Y}$ since this combinator would lead to a nonterminating calculation. Let $S = \epsilon; \lambda x.A$ and $S' = f{\mapsto}S; \lambda x.f(xx)$. It holds that:

$$\begin{aligned}(\epsilon; \mathbf{Y}S) &\Rightarrow S'S' \\ &\rightarrow f{\mapsto}S \cdot x{\mapsto}S'; f(xx) \\ &\Rightarrow f{\mapsto}S \cdot x{\mapsto}S'; f(S'S') \rightarrow \ldots\end{aligned}$$

We add an additional formal argument y to prevent the eager evaluation of S'. Let:

$$\mathit{fix} \overset{\text{def}}{=} \epsilon; \lambda f.(\lambda xy.f(xx)y)(\lambda xy.f(xx)y) \tag{3.2}$$

In order to simplify reading of recursive services, we define services f as:

$$f(y) = A$$

instead of $f = \mathit{fix}(\epsilon; \lambda fy.A)$. The agent A may refer to the arguments y and to f for recursive calls. The root context of the agent A will contain the variables f and y. To see how f works, consider $f_p = \epsilon; \lambda fy.A$. Now, $\mathit{fix} f_p$ reduces to:

$$\begin{aligned}f = \mathit{fix} f_p &\rightarrow f{\mapsto}f_p; (\lambda xy.f(xx)y)(\lambda xy.f(xx)y) \\ &\rightarrow f{\mapsto}f_p \cdot x{\mapsto}G; \lambda y.f(xx)y \;=\; F_p\end{aligned}$$

Let $G = f{\mapsto}f_p; \lambda xy.f(xx)y$. The invocation fH reduces to

$$\begin{aligned}fH = (\mathit{fix} f_p)H &\rightarrow (f{\mapsto}f_p \cdot x{\mapsto}G; \lambda y.f(xx)y)H \\ &\rightarrow f{\mapsto}f_p \cdot x{\mapsto}G \cdot y{\mapsto}H; f(xx)y \qquad \text{by rule } \textit{reduce beta}\end{aligned}$$

Now $xx \Rightarrow GG$ by projecting the variables x. Reducing this by rule *reduce beta* leads to $xx \Rightarrow f{\mapsto}f_p \cdot x{\mapsto}G; \lambda y.f(xx)y = F_p$. Thus

$$fH \Rightarrow f{\mapsto}F_p \cdot y{\mapsto}H; A$$

and $f = \mathit{fix} f_p$ is a fixed point of the abstraction A specified by $f(y) = A$. In Section 5.7 we will use the same combinator to define recursive services within the Piccola language.

3.4 Examples

In order to illustrate the expressiveness of the Piccola calculus, we present a few examples in this section. For this purpose we extend the notions of forms with constants numbers and the corresponding arithmetic operations.

3.4.1 Encoding Booleans

We can encode booleans by services that either project on the labels *true* or *false* depending on the boolean value they are supposed to model.

$$True \overset{\text{def}}{=} \epsilon; \lambda x.(x; true) \tag{3.3}$$

$$False \overset{\text{def}}{=} \epsilon; \lambda x.(x; false) \tag{3.4}$$

Consider now:

$$\begin{aligned}
True(true{\mapsto}1 \cdot false{\mapsto}2) &= (\epsilon; \lambda x.(x; true))(true{\mapsto}1 \cdot false{\mapsto}2) \\
&\rightarrow \epsilon \cdot x{\mapsto}(true{\mapsto}1 \cdot false{\mapsto}2); (x; true) && \text{by rule } \textit{reduce beta} \\
&\equiv (\epsilon \cdot x{\mapsto}(true{\mapsto}1 \cdot false{\mapsto}2); x); true && \text{by rule } \textit{sandbox assoc} \\
&\rightarrow (true{\mapsto}1 \cdot false{\mapsto}2); true && \text{by rule } \textit{reduce project} \\
&\equiv (false{\mapsto}2 \cdot true{\mapsto}1); true && \text{by rule } \textit{ext bind commute} \\
&\rightarrow 1 && \text{by rule } \textit{reduce project}
\end{aligned}$$

Note the swapping of bindings required to project on *true* in the last step. A similar reduction would show $False(true{\mapsto}1 \cdot false{\mapsto}2) \Rightarrow 2$.

One of the key points of forms is that a client can provide *additional* bindings which are ignored when they are not used. For instance we can use *True* and provide an additional binding $notused{\mapsto}F$ for arbitrary form F:

$$\begin{aligned}
&True(true{\mapsto}1 \cdot false{\mapsto}2 \cdot notused{\mapsto}F) \\
&\quad \Rightarrow (true{\mapsto}1 \cdot false{\mapsto}2 \cdot notused{\mapsto}F); true \\
&\quad \equiv (false{\mapsto}2 \cdot true{\mapsto}1 \cdot notused{\mapsto}F); true && \text{by rule } \textit{ext bind commute} \\
&\quad \equiv (false{\mapsto}2 \cdot notused{\mapsto}F \cdot true{\mapsto}1); true && \text{by rule } \textit{ext bind commute} \\
&\quad \rightarrow 1 && \text{by rule } \textit{reduce project}
\end{aligned}$$

Extending forms can also be used to *overwrite* existing bindings. For instance instead of the variable *notused* a client may override *true*:

$$True(true{\mapsto}1 \cdot false{\mapsto}2 \cdot true{\mapsto}3) \Rightarrow 3$$

Instead of overwriting bindings we can also overwrite services in Piccola. The word "overwriting" should not be understood in the object-oriented sense. The reader should keep in mind that forms are immutable. Overwriting means form extension with the same binding.

A conditional expression is encoded as a curried service that takes a boolean and a case form. When invoked, it selects the service in the case form and invokes it:

$$if \overset{\text{def}}{=} \epsilon; \lambda bc.b(true{\mapsto}(c; then) \cdot false{\mapsto}(c; else))\epsilon \tag{3.5}$$

Observe that the definition of *if* uses b, c as variables instead of channels. In examples we often use arbitrary letters for labels where this does not cause confusion. Now consider:

$$\begin{aligned}
&\mathit{if}\ \mathit{True}\ (\mathit{then}{\mapsto}(F;\lambda x.A)\cdot \mathit{else}{\mapsto}(G;\lambda x.B))\\
&\quad\Rightarrow \epsilon\cdot b{\mapsto}T\cdot c{\mapsto}(\mathit{then}{\mapsto}(F;\lambda x.A)\cdot \mathit{else}{\mapsto}(G;\lambda x.B)); b(\mathit{true}{\mapsto}(c;\mathit{then})\cdot \mathit{false}{\mapsto}(c;\mathit{else}))\epsilon\\
&\quad\Rightarrow T(\mathit{true}{\mapsto}(F;\lambda x.A)\cdot \mathit{false}{\mapsto}(G;\lambda x.B))\epsilon\\
&\quad\Rightarrow (F;\lambda x.A)\epsilon\\
&\quad\rightarrow F\cdot x{\mapsto}\epsilon; A && \text{by rule } \mathit{reduce\ beta}
\end{aligned}$$

The expression *if True* has triggered the agent A.

The contract of the conditional service is that it expects the cases bound by labels *then* and *else*. We can relax this contract and provide default services if those bindings are not provided by the client. To do so, we replace in the definition of *if* the sandbox expression $c;\mathit{else}$ with a default service. This service gets triggered when the case form does not contain an *else* binding:

$$\mathit{if}_d \overset{\text{def}}{=} \epsilon;\lambda bc.b(\mathit{true}{\mapsto}(c;\mathit{then})\cdot \mathit{false}{\mapsto}(\mathit{else}{\mapsto}(\lambda x.\epsilon)\cdot c;\mathit{else}))\epsilon \tag{3.6}$$

Now the *false* branch in $\mathit{if}_d\ \mathit{False}(\mathit{then}{\mapsto}(F;\lambda x.A))$ reduces as

$$\begin{aligned}
&\mathit{false}{\mapsto}(\mathit{else}{\mapsto}(\epsilon;\lambda x.\epsilon)\cdot c;\mathit{else})\\
&\quad\Rightarrow \mathit{false}{\mapsto}(\mathit{else}{\mapsto}(\epsilon;\lambda x.\epsilon)\cdot \mathit{then}{\mapsto}(F;\lambda x.A);\mathit{else})\\
&\quad\equiv \mathit{false}{\mapsto}(\mathit{then}{\mapsto}(F;\lambda x.A)\cdot \mathit{else}{\mapsto}(\epsilon;\lambda x.\epsilon);\mathit{else}) && \text{by rule } \mathit{ext\ bind\ commute}\\
&\quad\rightarrow \mathit{false}{\mapsto}(\epsilon;\lambda x.\epsilon) && \text{by rule } \mathit{reduce\ project}
\end{aligned}$$

and $\mathit{if}_d\ \mathit{False}(\mathit{then}{\mapsto}(F;\lambda x.A)) \Rightarrow \epsilon$.

3.4.2 Communication

The following example illustrates the emission of a message aF and shows how this message floats to its receiver $a?$:

$$\begin{aligned}
y{\mapsto}(aF)\cdot x{\mapsto}a? &\equiv y{\mapsto}(aF \mid \epsilon)\cdot x{\mapsto}a? && \text{by rule } \mathit{emit}\\
&\equiv aF \mid y{\mapsto}\epsilon\cdot x{\mapsto}a? && \text{by rule } \mathit{par\ app\ right}\\
&\equiv y{\mapsto}\epsilon\cdot(aF \mid x{\mapsto}a?) && \text{by rule } \mathit{par\ ext\ right}\\
&\equiv y{\mapsto}\epsilon\cdot x{\mapsto}(aF \mid a?) && \text{by rule } \mathit{par\ app\ right}
\end{aligned}$$

By rule *reduce comm* it holds that $aF \mid a? \rightarrow F$ and with the evaluation context $y{\mapsto}\epsilon\cdot x{\mapsto}[]$ and rule *reduce propagate*

$$\rightarrow y{\mapsto}\epsilon\cdot x{\mapsto}F$$

When we change the order of the bindings in the agent, the term will not reduce since the message aF cannot float to the receiver:

$$x{\mapsto}a?\cdot y{\mapsto}aF \not\rightarrow x{\mapsto}F\cdot y{\mapsto}\epsilon$$

The binding $x \mapsto a?$ blocks the extension. There is no evaluation context that would permit the reduction of aF beneath $x \mapsto a? \cdot []$.

The above example is independent from the extension operator. For instance if the extension was a sandbox operator we have:

$$\begin{aligned} y \mapsto aF; x \mapsto a? &\rightarrow x \mapsto F \\ x \mapsto a?; y \mapsto aF &\not\rightarrow y \mapsto \epsilon \end{aligned}$$

If the two expressions where composed in parallel, then reduction is of course possible:

$$\begin{aligned} &y \mapsto aF \mid x \mapsto a? \rightarrow x \mapsto F \\ &x \mapsto a? \mid y \mapsto aF \rightarrow x \mapsto F \mid y \mapsto \epsilon \equiv y \mapsto \epsilon \end{aligned}$$

The last congruence is due to rule *discard zombie*.

3.4.3 Replication

Even though the Piccola calculus has no explicit replication operator, it is enough to write agents with infinite behaviour. Consider the service $s = F; \lambda x.A \mid xx$. It holds:

$$\begin{aligned} ss &\rightarrow F \cdot x \mapsto s; (A \mid xx) && \text{by rule } \textit{reduce beta} \\ &\equiv (F \cdot x \mapsto s; A) \mid (F \cdot x \mapsto s; xx) && \text{by rule } \textit{par sandbox right} \\ &\equiv (F \cdot x \mapsto s; A) \mid (F \cdot x \mapsto s; x)(F \cdot x \mapsto s; x) && \text{by rule } \textit{sandbox app} \\ &\rightarrow (F \cdot x \mapsto s; A) \mid s(F \cdot x \mapsto s; x) && \text{by rule } \textit{reduce project} \\ &\rightarrow (F \cdot x \mapsto s; A) \mid ss && \text{by rule } \textit{reduce project} \\ &\Rightarrow (F \cdot x \mapsto s; A) \mid (F \cdot x \mapsto s; A) \mid ss \\ &\Rightarrow \cdots \end{aligned}$$

which means that the reduction will lead to infinitely many instantiations of $(F \cdot x \mapsto s; A)$. If the agent A was another sandbox expression $G; A'$ then we would have:

$$\begin{aligned} F \cdot x \mapsto s; A &= F \cdot x \mapsto s; (G; A') \\ &\equiv (F \cdot x \mapsto s; G); A' && \text{by rule } \textit{sandbox assoc} \\ &\equiv G; A' && \text{by rule } \textit{sandbox value} \end{aligned}$$

thus

$$ss \Rightarrow G; A' \mid ss \Rightarrow G; A' \mid G; A' \mid ss \rightarrow \cdots$$

The term ss reduces to infinitely many instances of $G; A'$. In the π-calculus there is a special operator $!P$ to create arbitrary many instances of a process P. In Piccola, infinitely many instantiations of an agent can be programmed and do not have to be made primitive. In the higher-order π-calculus replication can be modeled in a similar way [San01].

3.4.4 Form Inspection

This section demonstrates the use of **L** to inspect a given form. We show how to check if a form is the empty form. Furthermore we can decide if a form contains a given label. We extend the encoding of first-class labels to contain a generic *exists* service. Last but not least, we use **L** to iterate over all the labels of a form.

The service *isEmpty* returns true if the argument is the empty form and *isPlainService* returns true if the argument is a service. A form is a plain service if it is a service not extended with any bindings. In order to simplify reading we also define *aTrue* and *aFalse* which are services that return true and false respectively (see Definitions 3.3 and 3.4).

$$\mathit{aTrue} \overset{\text{def}}{=} \epsilon; \lambda x.\mathit{True} \tag{3.7}$$

$$\mathit{aFalse} \overset{\text{def}}{=} \epsilon; \lambda x.\mathit{False} \tag{3.8}$$

$$\mathit{isEmpty} \overset{\text{def}}{=} \epsilon; \lambda x.\mathbf{L}x(\mathit{isEmpty}{\mapsto}\mathit{aTrue} \cdot \mathit{isService}{\mapsto}\mathit{aFalse} \cdot \mathit{isLabel}{\mapsto}\mathit{aFalse}) \tag{3.9}$$

$$\mathit{isPlainService} \overset{\text{def}}{=} \epsilon; \lambda x.\mathbf{L}x(\mathit{isEmpty}{\mapsto}\mathit{aFalse} \cdot \mathit{isService}{\mapsto}\mathit{aTrue} \cdot \mathit{isLabel}{\mapsto}\mathit{aFalse}) \tag{3.10}$$

We can now decide if a form F contains a binding for label x. This is the case, when $x{\mapsto}\epsilon \cdot F; x$ is the empty form and $x{\mapsto}\mathbf{L} \cdot F; x$ is a plain service. The idea is to provide different default bindings for x and check if projection yields these defaults. We use **L** as a prototype for a plain service.

$$\begin{aligned}
\mathit{exists}_x \overset{\text{def}}{=} \epsilon; \lambda f.\mathit{if}(&\mathit{isEmpty}(x{\mapsto}\epsilon \cdot f; x))\\
&(\mathit{then}{\mapsto}(\lambda y.\mathit{if}(\mathit{isPlainService}(x{\mapsto}\mathbf{L} \cdot f; x))\\
&\quad(\mathit{then}{\mapsto}\mathit{aFalse} \cdot \mathit{else}{\mapsto}\mathit{aTrue}))\cdot\\
&\mathit{else}{\mapsto}\mathit{aTrue})
\end{aligned}$$

It holds that $\mathit{exists}_x(F) \Rightarrow \mathit{True}$ if $x \in \mathit{labels}(F)$ and $\mathit{exists}_x(F) \Rightarrow \mathit{False}$ otherwise.

Instead of having a single service exists_x we prefer to extend the notion of first-class labels with an exists predicate. The following service *label* extracts a label and returns a first class label bound in its argument. If the argument is a plain service or the empty form, the empty form is returned:

$$\begin{aligned}
\mathit{label} \overset{\text{def}}{=} \epsilon; \lambda f.&\mathbf{L}f\\
&\mathit{isEmpty}{\mapsto}(\lambda x.\epsilon)\cdot\\
&\mathit{isService}{\mapsto}(\lambda x.\epsilon)\cdot\\
&\mathit{isLabel}{\mapsto}(\lambda l.\\
&\quad l\cdot\\
&\quad \mathit{exists}{\mapsto}(\lambda f.\mathit{if}(\mathit{isEmpty}((l;\mathit{project})((l;\mathit{bind})\epsilon \cdot f)))\\
&\qquad (\mathit{then}{\mapsto}(\lambda y.\mathit{if}(\mathit{isPlainService}(l;\mathit{project})((l;\mathit{bind})\mathbf{L} \cdot f)))\\
&\qquad\quad (\mathit{then}{\mapsto}\mathit{aTrue}\\
&\qquad\quad \mathit{else}{\mapsto}\mathit{aFalse})\cdot\\
&\qquad \mathit{else}{\mapsto}\mathit{aFalse}))))
\end{aligned}$$

We extend the first-class label l in the *isLabel* service with the *exists* service. We use l in the *exists* service. For instance in the expression $x{\mapsto}\epsilon \cdot f; x$ that is passed to *isEmpty*, the binding $x{\mapsto}\epsilon$ gets replaced by $(l;\mathit{bind})\epsilon$ and the projection $A; x$ is replaced by an application $(l;\mathit{project})A$. Thus $(x{\mapsto}\epsilon \cdot f); x$ becomes $(l;\mathit{project})((l;\mathit{bind})\epsilon \cdot f)$.

The term $\mathit{label}(x{\mapsto}\epsilon)$ denotes a form reifying x with services *bind*, *project*, *hide* and *exists*.

The following service iterates over all the bindings of a form. It uses inspect as its conditional statement. If the inspected form is the empty form or a service, it is returned. If the inspected form contains a binding, a recursive call is performed with the binding hidden. The resulting form extended with the binding itself. Thus, *visit* $F \Rightarrow F$.

$$\begin{aligned} \mathit{visit}(f) \overset{\text{def}}{=} \mathbf{L}f(&\mathit{isEmpty}{\mapsto}(\lambda x.f)\cdot \\ &\mathit{isService}{\mapsto}(\lambda x.f)\cdot \\ &\mathit{isLabel}{\mapsto}(\lambda l.\mathit{visit}((l;\mathit{hide})f)\cdot(l;\mathit{bind})((l;\mathit{project})f))) \end{aligned}$$

In order to make uniform modification to a form F, we adapt the service *visit*. For instance, the following service "replaces" all the bindings $x{\mapsto}F$ with a binding to the empty form $x{\mapsto}\epsilon$ and removes the service.

$$\begin{aligned} \mathit{nullify}(f) \overset{\text{def}}{=} \mathbf{L}f(&\mathit{isEmpty}{\mapsto}(\lambda x.\epsilon)\cdot \\ &\mathit{isService}{\mapsto}(\lambda x.\epsilon)\cdot \\ &\mathit{isLabel}{\mapsto}(\lambda l.\mathit{nullify}((l;\mathit{hide})f)\cdot(l;\mathit{bind})\epsilon)) \end{aligned}$$

For instance it holds that $\mathit{nullify}(\mathbf{L}\cdot x{\mapsto}G) \Rightarrow x{\mapsto}\epsilon$.

3.5 Equivalence for Agents

In this section we define an equivalence relation for agents. The reduction relation we have presented is very intensional. It does not give us a feeling how two different agents might reduce. It keeps the algebraic structure of agent terms. We can abstract from the internal states of agents by quotienting by an *operational congruence*.

Recall the last example of the previous section. We want to say that $\mathit{nullify}(\mathbf{L}\cdot x{\mapsto}G)$ and $x{\mapsto}\epsilon$ are the same which allows us to replace in a program the first expression with the second. Two agents are equivalent if they exhibit the same behaviour, i.e., they enjoy the same reductions. Milner and Sangiorgi have defined the notion of *barbed bisimulation* [MS92]. The idea is that an agent A is barbed similar to B if A can exhibit any reduction that B does and if B is a barb, then A is a barb, too. If A and B are similar to each other they are bisimilar. The advantage of this bisimulation is that it can be readily be given for any calculus that contains barbs or values.

For the asynchronous π-calculus, barbs are usually defined as having the capability of doing an output on a channel. Since Piccola has the notion of forms we extend them to barbs by including parallel composition and close them by channel restriction and the congruence relation. This is due to the fact that an agent, while reducing to a form value, may spawn new subagents that are still active when the main agent is already fully evaluated to a form.

Definition 3.5 *A* barb V *is an agent expression A that is congruent to an agent generated by the the following grammar:*

$$V ::= F \mid A|V \mid \nu c.V$$

We write $A\downarrow$ for the fact that A is a barb, and $A\Downarrow$ when a barb V exists such that $A \Rightarrow V$.

The following lemma relates forms, barbs and agents:

Lemma 3.6 *The following inclusion holds and is strict:*

$$\mathcal{F} \subset \mathcal{F}^{\equiv} \subset \{A | A \downarrow\} \subset \mathcal{A}$$

Proof. The inclusions hold by definition. To see that the inclusion are strict, consider the empty form ϵ, the agent $hide_x\epsilon$, the barb $\mathbf{0} \mid hide_x\epsilon$ and the agent $\mathbf{0}$. $\square$

Definition 3.5 does not give a syntactical characterization of barbs. Instead, it is enough that the agent A is structural congruent to a syntactically characterized term. The following lemma gives a syntactical characterization of barbs.

Lemma 3.7 *For any form F, agent A, and label x, the following terms are barbs, given V_1 and V_2 are barbs.*

$$\begin{array}{ll} V_1 \cdot V_2 & \nu c.V_1 \\ V_1 ; V_2 & A \mid V_1 \\ x{\mapsto}V_1 & \end{array}$$

Proof. By definition we have $V \equiv \nu\tilde{c}.A \mid F$. The claim follows by induction over F. $\square$

We now define barbed bisimulation and the induced congruence:

Definition 3.8 *A relation $\mathcal{R}$ is a* (weak) barbed bisimulation*, if $A\ \mathcal{R}\ B$, i.e., $(A, B) \in \mathcal{R}$ implies:*

- *If $A \longrightarrow A'$ then there exists an agent B' with $B \Rightarrow B'$ and $A'\ \mathcal{R}\ B'$.*
- *If $B \longrightarrow B'$ then there exists an agent A' with $A \Rightarrow A'$ and $A'\ \mathcal{R}\ B'$.*
- *If $A\downarrow$ then $B\Downarrow$.*
- *If $B\downarrow$ then $A\Downarrow$.*

Two agents are (weakly) barbed bisimilar, written $A \approx B$, if there is some (weak) barbed bisimulation $\mathcal{R}$ with $A\ \mathcal{R}\ B$. Two agents are (weakly) barbed congruent, written $A \approx B$, if for all contexts $\mathcal{C}$ we have $\mathcal{C}[A] \approx \mathcal{C}[B]$.

We define behavioural equality using the notion of barbed congruence. As usual we can define strong and weak versions of barbed bisimulation. The strong versions are obtained in the standard way by replacing $\Rightarrow$ with $\longrightarrow$ and $\Downarrow$ with $\downarrow$ in Definition 3.8. We only concentrate on the weak case since it abstracts internal computation.

The following diagram may help the reader to understand the concept of similarity. If A and B are barbed bisimilar and A reduces to A' then we know that there exists a B' and we can complete the diagram

$$\begin{tikzcd} A \arrow[r, phantom, "\approx"] \arrow[d] & B \\ A' & \end{tikzcd} \quad \text{to} \quad \begin{tikzcd} A \arrow[r, phantom, "\approx"] \arrow[d] & B \arrow[d, Rightarrow] \\ A' \arrow[r, phantom, "\approx"] & B' \end{tikzcd}$$

Barbed bisimulation per se cannot serve as a congruence since it is not preserved by parallel composition. For instance, $\mathbf{0} \approx a?$ and $a\epsilon \approx \mathbf{0}$ since neither expression reduces and neither is a barb. But $a\epsilon \mid a? \longrightarrow \epsilon$. The discriminating power of the barbed congruence comes from the fact that we quantify over all contexts. Intuitively, this means that we cannot distinguish between A and B by putting them into any context $\mathcal{C}$, since $\mathcal{C}[A]$ and $\mathcal{C}[B]$ behave the same. We cannot *program* a context that would be able to distinguish between A and B.

Definition 3.8 does not give us much intuition on what expressions are barbed bisimilar. Clearly, congruent terms are barbed congruent, since congruent terms cannot be distinguished by definition. In order to show that something is not barbed congruent we have to give a discriminating context. For example:

$$x{\mapsto}\epsilon \not\approx y{\mapsto}\epsilon \tag{3.11}$$

$$\epsilon \not\approx \mathbf{0} \tag{3.12}$$

The context $[]; x$ distinguishes the first two agents. It holds that $(x{\mapsto}\epsilon); x \rightarrow \epsilon$ but $(y{\mapsto}\epsilon); x \not\rightarrow A$ for all agents A. For the second example it is enough to see that both agents do not reduce further, but only the empty form is a barb.

3.6 Erroneous Reductions

Not all agents reduce to forms. Some agents enjoy an infinite reduction like the agent *ss* in Section 3.4.3. Other agents are stuck. An agent is stuck if it is not a barb and cannot reduce further.

Definition 3.9 *An agent A is* stuck, *written $A\uparrow$, if A is not a barb and there is no agent B such that $A \rightarrow B$.*

Clearly it holds that $\mathbf{0}\uparrow$ and $\mathbf{R}\uparrow$. The property of being stuck is not compositional. For instance $c?\uparrow$ but obviously, $c() \mid c?$ can reduce to ϵ. We can put $\mathbf{R}$ into a context so that it becomes a barb, for instance $F; \mathbf{R} \equiv F$. Note that if an agent is stuck it is not a preform: $\mathcal{F}^{\equiv} \cap \{A | A\uparrow\} = \emptyset$ by definition.

One might argue that the null agent is stuck by intention. But there are other stuck agents where we prefer to say that an error appeared. Those agents are sandbox expressions where the variable is not defined and applications where the functor does not actually reduce to a service, or — more precisely — to a form containing a service. Proposition 3.15 in the following section contains a complete characterization of stuck agents.

Piccola considers illegal projections and application as runtime errors. It is possible to avoid those errors. With first-class labels and *exists* it is possible to detect unbound variables. We can introduce run-time checks by masking every variable x with a check if x is bound in the root context. If x is not bound we can signal an exception. For instance, if we replace all x by

$$\mathit{if}((\mathit{label}(x{\mapsto}\epsilon); \mathit{exists})\mathbf{R})(\mathit{then}{\mapsto}(\lambda y.x) \cdot \mathit{else}{\mapsto}(\lambda y.\mathbf{0})) \tag{3.13}$$

we block using the null agent when an x is not defined in the root context.

Detecting illegal service application is simpler. We replace any application AB by

$$((\epsilon; \lambda y.\mathbf{0}) \cdot A)B \tag{3.14}$$

and this ensures that the functor contains a service. Invoking the default service blocks, thus the original application AB blocks unless A contains a service.

Of course it makes more sense to call a global error handler in case an illegal projection is performed or a form without a service is used as a functor than blocking with the null-agent.

Instead of replacing all occurrences of variables and applications, we can use a type system to infer knowledge about the current root. With a type system, we could reduce the number of actual run-time type checks needed. However, we have not further explored this idea.

3.7 Canonical Terms

In this section we define a canonical representation for agent expressions. This representation allows us to identify where reduction happens. In general it is not obvious how an agent A can reduce since we have to consider all agents B congruent to A and see if we can write B as the composition of a evaluation context and a reduction head. We have to check all possibilities to write A as the composition of a evaluation context and a left-hand side of one of the reduction rules. The canonical representation helps us discover all possible reductions of an agent expression.

The main result of this section is that we can write any agent A in the following format:

$$A \equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k)$$

where all the M_i are messages and all the agents A_j are agents that do not contain parallel composition operators nor channel restrictions except in locations, where they are not *visible*, i.e., beneath a closure. For any agent A_j for $j < k$ there is exactly one reduction possible. The main agent A_k can reduce like any A_j or it is a form. In that case the agent A is a barb, $A \downarrow$.

In order to make the possible reduction of A_j more explicit, we introduce the notion of *threads* and *thread contexts*.

A thread context $\widehat{\mathcal{E}}$ is a evaluation context where parallel composition and restriction appears only to the right of a thread context. A thread context can be understood as the context of a single thread. The thread context exposes the only possible next reduction.

Definition 3.10 *A* thread context $\widehat{\mathcal{E}}$ *is generated by the following grammar:*

$$\widehat{\mathcal{E}} ::= [] \mid \widehat{\mathcal{E}} \cdot A \mid F \cdot \widehat{\mathcal{E}} \mid \widehat{\mathcal{E}}; A \mid \widehat{\mathcal{E}} A \mid F \widehat{\mathcal{E}}$$

Examples of thread contexts are $x \mapsto []$ or $x \mapsto \epsilon \cdot []$. In contrast, $x \mapsto c? \cdot []$ is not a thread context, since $c?$ and thus $x \mapsto c?$ are not forms.

Definition 3.11 (Thread) *An agent A is a thread if there is a thread context $\widehat{\mathcal{E}}$ and A can be written as one of the following:*

1. *an open thread $\widehat{\mathcal{E}}[\mathbf{R}]$, $\widehat{\mathcal{E}}[x]$, or $\widehat{\mathcal{E}}[\lambda x.A]$,*
2. *an application $\widehat{\mathcal{E}}[GH]$ where G is either a closure, $\mathbf{L}$, or a form without a service,*
3. *a projection $\widehat{\mathcal{E}}[F; x]$, or*
4. *a receiver $\widehat{\mathcal{E}}[c?]$ or $\widehat{\mathcal{E}}[F; c?]$.*

If we can write an agent as a thread, we immediately see what the next step of the agent can be. For instance the agent $A = \widehat{\mathcal{E}}[SF]$ is going to beta reduce and the agent $\widehat{\mathcal{E}}[c?]$ is waiting to fetch a message from channel c. This will be formalized in Proposition 3.15.

Notice the side condition for case 2 which requires that the functor of the application may not be a output service, a label hide or bind service. The semantics of these applications is captured by congruence rules. If the functor is, for instance, an output service we apply rule *emit* and cannot determine the possible next reduction of the thread.

Note that $F; \widehat{\mathcal{E}}$ is not a thread context. Consider for instance the context $\widehat{\mathcal{E}} = (F; [])(A \mid B)$ and a stuck agent $\mathbf{R}$. It holds $\widehat{\mathcal{E}}[\mathbf{R}] \equiv A \mid FB$ and we have no control over the possible reductions of A and thus of $\widehat{\mathcal{E}}[\mathbf{R}]$. The following lemma says that a thread context does not contain any internal reductions nor can it enable a reduction to a stuck agent.

Lemma 3.12 *If $A\!\uparrow$ then $\widehat{\mathcal{E}}[A]\!\uparrow$.*

Proof. This is proved inductively over the grammar of thread contexts. We only consider the case for bindings. Assume $A\!\uparrow$. We have to show that also $(x \mapsto A)\!\uparrow$. Obviously we cannot use rule *reduce propagate* to reduce $x \mapsto A$ since A is stuck. We also cannot use any of the struct rules to rewrite $x \mapsto A$. This is due to the fact that if $A = A_1 \mid A_2$ then both $A_i\!\uparrow$. □

In general the property of being stuck is not compositional. The lemma says that this property is preserved by thread contexts. The converse of this lemma can be used to infer reduction. If $\widehat{\mathcal{E}}[A] \rightarrow B$ then we know that there is an agent C such that $A \rightarrow C$ and $B \equiv \widehat{\mathcal{E}}[C]$. Furthermore, and more important, if we have a thread $\widehat{\mathcal{E}}[A]$ where the agent A is one of the cases in Definition 3.11, then there is exactly one possible reduction A and thus $\widehat{\mathcal{E}}[A]$ can participate.

Every agent can be written in the following *canonical* form. We will use this fact to characterize the possible reductions of an agent.

Proposition 3.13 *Each agent A is congruent to a canonical agent:*

$$A \equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k)$$

for messages M_i and threads A_j for $0 \leq i \leq m, 0 \leq j < k$ and A_k is a thread or a form for $k \geq 1$. Furthermore, if A is closed then all A_j are either application, projection, or receiver threads.

An agent A *has a thread* B if A can be written as $\nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k)$ and one of the $A_j = B$. An agent *contains a message* M if $M = M_i$ for some i.

The proof for this proposition is rather long but not difficult. It requires a technical lemma that restricts the claim of the proposition to closed agents. The advantage of closed agents is that the environment is available. For such agents we can do an inductive proof over A and bring the agent into the canonical form.

Lemma 3.14 *Every closed agent is congruent to a canonical term where all the threads are either application, projection, or receiver threads.*

The property that every agent of the form $F; A$ can be written as a canonical agent is used for the sandbox agent $A; B$ for an inductive proof over the structure of all agents which proves the proposition. We recursively convert both A and B into canonical agents and compose the canonical agents. The difficult case is when A is a barb. In this case the composition leads to a term $F; \widehat{\mathcal{E}}[...]$ which is handled by the lemma. The technical details are in Appendix B.

The following proposition formalizes our intuition of threads. Basically, an agent reduces if it contains a thread that is an application or it contains a thread that is a receiver on a channel where a message is available.

Proposition 3.15 *For any agent A it holds that $A \rightarrow B$ if and only if one of the following is true:*

- *A contains a thread $\widehat{\mathcal{E}}[GH]$ and G is either a closure, or the primitive inspect or project service, or*
- *A contains a thread $\widehat{\mathcal{E}}[c?]$ or $\widehat{\mathcal{E}}[F; c?]$ and a message cG for any channel c.*

Proof. We assume that $A \rightarrow B$. From this it follows that that A must contain a thread $\widehat{\mathcal{E}}[A']$ and A' is an application or a receiver. We only have to see that the message can float through the thread context of the receiver by the appropriate parallel congruence rules.

For the other direction, assume that A consists only of threads $\widehat{\mathcal{E}}[\mathbf{R}], \widehat{\mathcal{E}}[\lambda x.B], \widehat{\mathcal{E}}[GF]$ with G has no service, or $\widehat{\mathcal{E}}[c?], \widehat{\mathcal{E}}[F; c?]$ and A does not contain a message cG. By Lemma 3.12 the agent A is then the parallel composition of stuck threads and the main agent and thus $A\uparrow$ or the main agent of A is a form. In both cases, A cannot evolve anymore and thus $A \not\rightarrow B$. □

3.8 Proving the Beta Equivalence

In this section we show that reduction induced by rule *reduce beta* is in fact a valid law, i.e., that is:

$$(F; \lambda x.A)\; G \approx F \cdot x{\mapsto}G; A \tag{3.15}$$

This proof is interesting for two reasons. First, it is an application of the canonical representation established in the previous section. Second, it shows that the reduction induced by rule *reduce beta* is transparent. In general it is not the case that reduction induction is transparent. Consider the reduction $c\epsilon \mid c? \rightarrow \epsilon$ but the terms are not congruent: $c\epsilon \mid c? \not\approx \epsilon$. A discriminating context is $c(x{\mapsto}\epsilon) \mid [\,]; x$ might reduce to the empty form if we receive $x{\mapsto}\epsilon$ from c and gets stuck if we receive the empty form.

Given that law 3.15 holds we can replace any application $(F; \lambda x.A)G$ with the above sandbox expression. In order to prove the law we have to give a barbed bisimulation and show that it is closed for all contexts. To reason about all context we first have to establish some properties about contexts and barbs and their possible interaction.

The definition of barbs is extended to contexts:

Definition 3.16 *A context $\mathcal{C}$ is a barb, written $\mathcal{C}\downarrow$, if $\mathcal{C}[\mathbf{0}]\downarrow$.*

Clearly, $\mathcal{C}\downarrow$ implies $\mathcal{C}[A]\downarrow$ for all agents A, since all the holes in $\mathcal{C}$ are either in the left component of a parallel composition or inside an abstraction.

Lemma 3.17 *$\mathcal{C}[(F; \lambda x.A)G]\downarrow$ implies that $\mathcal{C}\downarrow$ and $\mathcal{C}$ is not an evaluation context.*

Proof. Assume $\mathcal{C}[\mathbf{0}] \not\downarrow$. In that case $\mathcal{C}[(F; \lambda x.A)G]$ would reduce as well which contradicts the assumption. If $\mathcal{C}$ was an evaluation context we could reduce $\mathcal{C}[(F; \lambda x.A)G]$. □

The next lemma states that a redex SG that appears in a context $\mathcal{C}$ either gets reduced in one step, or that the context reduces on its own. In the first case, $\mathcal{C}$ is an evaluation context. In the latter case however, the reduction does not depend on the fact that there is a subterm SG. We may replace it by any agent D and are still able to reduce it further.

Lemma 3.18 *Assume a context $\mathcal{C}$ with $\mathcal{C}[(F; \lambda x.A)G] \to B$. Then exactly one of the following holds:*

- *The reduction comes from the* beta reduction. *That is $B \equiv \mathcal{C}[F \cdot x{\mapsto}G; A]$.*
- *The reduction comes from the surrounding* context. *In that case there exists a context $\mathcal{C}'$ and $B \equiv \mathcal{C}'[(F; \lambda x.A)G]$. Furthermore for all agents D it holds: $\mathcal{C}[D] \to \mathcal{C}'[D]$.*

Proof. Let $S = F; \lambda x.A$. Consider the prooftree that infers the reduction $\mathcal{C}[SG] \to B$. By rule *reduce struct* and Proposition 3.13 we can also derive:

$$\frac{\mathcal{C}[SG] \equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k) \qquad \dfrac{\cdots}{\mathcal{C}[SG] \to B}}{\nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k) \to B}$$

Now there are two cases, either there is a thread $A_j = \widehat{\mathcal{E}}[SG]$ or the application is part of a thread where it is not enabled, $A_j = \mathcal{C}_1[SG] = \widehat{\mathcal{E}}[A']$ where $SG \neq A'$.

In the first case, it might be that A_j reduces and the reduction comes from the beta reduction. If another thread than A_j reduces or we have the second case, then the reduction comes from the surrounding context. The interesting case is when $A_j = \mathcal{C}_1[SG]$ reduces. We have $\widehat{\mathcal{E}}A' \to \widehat{\mathcal{E}}A''$ and the application SG is still visible. □

Observe that the main work for proving the previous lemma has been done in Section 3.7. Now we are ready to prove the main result of this section.

Proposition 3.19 *For any form F, G, agent A, and label x it holds that:*

$$(F; \lambda x.A)G \approx F \cdot x{\mapsto}G; A$$

Proof. Let $\mathcal{R}$ be the relation defined by:

$$\mathcal{R} = Id \cup \{(\mathcal{C}[(F; \lambda x.A)G], \mathcal{C}[F \cdot x{\mapsto}G; A]) \mid \text{for all } \mathcal{C}, F, G, A\}$$

where Id is the identity relation.

Now let $S = F; \lambda x.A$, $A_1 = \mathcal{C}[SG]$ and $A_2 = \mathcal{C}[F \cdot x{\mapsto}G; A]$. We have to consider 4 cases:

- Assume $A_1 \to A_1'$. We have to show that there exists a A_2' with $A_2 \Rightarrow A_2'$ and $(A_1', A_2') \in \mathcal{R}$. By Lemma 3.18 there are two possibilities for the reduction $A_1 \to A_1'$. If the reduction comes from the beta reduction, we have $A_2' = A_2$. If, on the other hand, the reduction comes from the context, there exists a context $\mathcal{D}$ and we have $\mathcal{C}[D] \to \mathcal{D}[D]$ for any agent D and thus $A_2' = \mathcal{D}[F \cdot x{\mapsto}G; A]$.
- Assume $A_2 \to A_2'$. We have to show that there exists a A_1' with $A_1 \Rightarrow A_1'$. If $\mathcal{C}$ is an evaluation context we have $A_1 \to A_2 \to A_2'$. If the reduction comes from the context by Lemma 3.18 there exists a context $\mathcal{D}$ and $A_1 \to \mathcal{D}[SG]$.
- Assume $A_1 \downarrow$. We have to show that $A_2 \Downarrow$. By Lemma 3.17 $\mathcal{C}[SG] \downarrow$ implies $\mathcal{C} \downarrow$ and thus also $A_2 \downarrow$.
- Assume $A_2 \downarrow$. Similar to the above. If $\mathcal{C} \downarrow$ then $A_1 \downarrow$ holds trivially. Otherwise, $\mathcal{C}$ is an evaluation context and we have $A_1 \to A_2 \downarrow$.

This shows that $\mathcal{R}$ is a barbed bisimulation. Since $\mathcal{R}$ is closed for all context this proves barbed equivalence. □

We have shown that beta reduction is a valid law. There are other reductions that generate laws, like projections or inspecting the empty form, a service, or a form with a single binding:

$$
\begin{aligned}
F \cdot x{\mapsto}G; x &\approx G && (3.16)\\
\mathbf{L}\epsilon &\approx \epsilon; \lambda x.(x; \mathit{isEmpty})\epsilon && (3.17)\\
\mathbf{L}S &\approx \epsilon; \lambda x.(x; \mathit{isService})\epsilon && (3.18)\\
\mathbf{L}(x{\mapsto}G) &\approx \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_x && (3.19)
\end{aligned}
$$

We conjecture that similar proofs like the one presented in this section can be given to show these equivalences.

Below are some other laws that hold in the Piccola calculus.

$$
\begin{aligned}
\nu c.(cF \mid \widehat{\mathcal{E}}[c?]) &\approx \nu c.\widehat{\mathcal{E}}[F] && (3.20)\\
A; \mathbf{R} &\approx A && (3.21)\\
F; c? &\approx c? && (3.22)\\
A \cdot (B \cdot C) &\approx (A \cdot B) \cdot C && (3.23)\\
A; FB &\approx F(A; B) && (3.24)
\end{aligned}
$$

Law 3.20 is the deterministic communication law. Since there is at most one message cF available this will be the message consumed by $c?$. The remaining laws remove unnecessary sandbox expressions. Instead of proving these laws directly we present a faithful encoding of the Piccola calculus in the $L\pi$-calculus [San00] and use proof techniques from the π-calculus. This will allow us to derive laws as the ones above more mechanically. This is the main program for Chapter 4. In Section 4.6 we will give proofs for law 3.16 and 3.20.

3.9 Comparison with the Form- and the $\pi\mathcal{L}$-calculus

In our earlier work on the foundations of Piccola, we specified the semantics of Piccola in terms of translations to $\pi\mathcal{L}$ [Lum99, LAN00] or to the Form calculus [Sch99]. The difference between $\pi\mathcal{L}$ (i.e., the π calculus with labels) and the Form calculus is that the latter allows hiding of labels and forms and contains a testing primitive for labels.

The Piccola calculus is better suited to give a direct semantics of the Piccola language we present in Chapter 5. The enhanced expressiveness of the Piccola calculus with respect to the Form- and the $\pi\mathcal{L}$-calculus are as follows:

- **Form extension.** In $\pi\mathcal{L}$ and the form calculus we have two primitives to extend a form: (i) Binding or normal extension written $F\langle l{=}V\rangle$ where F denotes a form, l a label, V a channel name or a projection expression X_l with X a form variable. In addition in the form calculus, V can be $\mathcal{E}$. In that case, the binding l gets removed or hidden in the form $F\langle l{=}\mathcal{E}\rangle$. (ii) Polymorphic extension written $F \cdot X$ which concatenates the form F with the bindings in the form variable X. In the Piccola calculus there is only a single extension operator $\cdot$ for asymmetric form concatenation. The form calculus and $\pi\mathcal{L}$ differ in the semantics for polymorphic form extension when the right hand side form contains empty bindings (see [Sch99] Section 7.4 for details).

- **Label hiding.** Modification of an existing form by removing the visibility of labels is not supported in $\pi\mathcal{L}$. In the form calculus, the extension with $\mathcal{E}$ can be used to hide a single label. $\mathcal{E}$ is formally constructed as a projection on a label that is not defined, i.e., $\langle\rangle_l$. In addition, polymorphic form restriction, written $F\backslash X$ can be used to hide all labels in F that are bindings in the form denoted by the form variable X. In the Piccola calculus label hiding is available as a primitive service. Hiding several labels is modeled by using inspection.

- **Nested forms.** The syntax for binding is simplified in the Piccola calculus since nested forms are primitive. In the form calculus and $\pi\mathcal{L}$ nested forms must be encoded as constant services.

- **Label matching.** The form calculus provides a matching construct that allows an agent to check whether a given form contains a label. In the Piccola calculus we do not need this primitive since inspect is more expressive and allows us to build a checking predicate within the language. In contrast, we cannot iterate over all the labels in a form in the form calculus nor in $\pi\mathcal{L}$. However, we can encode a label testing service in $\pi\mathcal{L}$ and the Form calculus in the same way like in Section 3.4.4.

- **Higher-order abstractions.** In Piccola, lambda abstractions are specified as user services. Neither $\pi\mathcal{L}$ nor the form-calculus allow us to specify lambda abstractions directly. Services are encoded by channels together with a replicated receiver agent. A drawback of such an encoding is that the reply channels are visible in the resulting program and can be misused, either accidently or by intention. A law similar to the beta equivalence law in Section 3.8 must therefore require much stronger preconditions.

- **Value semantics.** In the Piccola calculus we have value expressions as the basic principle. In $\pi\mathcal{L}$ and in the form calculus, a (parallel) process does not denote a value. In our calculus we only have form expressions.

3.10 Discussion

We have presented the syntax and semantics of the Piccola calculus. The calculus unifies concurrency and values as forms. It contains first-class environments. We have presented a few examples to illustrate its expressiveness, defined an equivalence and given an example of how one can prove that two agents are equivalent.

We summarize the features of the Piccola calculus and how they support composition abstractions.

- **Form extension** allows us to overwrite bindings or to provide defaults. This is illustrated with the encoding of booleans in Section 3.4.

- **Form introspection** and the fact that forms are **finitary** allows us to iterate over all the labels of a form. This is illustrated with *visit* and *nullify* in Section 3.4.4.

- Forms are **immutable** values. In Chapter 6 we use this fact to remove the additional indirection associated with the generic adaption of components.

- Piccola supports **higher-order services** similar to the λ-calculus with eager evaluation strategy.

- The runtime model consists of **agents and channels**. This will be used to overcome compositional mismatches and to define coordination abstractions.

For some of the features, we have not yet seen how they contribute to the validation of this thesis. This will be the main topic of Chapters 7 and 8. Through the design principle of generalization, the Piccola calculus unifies services and forms. Forms are arguments in applications, the explicit environment, and they serve as basis for Piccola's simple component model.

There are some design issues of the calculus that can trigger further work:

Classical input prefix. This would mean that we have $c(x).A$ instead of only $c?$. Intentionally, on receiving a value F from c, the agent $A[x/F]$ is evaluated. However, we also have to take the explicit environment into account which leads to the following rule

$$cG \mid F; c(x).A \rightarrow F \cdot x \mapsto G; A$$

with a similar structure as rule *reduce beta*.

Unify forms and abstractions. It is tempting to say that a form is a finite function from labels to values. Since we have first-class labels, we might try to unify the syntax for sandbox $A; B$ and application AB.

However, this unification comes at a great cost. First, we complicate the evaluation strategy since we distinguish on the functor if we invoke a service or evaluate a term inside a sandbox. Second, we give up the assumption that forms are finite and that we can iterate over all the labels of a form. However, this assumption is crucial to define wrappers and glue code in general. For that reason we have not explored this idea further.

Labeled transition system. We could also investigate a labeled transition system for the Piccola calculus and adopt one of the classical bisimulation (early, late, open). This would require to prove that the chosen bisimulation coincides or subsumes the barbed bisimulation. We have not chosen this approach since it seems preferable for our task to reuse foundational work on the π-calculus to provide a proof system for barbed equivalence. In the next chapter we present a sound encoding of Piccola into the π-calculus for that purpose.

Chapter 4

Pi semantics of Piccola

In this chapter we show that the Piccola calculus can be faithfully embedded into the asynchronous π-calculus. We present an encoding of Piccola into the π calculus that is sound and preserves reductions. We do not require a fully abstract encoding. A fully abstract encoding would mean that equivalent Piccola agents translated into the π-calculus cannot be distinguished by any π-processes. Our milder requirement means that we consider only π-processes which are translations of Piccola agents themselves and state that they cannot distinguish similar agents. Such processes do not break our encoding protocol.

4.1 The Localized π-calculus

We use the localized π-calculus Lπ of Merro and Sangiorgi [Mer00, MS98, San01]. This is the asynchronous π-calculus first introduced by Honda and Tokoro [HT92] with the restriction that the recipient of a channel may only use it in output actions. Only the *output capability* of channels is transmitted. This restriction also forbids the introduction of a matching construct, since testing the identity of channels requires more than output capability. In any case, matching was not defined in the original asynchronous variant. More information on the asynchronous π-calculus can be found in [ACS96, Bou92, FG98, HHK95, San00].

The Lπ-calculus has some important laws that do not hold in the asynchronous case. For example the laws

$$(\nu c)\overline{a}\langle c\rangle = (\nu c)\overline{a}\langle c\rangle \mid c(x) \tag{4.1}$$

$$(\nu c)\overline{a}\langle c\rangle = (\nu c)\overline{a}\langle c\rangle \mid \overline{c}\langle b\rangle \tag{4.2}$$

may seem surprising at first sight. However, all processes can make at first an output of a fresh channel c along a. After that, the derivatives of observables are very different in the classical asynchronous case. In (4.1) a value sent along c is consumed. This is only observable as long as the observer has the input capability on c. A capability he is not granted in Lπ thus he cannot observe a difference. Similar for equation (4.2). An observer cannot receive the communicated value b along c, since he cannot read from c.

4.1.1 Syntax

We use a polyadic calculus which we extend with countably many *constants*. For the constants we have a matching operation. We communicate the constants but will not use them

P, Q	$::=$	$\mathbf{0}$	*inaction*	$\mid$	$a(\tilde{v}).P$	*input prefix*
	$\mid$	$\overline{a}\langle\tilde{v}\rangle$	*output*	$\mid$	$!a(\tilde{v}).P$	*replicated input*
	$\mid$	$(\nu a)P$	*restriction*	$\mid$	$P \mid Q$	*parallel*
	$\mid$	$[x = y]P, Q$	*match*			

Table 4.1: The Lπ-calculus

as channels (See Section 4.1.3).

The grammar of Lπ is given in Table 4.1. In input patterns all the names in $\tilde{v}$ must be distinct. Input processes $a(\tilde{v}).P$ and $!a(\tilde{v}).P$ have the restriction that none of the v_i may *occur free in P in input subject position,* see below.

We introduce a few syntactic categories: the set of all *variables* is the union of the two disjoint sets $\mathcal{C}$ of *channels* or names and $\mathcal{K}$ of *constants*. We use the letters v, w to range over variables, the letters a, b, c, s, p, q to range over channels, and the letters x, y, z to range over constants. We use the terms names, channels and locations interchangeably. We use tilde to denote tuples: $\tilde{v}$ is $v_1, \ldots, v_n$ for $n \geq 0$ and n is the length of the tuple. When $n = 0$, the tuple is empty. We normally omit the brackets. The terms $\overline{a}$ and $a.P$ stand for $\overline{a}\langle\rangle$ and $a().P$, respectively. We write $v_i \in \tilde{v}$ when $0 < i \leq n$ for $\tilde{v} = v_1, \ldots v_n$. $(\nu\tilde{a})P$ abbreviates $(\nu a_1)\ldots(\nu a_n)P$. Here are the rules for scoping variables and channels. Restriction $(\nu a)P$ binds the channel a in P. Input and replicated input $a(\tilde{v}).P$ bind all the $v_i \in \tilde{v}$ in P. The set of *free channels* $fc(P)$ and *bound channels* $bc(P)$ of a process P are defined in the usual way. We identify processes up to α-conversion. By $P\{\tilde{v}/\tilde{w}\}$ with all v_i distinct we denote the process which results from a name-capture avoiding parallel substitution of $\tilde{v}$ for $\tilde{w}$ in P.

To avoid writing too many parentheses, we give ν a higher precedence than parallel composition and make the bodies of input and testing expressions extend as far to the right as possible. For instance $x(y).(\nu z)\overline{y}\langle z\rangle \mid z.P$ means $x(y).(\nu z)(\overline{y}\langle z\rangle \mid z.P)$.

In an input $a(\tilde{v}).P$ and an output $\overline{a}\langle\tilde{v}\rangle$ we say that the channel a is the subject part, the tuple v are the object parts.

4.1.2 Labeled Transition Semantics

We now give the operational semantics of the Lπ-calculus by means of a labeled transition system (LTS). An LTS has the advantage over a reduction relation that it can express both: the internal actions a process can make and the (possible) communications with other processes. The price we have to pay for this expressiveness is that the rules are not so intuitive as the plain reduction semantics.

Transitions are of the form $P \xrightarrow{\mu} P'$, where the label μ ranges over *actions* of the following forms:

τ	interaction (or reduction)
$a[\tilde{v}]$	input of $\tilde{v}$ at a
$(\nu\tilde{b})\overline{a}\langle\tilde{v}\rangle$	output of $\tilde{v}$ at a, extruding bound names $\tilde{b}$ with $\tilde{b} \subset \tilde{v}$.

In an output action $(\nu\tilde{b})\overline{a}\langle\tilde{v}\rangle$ the part $(\nu\tilde{b})$ is used to record those channels in $\tilde{v}$ that were created and are not yet known to the environment. If $\tilde{b}$ is empty, then $(\nu\tilde{b})\overline{a}\langle\tilde{v}\rangle$ is $\overline{a}\langle\tilde{v}\rangle$. The bound and free names of actions are defined in the obvious way. We also say that a is the subject, and $\tilde{v}$ the object of the action.

$$\frac{}{a(\tilde{x}).P \xrightarrow{a[\tilde{v}]} P\{\tilde{v}/\tilde{x}\}}\ \mathit{Inp} \qquad \frac{}{!a(\tilde{x}).P \xrightarrow{a[\tilde{v}]} P\{\tilde{v}/\tilde{x}\} \mid !a(\tilde{x}).P}\ \mathit{Rep}$$

$$\frac{}{\overline{a}\langle\tilde{v}\rangle \xrightarrow{\overline{a}\langle\tilde{v}\rangle} \mathbf{0}}\ \mathit{Out} \qquad \frac{P \xrightarrow{(\nu\tilde{b})\overline{a}\langle\tilde{v}\rangle} P' \quad c \in fc(\tilde{v}) - \tilde{b}}{(\nu c)P \xrightarrow{(\nu\tilde{b},c)\overline{a}\langle\tilde{v}\rangle} P'}\ \mathit{Open}$$

$$\frac{P_1 \xrightarrow{\mu} P_1' \quad bc(\mu) \cap fc(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}\ \mathit{Par} \qquad \frac{P \xrightarrow{\mu} P' \quad c \notin (fc(\mu) \cup bc(\mu))}{(\nu c)P \xrightarrow{\mu} (\nu c)P'}\ \mathit{Res}$$

$$\frac{P_1 \xrightarrow{(\nu\tilde{b})\overline{a}\langle\tilde{v}\rangle} P_1' \quad P_2 \xrightarrow{a[\tilde{v}]} P_2' \quad \tilde{b} \cap fc(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} (\nu\tilde{b})P_1' \mid P_2'}\ \mathit{Com}$$

$$\frac{P \xrightarrow{\mu} P'}{[x = x]P, Q \xrightarrow{\mu} P'}\ \mathit{Test}^T \qquad \frac{Q \xrightarrow{\mu} Q' \quad x \neq y}{[x = y]P, Q \xrightarrow{\mu} Q'}\ \mathit{Test}^F$$

Table 4.2: Labeled Transition System for the Lπ-calculus

The set of rules for the LTS of Lπ is given in Table 4.2. The symmetric rules of (Com) and (Par) are omitted. This is the standard transition system for the π-calculus extended with straightforward rules for testing constants. *Weak transitions* are defined as usual: $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$; $\overset{\mu}{\Longrightarrow}$ stands for the relation composition $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$. Two relations $\mathcal{R} \in \mathcal{A} \times \mathcal{B}$ and $\mathcal{R}' \in \mathcal{B} \times \mathcal{C}$ are composed to $\mathcal{R}\mathcal{R}' = \{(a, c) \in \mathcal{A} \times \mathcal{C} \mid \exists b \in \mathcal{B} : (a, b) \in \mathcal{R}$ and $(b, c) \in \mathcal{R}'\}$ where $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ are any sets.

An important fact relating the LTS and the reduction relation are that the τ reductions coincide with the ordinary reduction relation. For a formal proof see for instance [ACS96] or [Mil99].

4.1.3 Sorting

As usual we only consider well-sorted terms. In such terms, the length of output and input prefix match as well as the type of variables with concrete values communicated. E.g., a variable that is used in a matching expression cannot be substituted with a channel and vice versa. A sorting is an assignment of *sorts* to names. It specifies the arity of each channel and, by recursion, of the names carried by that name. We write $a : s$ if channel a has sort s. The formal presentation of a sorting system is not a technical contribution here and we refer the reader to the standard literature [Mil91, Mil99, SW01]. A typed version of the polyadic Lπ extended with variants, tuples, and constants is used by Merro *et al.* to give a semantics to Obliq [Mer00, MKN00].

4.2 Behavioural Equivalence

We define behavioural equality using the notion of barbed congruence [MS92]. As in the last chapter we first have to define barbs as our observability predicate.

Definition 4.1 (asynchronous observability) *For a process P the predicate $P \downarrow_a$ holds if there is a derivative Q and an output action μ with subject a such that $P \xrightarrow{\mu} Q$. We write $P \Downarrow_a$ if there is a Q such that $P \Longrightarrow Q$ and $Q \downarrow_a$.*

Then barbed congruence is defined as:

Definition 4.2 (Barbed congruence) *A symmetric relation $\mathcal{S}$ is a* (weak) barbed bisimulation *if $P \ \mathcal{S}\ Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$ then there is a Q' with $Q \Longrightarrow Q'$ and $P' \ \mathcal{S}\ Q'$*
2. *If $P \downarrow_a$ then $Q \Downarrow_a$.*

Two processes P and Q are (weakly) barbed bisimilar, *written $P \dot{\cong} Q$ if $P \ \mathcal{S}\ Q$ for some barbed bisimulation $\mathcal{S}$. Two processes are* (weakly) barbed congruent, *written $P \cong Q$ if for each context $\mathcal{C}$, we have $\mathcal{C}[P] \dot{\cong} \mathcal{C}[Q]$.*

Barbed congruence is the equality we are mainly interested in. It says that an observer cannot distinguish two processes because they both have the same output capabilities. However proving directly that two agents are barbed congruent is often hard since it requires us to exhibit a bisimulation and to show that it is closed under all contexts. In the π-calculus literature there is a large body of proof techniques. In the following we will present those needed for the proofs in the rest of this chapter.

4.2.1 Proof Techniques

Barbed congruence suffers from the universal quantification on contexts. This quantification makes it very hard to prove process equalities and makes mechanical checking impossible. It is therefore important to find simpler proof techniques that do not use universal context quantification. Such formulations should ideally coincide with, or imply barbed congruence.

In the asynchronous π-calculus, *ground bisimilarity* is such a technique [San00]. The key idea is that for input actions, we can assume that all the received names are fresh.

Definition 4.3 (Ground bisimiliarity) *A symmetric relation $\mathcal{S}$ is a* strong ground bisimulation *if $P \ \mathcal{S}\ Q$ implies:*

1. *If $P \xrightarrow{\mu} P'$ then there is a Q' with $Q \xrightarrow{\mu} Q'$ and $P' \ \mathcal{S}\ Q'$ if μ is τ or an output action.*
2. *If $P \xrightarrow{a[\tilde{v}]} P'$ and $\tilde{v} \cap fv(P,Q) \subset \mathcal{K}$ then there is a Q' with $Q \xrightarrow{a[\tilde{v}]} Q'$.*

Two processes P and Q are strong ground bisimilar, *written $P \sim Q$ if $P \ \mathcal{S}\ Q$ for some strong ground bisimulation $\mathcal{S}$. The weak versions of the relations are defined in the usual way by replacing $Q \xrightarrow{\mu} Q'$ with $Q \overset{\mu}{\Longrightarrow} Q'$. Weak ground bisimilarity is indicated by $\approx$.*

The side condition for input actions means that the channels received must be fresh. Thus no channel instantiation is required in the clause for input actions. To check whether two processes $a(\tilde{v}).P$ and $b(\tilde{v}).Q$ are equivalent it is enough to compare Q and P.

The following proposition states that ground bisimulation is a congruence and that it implies barbed congruence.

Proposition 4.4 *$P \approx Q$ implies*

1. *$\mathcal{C}[P] \approx \mathcal{C}[Q]$, and*
2. *$P \cong Q$.*

(1) is proved in [San00]. (2) is proved like analogous results for the asynchronous π-calculus, see [ACS96, San01, San00].

To prove bisimulation, it is helpful to reduce the size of the necessary relation to exhibit. One such technique is bisimulation up to expansion [AKH92, SM92]. The expansion relation is a preorder derived from ground congruence by comparing the number of silent actions performed. Intuitively, $P \lesssim Q$ implies $P \approx Q$ and that P is at least as fast as Q, i.e., Q may have more internal actions than P.

We write $P \xrightarrow{\hat{\tau}} Q$ for $P = Q$ or $P \rightarrow Q$, and $P \xrightarrow{\hat{\mu}} Q$ for $P \xrightarrow{\mu} Q$ when $\mu \neq \tau$.

Definition 4.5 (Expansion) *A relation $\mathcal{S}$ on processes is an* expansion *if $P \mathrel{\mathcal{S}} Q$ implies:*

1. *If $P \xrightarrow{\mu} P'$ then there is a Q' such that $Q \overset{\mu}{\Longrightarrow} Q'$ and $P' \mathrel{\mathcal{S}} Q'$ if μ is τ or an output action,*
2. *If $Q \xrightarrow{\mu} Q'$ then there is a P' such that $P \xrightarrow{\hat{\mu}} P'$ and $P' \mathrel{\mathcal{S}} Q'$ if μ is τ or an output action.*
3. *If $P \xrightarrow{a[\tilde{v}]} P'$ and $\tilde{v} \cap fv(P,Q) \subset \mathcal{K}$ then there is a Q' such that $Q \overset{a[\tilde{v}]}{\Longrightarrow} Q'$ and $P' \mathrel{\mathcal{S}} Q'$.*
4. *If $Q \xrightarrow{a[\tilde{v}]} Q'$ and $\tilde{v} \cap fv(P,Q) \subset \mathcal{K}$ then there is a P' such that $P \xrightarrow{\widehat{a[\tilde{v}]}} P'$ and $P' \mathrel{\mathcal{S}} Q'$.*

A process Q expands P, written $P \lesssim Q$, if $P \mathrel{\mathcal{S}} Q$ for some expansion $\mathcal{S}$.

The expansion relation was originally defined for CCS [Mil89] by Arun-Kumar and Hennessy [AKH92]. They showed that it is a preorder and preserved by all CCS contexts except summation. In the asynchronous π-calculus, expansion is strictly in the middle of strong and weak and ground bisimilar: $\sim \subset \lesssim \subset \approx$.

The expansion relation can be used for so called "up-to" proof techniques. For showing a relation $\mathcal{S}$ to be a bisimulation we have to show that $P \xrightarrow{\mu} P'$ implies that there is a Q' with $Q \xrightarrow{\mu} Q'$ and $P' \mathrel{\mathcal{S}} Q'$. An up-to proof technique relaxes the requirement of $P' \mathrel{\mathcal{S}} Q'$ to a weaker form. The technique we use allows us to consider $P' \mathrel{\mathcal{S}} Q'$ only up to a static context and to expansion.

Definition 4.6 (Static context) *A static context has the form $(\nu\tilde{a})[] \mid P$.*

Static contexts ensure the locality principle of Lπ. They ensure that we cannot put a process which does input on a free name into the context that binds that name in an input prefix. For instance, $a(c).[]$ is not a static context, since, if we replace $[]$ by $c(x).P$ is not a process of $L\pi$ anymore.

Definition 4.7 (Bisimulation up-to context and $\gtrsim$) *A symmetric relation $\mathcal{S}$ is a* bisimulation up-to $\gtrsim$ and context *if $P\ \mathcal{S}\ Q$ implies:*

1. *If $P \xrightarrow{\mu} P''$ then there is a static context $\mathcal{C}$ and processes P' and Q' such that $P'' \gtrsim \mathcal{C}[P']$, $Q \overset{\hat{\mu}}{\Longrightarrow}\gtrsim \mathcal{C}[Q']$, and $P'\ \mathcal{S}\ Q'$ for μ is τ or an output transition,*

2. *If $P \xrightarrow{a[\tilde{v}]} P''$ and $\tilde{v} \cap fv(P,Q) \subset \mathcal{K}$ then there is a static context $\mathcal{C}$ and processes P' and Q' such that $P'' \gtrsim \mathcal{C}[P']$, $Q \overset{\widehat{a[\tilde{v}]}}{\Longrightarrow}\gtrsim \mathcal{C}[Q']$, and $P'\ \mathcal{S}\ Q'$.*

The following lemma states that we can use the up-to context and expansion technique for proving ground congruence which is included in barbed congruence [SM92]:

Lemma 4.8 *If $\mathcal{S}$ is a bisimulation up to context and $\gtrsim$ then it holds that $\mathcal{S} \subseteq \approx$.*

In order to show that $\mathcal{S}$ is a bisimulation up to context and expansion we have to complete the diagram

$$\begin{tikzcd}
P \arrow[d, "\mu"'] & \mathcal{S} & Q \\
P'' & &
\end{tikzcd}
\qquad \text{to} \qquad
\begin{tikzcd}
P \arrow[d, "\mu"'] & & \mathcal{S} & Q \arrow[d, Rightarrow, "\hat{\mu}"] & \\
P'' \gtrsim & \mathcal{C}[P'] & & Q'' \gtrsim & \mathcal{C}[Q'] \\
& P' & \mathcal{S} & & Q'.
\end{tikzcd}$$

4.2.2 Some Laws for Lπ

In this section we collect important laws of Lπ. We will use them in the rest of this chapter. The first law states that deterministic communication is an expansion:

Lemma 4.9 *For all a, P, and $\tilde{v}$, $\tilde{x}$ it holds:*

$$(\nu a)\overline{a}\langle\tilde{v}\rangle \mid a(\tilde{x}).P \gtrsim (\nu a)P\{\tilde{v}/\tilde{x}\}$$

This lemma is one of the most basic laws. The left hand side makes a single τ transition and becomes the right-hand side. The only transition the process $a(\tilde{x}).P$ can make is inputing a tuple from channel a. The sorting regime restricts the above lemma to a, $\tilde{v}$, and $\tilde{x}$ of matching sorts.

The *replication theorems* [Mil99] express important properties of replicated processes. The idea is that we can distribute — this means duplicate — a replicated processes over parallel composition as long as these processes are "used correctly". The words "using correctly" mean that the channels from which the replicated processes input are only used for sending values and not for other communication.

Lemma 4.10 *Let R, P, Q be processes not containing the name a in input position. Then:*

1. $!a(\tilde{v}).S \approx (!a(\tilde{v}).S) \mid !a(\tilde{v}).S$

2. $(\nu a)P \mid Q \mid a(\tilde{x}).R \approx ((\nu a)P \mid a(\tilde{x}).R) \mid (\nu a)Q \mid a(\tilde{x}).R$

The first law is a classical result valid for the π-calculus [Mil99]. The second law is shown by Milner [Mil91] for the π-calculus with the restriction that a occurs free in P, Q, R only in output subject position. The stronger variant without the side condition for the Lπ is shown in [Mer00].

The standard replication lemma requires the precondition that a occurs free in P, Q, R only in output subject position. Otherwise P and Q may communicate along a in the left-hand side, whereas such a communication is not possible in the right-hand side where a are two distinct channels. Pierce and Sangiorgi [PS96] showed that the precondition can be relaxed using a type system with input and output capabilities and requiring that the processes P, Q, R only have the output capability on a.

4.3 Recursive Definitions

Our encodings of forms make use of processes that are defined parametrically. We define a process abstraction as:

$$A(\tilde{v}) \overset{\text{def}}{=} P_A,$$

where A is a process identifier and P_A is a process that may contain "processes" of the form $A\langle\tilde{w}\rangle$. It is standard to translate such process definitions into plain π processes using replication [Mil92]. The translation works as follows.

Suppose the process P_A is $P_A = ...A\langle\tilde{w}\rangle...$, and we have a process $Q = ...A\langle\tilde{u}\rangle....$ Now we can translate Q into a π process as follows:

1. Invent a fresh name, e.g. a to stand for A. Fresh means that a should not be used anywhere in P_A and Q.

2. Replace any process $A\langle\tilde{w}\rangle$ in P_A and Q with a "call" or sending process $\overline{a}\langle\tilde{w}\rangle$. The resulting processes are $\hat{P}_A$ and $\hat{Q}$, respectively.

3. The translated process Q is:
$$(\nu a)\hat{Q} \mid !a(\tilde{v}).\hat{P}_A$$

Observe that this translation generalizes directly to the case of several processes defined by mutual recursion.

The difference between the original process and the translated is that the latter has more τ transitions. The encoding uses a communication to instantiate the process $A\langle\tilde{v}\rangle$. Since we are mainly concerned with weak observability this is not an issue.

Special care is needed to ensure that we do not violate the locality constraint of Lπ by the translation. Merro [Mer00] shows how to ensure that all v_i are not used in subject input position.

4.4 Encoding Piccola in Lπ

In this section we present an encoding of the Piccola calculus of Chapter 3 into the Lπ-calculus. We split the encoding into two parts. In Section 4.4.2 we define some helper abbreviations that end up being an encoding of form values. In Section 4.4.3 we give the encoding of Piccola to π.

4.4.1 Terminology

It is sometimes cumbersome to discuss the behaviour of processes in terms of sending and receiving. The intuition behind our encodings is better understood when we use terms like invoking a function and returning a result. We now explain some idioms to simplify the explanations of processes.

We use replicated processes to *model functions* [Mil92] using a continuation passing style. The invocation of a function is done by sending a request along the channel on which the replicated process listens. The request consists of the argument for the function and a *reply channel* and optionally an *error channel*. The reply or error channel triggers the continuation of the function. The process modeling the function then either sends the result back along the reply channel from where the client depicts it or triggers an error by sending the empty tuple along the error channel. We say that a process *invokes a function* to describe a process sending a request along the service channel and in parallel waiting on the reply or error channel for the result. The input prefixed processes on the reply or error channel are the continuation of the invocation.

We model a form as the composition of four functions. The semantics of theses functions is explained in the following section. When we say *a form is sent along a channel* we actually mean that the replicated processes encoding the form are instantiated and the quadruple of names that give access to these functions is sent along the channel.

4.4.2 Encoding of Forms

We use the following conventions for variables for forms and tuples. A Piccola form F is encoded by four functions on the channels $\tilde{f} = \langle f_p, f_i, f_h, f_s \rangle$. Similar, the form G corresponds to $\tilde{g}$ etc. The names $\tilde{f}$ give access to the following functions: *projection, invocation, hiding*, and *selection*.

Projection is modeled by the process $!f_p(x, p, q).P$ where x is the label to be looked up. If F contains a binding for x then the form $F; x$ — or more precisely its encoding — is returned along p. If the form F does not contain the label x, an empty tuple is returned on the error channel q. The process P ensures that precisely either a form is sent along p or the empty tuple along q.

Invocation is modeled by the process $!f_i(p, q).P$. If the form F contains a service, the channel giving access to the service is returned along p. If the form does not contain a service, an empty tuple is sent along q to signal the absence. The agent P ensures that precisely either a channel is sent along p or the empty tuple along q.

Hiding is modeled by the process $!f_h(x, p).P$ where x represents the label to be hidden. P sends the form $hide_x F$ along p. Note that the form $hide_x F$ is always defined, there is no need for an error channel.

Selection is modeled by the process $!f_s(p, q).P$. If the form F contains a binding for arbitrary label x the label x is returned along p. If F does not contain any bindings we send the empty tuple along q. The agent P ensures that precisely either a label x is sent along p or the empty tuple along q.

$$\begin{array}{rlll}
\mathit{empty}(\tilde{f}) & \stackrel{\text{def}}{=} & (!f_p(x,p,q).\overline{q}) & \mid (!f_i(p,q).\overline{q}) \\
 & & \mid (!f_h(x,p).\overline{p}\langle\tilde{f}\rangle) & \mid (!f_s(p,q).\overline{q}) \\
\\
\mathit{fun}(\tilde{f},s) & \stackrel{\text{def}}{=} & (!f_p(x,p,q).\overline{q}) & \mid (!f_i(p,q).\overline{p}\langle s\rangle) \\
 & & \mid (!f_h(x,p).\overline{p}\langle\tilde{f}\rangle) & \mid (!f_s(p,q).\overline{q}) \\
\\
\mathit{bind}(\tilde{f},y,\tilde{g}) & \stackrel{\text{def}}{=} & (!f_p(x,p,q).[y=x]\overline{p}\tilde{g},\overline{q}) & \mid (!f_i(p,q).\overline{q}) \\
 & & \mid (!f_h(x,p).[y=x](\nu\tilde{h})\overline{p}\langle\tilde{h}\rangle \mid \mathit{empty}\langle\tilde{h}\rangle,\overline{p}\langle\tilde{f}\rangle) & \mid (!f_s(p,q).\overline{p}\langle y\rangle) \\
\\
\mathit{ext}(\tilde{f},\tilde{g},\tilde{h}) & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{(!f_p(x,p,q).(\nu r)\overline{h_p}\langle x,p,r\rangle \mid r.\overline{g_p}\langle x,p,q\rangle)} \\
 & & \multicolumn{2}{l}{\mid (!f_i(p,q).(\nu r)\overline{h_i}\langle r,r\rangle \mid r.\overline{g_i}\langle p,q\rangle)} \\
 & & \multicolumn{2}{l}{\mid (!f_h(x,p).(\nu r_1,r_2)\overline{g_h}\langle x,r_1\rangle \mid \overline{h_h}\langle x,r_2\rangle \mid r_1(\tilde{g}).r_2(\tilde{h}).(\nu\tilde{f})\overline{p}\langle\tilde{f}\rangle \mid \mathit{ext}\langle\tilde{f},\tilde{g},\tilde{h}\rangle)} \\
 & & \multicolumn{2}{l}{\mid (!f_s(p,q).(\nu r_1,r_2)\overline{g_s}\langle r_1,r_2\rangle \mid \overline{h_s}\langle r_1,r_2\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q})}
\end{array}$$

Table 4.3: Encoding Forms

Table 4.3 defines a set of process abstractions. We use them as basic building blocks to encode forms. Informally, they are used to instantiate the empty form, bindings and service forms, and to compose forms by extension. It is important to note that these abbreviations are only in Lπ when defined on channel names $\tilde{f}$ that are bound by restriction and not received. For instance the following term is not a valid process expression:

$$a(\tilde{f}).\mathit{empty}\langle\tilde{f}\rangle$$

since it violates the locality constraint of the Lπ-calculus. We have to ensure that all occurrences of the above abbreviations happen in scopes as:

$$a(\tilde{g}).(\nu\tilde{f})P \mid \mathit{bind}\langle\tilde{f},l,\tilde{g}\rangle$$

Of course, this restriction applies only for the first "argument" part, i.e., for $\tilde{f}$ and not for $\tilde{g}$ in the example. We now explain the individual form encodings.

Empty Form. The process $\mathit{empty}\langle\tilde{f}\rangle$ installs the empty form on the channels of $\tilde{f}$. The process $!f_p(x,p,q).\overline{q}$ serves requests for projections on a label x. Since the empty form does not contain any bindings, the error process $\overline{q}$ signals that no binding (for label x) is defined. The invocation function f_i has a similar behaviour: it signals that there is no invocation channel. The selection function on f_s signals that there is no label in the form. The hiding process $!f_h(y,p).\overline{p}\langle\tilde{f}\rangle$ sends the form $\tilde{f}$ back along the return channel p as result for the empty form with label y hidden. This is the empty form as defined by the *hide empty* structural rule: $\mathit{hide}_x\ \epsilon \equiv \epsilon$.

Abstraction. The encoding of a user defined service is straightforward given a service channel s. Essentially $\mathit{fun}\langle\tilde{f},s\rangle$ corresponds to encoding of the empty form. The only difference is the invocation process $f_i(p,q).\overline{p}\langle s\rangle$ which returns the service channel s instead of signaling the absence of a service on the error channel.

Binding. The process $bind\langle \tilde{f}, y, \tilde{g}\rangle$ defines the binding $y \mapsto G$. The process $!f_p(x, p, q).P$ serves request for projections. If a client projects on y, the tuple $\tilde{g}$ is returned on the result channel p by the process $\overline{p}\langle \tilde{g}\rangle$. Otherwise — a projection on a label different than y is made — the error process $\overline{q}$ signals the absence of the label. The process $!f_i(p, q).P$ signals on q that there is no service available. The process serving hide requests is the most complicated. If the hide request is to hide the label x then an empty form is instantiated at fresh channels $\tilde{h}$ and returned along the result channel p by $(\nu\tilde{h})\overline{p}\langle\tilde{h}\rangle \mid empty\langle\tilde{h}\rangle$. Otherwise the tuple $\tilde{f}$ is returned. Finally the select label process returns the label x.

Extension. This encoding is the most complicated one. It relies on the encodings of $\tilde{g}$ and $\tilde{h}$ to model the form $F = G \cdot H$. However, the processes are easy to read and understand once we are familiar with the continuation passing style necessary for the π-calculus. The process handling projections on label x is:

$$(\nu r)\overline{h_p}\langle x, p, r\rangle \mid r.\overline{g_p}\langle x, p, q\rangle$$

It creates a private channel r and runs two parallel processes. The first calls a projection on h_p with p as the result channel and r as the error channel. The other process waits on r and forwards the projection request to g_p. Now two cases may occur: (1) either H contains a binding for label x. In this case the result is returned on p and there will never be a sender on r. (2) If H does not contain a binding for the label x this gets signaled on r and h_p (or, to be more precise, the replicated process that reads on it) will not use r anymore. In this case the process $r.\overline{g_p}\langle x, p, q\rangle$ forwards the projection request to g_p. The process handling invocation requests is similar.

The process handling hiding recursively forwards the hiding request to g_h and h_h. Both these processes are expected to return forms along r_1 and r_2, respectively. These forms are concatenated and returned.

The selection function works as follows: It makes use of two private channels r_1 and r_2. Now both sub-forms are requested to send either a label on r_1 or to signal the absence of any label on r_2. If a label is present in either $\tilde{g}$ or $\tilde{h}$ then at least one label is sent along r_1 and forwarded to p. If both forms $\tilde{g}$ and $\tilde{h}$ do not contain any label the process $r_2.r_2.\overline{q}$ receives both inputs and signals the absence along q. When both forms return a label, arbitrary is returned by $\overline{p}\langle x\rangle$. Such an encoding of internal choice [NP96] is only valid since the processes on g_s and h_s make a strict boolean use of p and q, i.e., they exactly either send on p or on q.

Using these encodings we define *form processes* up to strong ground congruence. The idea is that we require for any process P that models a form that it can be constructed by finitely many applications of the basic abstractions.

Definition 4.11 (Form process) *Let $\mathcal{F_P}$ be the set of processes closed under the following rules and strong ground congruence.*

1. $empty\langle\tilde{f}\rangle \in \mathcal{F_P}$.
2. $fun\langle\tilde{f}, s\rangle \in \mathcal{F_P}$.
3. *if* $P \in \mathcal{F_P}$ *with* $\tilde{g} \in fv(P)$ *and* $\tilde{f} \notin fv(P)$ *then* $(\nu\tilde{g})P \mid bind\langle\tilde{f}, x, \tilde{g}\rangle \in \mathcal{F_P}$.
4. *if* $P \in \mathcal{F_P}$ *with* $\tilde{g}, \tilde{h} \in fv(P)$ *and* $\tilde{f} \notin fv(P)$ *then* $(\nu\tilde{g}, \tilde{h})P \mid ext\langle\tilde{f}, \tilde{g}, \tilde{h}\rangle \in \mathcal{F_P}$.

The form processes are processes that encode forms. By construction, a form process has one quadruple of free names $\tilde{f}$. We say that the process makes the form available at $\tilde{f}$. In addition a form process may have free names s that come from $fun\langle\tilde{f}, s\rangle$. The side conditions in the last two rules of Definition 4.11 ensure that we do not define cyclic forms. Such forms would have non-terminating behaviour. For example assume the following process.

$$P = empty\langle\tilde{e}\rangle \mid ext\langle\tilde{f}, \tilde{e}, \tilde{f}\rangle$$

Intuitively the process P stands for the recursive form $x = \epsilon \cdot x$, i.e., a form which is the extension of the empty form with itself. Projection on any label y leads to a nonterminating communication:

$$P \xrightarrow{\overline{f_p}\langle y,p,q\rangle} P \mid (\nu r_1)\overline{f_p}\langle y,p,r_1\rangle \mid r_1.\overline{e_p}\langle y,p,q\rangle \xrightarrow{(\nu r_1)\overline{f_p}\langle y,p,r_1\rangle} \ldots$$

The following lemma states that for any form process, the projection function deterministically either returns a result or signals the absence of an appropriate binding on the error channel. The hiding service always returns a form. The selection function either signals the absence of any label binding in the form or randomly returns one. In the correctness proofs of the encoding of Piccola agents later on we rely on these properties of form processes. We use this lemma for up-to expansion proofs and replace form projection by its result.

Lemma 4.12 *For any $P \in \mathcal{F_P}$ with $\tilde{f} \in fv(P)$ and $p, q \notin fv(P)$ the following holds:*

1. *Projection: For any x constant $P \xrightarrow{f_p[x,p,q]} P'$ implies*
   - *(a) either $P' \gtrsim P \mid (\nu\tilde{g})\overline{p}\langle\tilde{g}\rangle \mid P'')$ and $P'' \in \mathcal{F_P}$ with $\tilde{g} \in fv(P'')$*
   - *(b) or $P' \gtrsim P \mid \overline{q}$.*
2. *Invocation: $P \xrightarrow{f_i[p,q]} P'$ implies*
   - *(a) either $P' \gtrsim P \mid \overline{p}\langle s\rangle$*
   - *(b) or $P' \gtrsim P \mid \overline{q}$*
3. *Hiding: $P \xrightarrow{f_h[x,p]} P'$ with $P' \gtrsim P \mid (\nu\tilde{g})P'' \mid \overline{p}\langle\tilde{g}\rangle$, $P'' \in \mathcal{F_P}$, and $g \in fc(P'')$*
4. *Selection: $P \xrightarrow{f_s[p,q]} P'$ implies*
   - *(a) either $P' \stackrel{p[x]}{\Longrightarrow} P$ for some x*
   - *(b) or $P' \gtrsim P \mid \overline{q}$.*

Proof. See Appendix C.1. □

We give an informal explanation of the four parts of the lemma. Assume we have a form process P that encodes form F at $\tilde{f}$. Part (1) says that we can interact along f_p with the process to project on a label. Sending the tuple $\langle x, p, q\rangle$ along f_p either returns along p a tuple $\tilde{g}$ encoding another form or signals the absence of the label x by sending the empty tuple along q. Note that the *answer* given for a projection request is deterministic.

Part (2) says that we can interact along f_i to get the service channel s or to receive the absence of any service in the form F along channel q.

Part (3) defines how to hide a label. When we send the request $\overline{f_h}\langle x, p\rangle$ we receive a form $\tilde{g}$ along p.

Part (4) explains how form inspection works. When we interact with P by sending the tuple $\langle p, q\rangle$ along f_s, we either get a label x back along p or we get the empty tuple along q. Notice that the second case is deterministic, i.e., we always receive the empty tuple along q. On the other hand the first alternative is non-deterministic: we can receive different labels x for each invocation $\overline{f_s}\langle p, q\rangle$.

4.4.3 Encoding Agents

In Table 4.4 we define an encoding of a Piccola agent A into the π-process $[\![A]\!]_a^{\tilde{e}}$. The process $[\![A]\!]_a^{\tilde{e}}$ sends the value of A along channel a and uses the environment form $\tilde{e}$. The process $[\![A]\!]_a$ evaluates A in the environment given by the empty form. We assume that the helper channels r and r_i are fresh.

The encoding works as follows:

- The empty form is translated by the form process for the empty form and a sender process that emits the accessors names for the empty form.
- A variable x is translated to a process that uses the environment's project function e_p to project on the label x. If the environment contains a binding for x then this value is sent along the result channel a. Otherwise an empty tuple is sent on the dummy error channel r. The agent is stuck. See Section 3.6 how to detect illegal projections within the Piccola agents.
- There are several primitive services that are all encoded using the same schema:

$$[\![A]\!]_a^{\tilde{e}} = (\nu \tilde{f}, s)\overline{a}\langle\tilde{f}\rangle \mid fun\langle\tilde{f}, s\rangle \mid !s(\tilde{g}, r).P$$

  The encoding instantiates a fresh service form $fun\langle\tilde{f}, s\rangle$ and sends its names along channel a. The service body itself is represented by the replicated process $!s(\tilde{g}, r).P$. Thus, in the process P the argument is available at the names $\tilde{g}$ and the result has to be sent along r.

  - The process P for the primitive hiding service $hide_x$ calls the hiding function of the passed form $\tilde{g}$ and returns the new form with the label x hidden.
  - The process P for the primitive bind service $(x\mapsto)$ creates a new binding form with the help of process $bind\langle\tilde{h}, x, \tilde{g}\rangle$ and returns the names $\tilde{h}$ back along the reply channel r.
  - The process P for user defined abstractions $\lambda x.A$ is recursively defined by $[\![A]\!]_r^{\tilde{e}'}$ where $\tilde{e}'$ is the environment $\tilde{e}$ extended with the binding $x\mapsto G$.
  - For the primitive inspect **L** the associated process P looks more complicated. It uses the inspection facility encoded by g_s to either fetch a label x or to know that the passed form G does not contain any bindings. If G has a label a constant x is sent along r_1. In that case we return along r the encoding of the service $\lambda x.(x; \text{isLabel})label_x$. If G has no labels the empty tuple is received from r_2. In that case P tries to fetch the invocation channel by using g_i. If there is a service channel s we return along r the encoding of $\lambda x.(x; \text{isService})\epsilon$. Otherwise the inspected

$$
\begin{aligned}
\llbracket A \rrbracket_a &= (\nu \tilde{e}) empty\langle \tilde{e} \rangle \mid \llbracket A \rrbracket_a^{\tilde{e}} \\
\llbracket \epsilon \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f})\overline{a}\langle \tilde{f} \rangle \mid empty\langle \tilde{f} \rangle \\
\llbracket x \rrbracket_a^{\tilde{e}} &= (\nu r)\overline{e_p}\langle x, a, r \rangle \\
\llbracket hide_x \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f}, s)\overline{a}\langle \tilde{f} \rangle \mid fun\langle \tilde{f}, s \rangle \mid !s(\tilde{g}, r).\overline{g_h}\langle x, r \rangle \\
\llbracket x\mapsto \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f}, s)\overline{a}\langle \tilde{f} \rangle \mid fun\langle \tilde{f}, s \rangle \mid !s(\tilde{g}, r).(\nu \tilde{h})\overline{r}\langle \tilde{h} \rangle \mid bind\langle \tilde{h}, x, \tilde{g} \rangle \\
\llbracket \mathbf{R} \rrbracket_a^{\tilde{e}} &= \overline{a}\langle \tilde{e} \rangle \\
\llbracket A; B \rrbracket_a^{\tilde{e}} &= (\nu r)\llbracket A \rrbracket_r^{\tilde{e}} \mid r(\tilde{f}).\llbracket B \rrbracket_a^{\tilde{f}} \\
\llbracket A \cdot B \rrbracket_a^{\tilde{e}} &= (\nu r_{1\ldots2})\llbracket A \rrbracket_{r_1}^{\tilde{e}} \mid r_1(\tilde{f}).\llbracket B \rrbracket_{r_2}^{\tilde{e}} \mid r_2(\tilde{g}).(\nu \tilde{h})\overline{a}\langle \tilde{h} \rangle \mid ext\langle \tilde{h}, \tilde{f}, \tilde{g} \rangle \\
\llbracket \lambda x.A \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f}, s)\overline{a}\langle \tilde{f} \rangle \mid fun\langle \tilde{f}, s \rangle \\
&\qquad \mid !s(\tilde{g}, r).(\nu \tilde{h}, \tilde{e}')\llbracket A \rrbracket_r^{\tilde{e}'} \mid bind\langle \tilde{h}, x, \tilde{g} \rangle \mid ext\langle \tilde{e}', \tilde{e}, \tilde{h} \rangle \\
\llbracket AB \rrbracket_a^{\tilde{e}} &= (\nu r_{1\ldots4})\llbracket A \rrbracket_{r_1}^{\tilde{e}} \mid r_1(\tilde{f}).\llbracket B \rrbracket_{r_2}^{\tilde{e}} \mid r_2(\tilde{g}).\overline{f_i}\langle r_3, r_4 \rangle \mid r_3(s).\overline{s}\langle \tilde{g}, a \rangle \\
\llbracket \mathbf{L} \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f}, s)\overline{a}\langle \tilde{f} \rangle \mid fun\langle \tilde{f}, s \rangle \\
&\qquad \mid !s(\tilde{g}, r).(\nu r_{1\ldots4})\overline{g_s}\langle r_1, r_2 \rangle \mid (r_1(x).\llbracket \lambda x.(x; isLabel) label_x \rrbracket_r) \\
&\qquad\qquad\qquad\qquad\qquad \mid (r_2.\overline{g_i}\langle r_3, r_4 \rangle) \\
&\qquad\qquad\qquad\qquad\qquad \mid (r_3(s).\llbracket \lambda x.(x; isService)\epsilon \rrbracket_r) \\
&\qquad\qquad\qquad\qquad\qquad \mid (r_4.\llbracket \lambda x.(x; isEmpty)\epsilon \rrbracket_r) \\
\llbracket A \mid B \rrbracket_a^{\tilde{e}} &= ((\nu r)\llbracket A \rrbracket_r^{\tilde{e}}) \mid \llbracket B \rrbracket_a^{\tilde{e}} \\
\llbracket \nu c.A \rrbracket_a^{\tilde{e}} &= (\nu c)\llbracket A \rrbracket_a^{\tilde{e}} \\
\llbracket c \rrbracket_a^{\tilde{e}} &= (\nu \tilde{f}, s)\overline{a}\langle \tilde{f} \rangle \mid fun\langle \tilde{f}, s \rangle \mid !s(\tilde{g}, r).\overline{c}\langle \tilde{g} \rangle \mid (\nu \tilde{h})\overline{r}\langle \tilde{h} \rangle \mid empty\langle \tilde{h} \rangle \\
\llbracket c? \rrbracket_a^{\tilde{e}} &= c(\tilde{f}).\overline{a}\langle \tilde{f} \rangle
\end{aligned}
$$

where $label_x$ is the Piccola agent to encode a first-class label for x

$$label_x = project{\mapsto}(\epsilon; \lambda x.x; x) \cdot hide{\mapsto}hide_x \cdot bind{\mapsto}(x{\mapsto})$$

Table 4.4: Encoding of Piccola in Lπ

form G is the empty form and we return $\lambda x.(x; \text{isEmpty})\epsilon$. Note that the helper names $r_{1...4}$ are restricted. If a tuple is sent along r_1, the other names are not used and the associated receiver processes can be garbage collected.

  – The last primitive service is sending a form. The process P sends the tuple $\tilde{g}$ along c and returns the empty form along r.

- The translation of an application AB is done in three steps: First the value $\tilde{f}$ of A is communicated along a private helper channel r_1. Then the value $\tilde{g}$ of B is communicated along another helper channel r_2. Finally, the service channel s is fetched from the form $\tilde{f}$. If there is no service channel, the error is signaled along r_4 which is a restricted channel. As for projection, the application is stuck. If there is a service channel s the associated service is invoked by sending the argument $\tilde{g}$ and the reply channel a along s. Note that the translation enforces the evaluation order as specified by the evaluation contexts of Piccola: first agent A is evaluated, then agent B is evaluated, then if the value of A has a service, the service is invoked.
- Translation of the sandbox term $A; B$ is very similar. First we communicate the value $\tilde{f}$ of agent A along a private channel r. Then we evaluate B in the context $\tilde{f}$.
- Extension $A \cdot B$ is translated similar. The values $\tilde{f}$ and $\tilde{g}$ are made available in sequence. We construct a form $\tilde{h}$ with the process $ext\langle \tilde{h}, \tilde{f}, \tilde{g} \rangle$ and return $\tilde{h}$ along a.
- Translation of the static root context $\mathbf{R}$ returns the tuple $\tilde{e}$ along a.
- A parallel agent $A \mid B$ is translated by the parallel composition of A and B which returns its value along a. The value of A is returned along a helper channel r. Once A is fully evaluated and emits a form $\tilde{f}$ along r, the process $\overline{r}\langle \tilde{f} \rangle$ can be garbage collected.
- The translation of channel restriction $\nu c.A$ is transparent since the channel c is a free name within the translation of A.
- Finally, receiving a form $\tilde{f}$ from channel c is done by the corresponding $L\pi$ prefix and forwarding $\tilde{f}$ along a.

The encoding relies on an correct use of continuation channels. We always pass the reply channel when we invoke another process. Such an approach in general violates fully abstract encodings of functions into processes. There might be a malicious client that uses the reply channel several times or that tries to read from it. In $L\pi$, a client cannot read from a received channel. Furthermore, we know by construction that reply channels are used at most once. This could be enforced by linear types [KPT96]. However, we do not enforce this on service channels.

4.5 Soundness of the Encoding

In this section we show that the encoding $[\![.]\!]$ is sound. We let it open to construct a complete encoding of Piccola into the $L\pi$ calculus. Most of the lemmas are stated for processes $[\![A]\!]_a$ instead of $[\![A]\!]_a^{\tilde{e}}$. Technically, this simplifies many of the proofs because we do not have to consider the environment. The only drawback is that $[\![A]\!]_a \approx [\![\epsilon; A]\!]_a$ thus any agent is assumed to live in a sandbox with the empty root. Our soundness theorem will therefore be stated with respect to closed agents. However, open agents are recovered by turning them into root abstractions, i.e., abstractions that expect the root context to be the argument: $\epsilon; \lambda x.(x; A)$.

Lemma 4.13 *The free names of translated forms and agents are:*

- $fc([\![A]\!]_a^{\tilde{f}}) \subseteq \{a, \tilde{f}\} \cup fc(A)$
- $fc([\![A]\!]_a) = \{a\} \cup fc(A)$
- $fc([\![F]\!]_a^{\tilde{f}}) = \{a\} \cup fc(F)$
- $fc([\![F]\!]_a) = \{a\} \cup fc(F)$

Proof. Checked by structural induction over A and F. □

The following lemma gives some evidence of how forms are encoded. The encoding of a form F instantiates a form process P that models F.

Lemma 4.14 *For any $F \in \mathcal{F}$ there exists a process $P \in \mathcal{F}_\mathcal{P}$ with $fc(P) = fc(F) \cup \{\tilde{f}\}$*

$$[\![F]\!]_a \gtrsim (\nu \tilde{f})\overline{a}\langle \tilde{f} \rangle \mid P.$$

Proof. By structural induction on F and using expansion. □

For the process P we usually write $P_F\langle \tilde{f} \rangle$ to indicate that the process P makes the form F available at the channels $\tilde{f}$. We say that $P_F\langle \tilde{f} \rangle$ models the form F.

We are now going to show that congruence is adequately encoded. If $A \equiv B$ then the corresponding encoding of agents are equivalent. We need some technical lemmas to modularize the proof for Lemma 4.17 which states the adequacy of congruence.

Lemma 4.15 is an adapted version of Lemma 4.10 of the $L\pi$ calculus. Lemma 4.15(1) says that in the input prefixed process $a(\tilde{f}).P \mid Q$ where only process Q "needs", the form $\tilde{f}$ the process P can be factored out. Lemma 4.15(2) is similar but now both processes P and Q require the value of the form F. In that case, we duplicate the process that encodes the form. Lemma 4.15(3) shows that the parallel composition operator of Piccola is transparent to Lπ.

Lemma 4.15 *For $a \notin fc(P \mid Q)$, $\tilde{f} \notin fc(P)$, and $\tilde{g}$ only free in output subject position in P and Q, we have:*

1. $(\nu a)[\![F]\!]_a \mid a(\tilde{f}).P \mid Q \approx P \mid (\nu a)[\![F]\!]_a \mid a(\tilde{f}).Q$
2. $(\nu a)[\![F]\!]_a \mid a(\tilde{g}).P \mid Q \approx ((\nu a)[\![F]\!]_a \mid a(\tilde{g}).P) \mid ((\nu a)[\![F]\!]_a \mid a(\tilde{g}).Q)$
3. $[\![A \mid B]\!]_a \mid a(\tilde{g}).P \equiv ((\nu r)[\![A]\!]_r) \mid [\![B]\!]_a \mid a(\tilde{g}).P$

Proof. (1) The proof uses the fact that forms are immediately available:

$$\begin{array}{rl}
(\nu a)[\![F]\!]_a \mid a(\tilde{f}).P \mid Q \gtrsim & \text{(by Lemma 4.14 } P_F \text{ exists)} \\
(\nu a, \tilde{g})\overline{a}\langle \tilde{g} \rangle \mid P_F \mid a(\tilde{f}).P \mid Q \gtrsim & \text{(expand } a\text{)} \\
(\nu \tilde{g})P_F \mid (P \mid Q)\{\tilde{f}/\tilde{g}\} \equiv & (a \notin fv(P)) \\
(\nu \tilde{g})P \mid P_F \mid Q\{\tilde{f}/\tilde{g}\} \lesssim & \text{(introduce } a\text{)} \\
(\nu a, \tilde{g})P \mid P_F \mid \overline{a}\langle \tilde{g} \rangle \mid a(\tilde{f}).Q \lesssim & \\
(\nu a)P \mid [\![F]\!]_a \mid a(\tilde{f}).Q \equiv & \\
P \mid (\nu a)[\![F]\!]_a \mid a(\tilde{f}).Q &
\end{array}$$

(2) By Lemma 4.14, F is modeled by $P_F\langle\tilde{g}\rangle$, we have $a \notin fv(P_F)$ and we have:

$$\begin{array}{rl}
(\nu a)[\![F]\!]_a \mid a(\tilde{g}).P \mid Q \gtrsim & \\
(\nu a,\tilde{g})\overline{a}\langle\tilde{g}\rangle \mid P_F\langle\tilde{g}\rangle \mid a(\tilde{g}).P \mid Q \gtrsim & \text{(expand } a\text{)} \\
(\nu\tilde{g})P_F\langle\tilde{g}\rangle \mid P \mid Q \approx & \text{(by Lemma 4.10(1))} \\
((\nu\tilde{g})P_F\langle\tilde{g}\rangle \mid P) \mid ((\nu\tilde{g})P_F\langle\tilde{g}\rangle \mid Q) \lesssim & \\
((\nu a,\tilde{g})\overline{a}\langle\tilde{g}\rangle \mid P_F\langle\tilde{g}\rangle \mid a(\tilde{g}).P) \mid ((\nu a,\tilde{g})\overline{a}\langle\tilde{g}\rangle \mid P_F\langle\tilde{g}\rangle \mid a(\tilde{g}).Q) = & \\
((\nu a)[\![F]\!]_a \mid a(\tilde{g}).P) \mid ((\nu a)[\![F]\!]_a \mid a(\tilde{g}).Q) &
\end{array}$$

(3) Apply the definition of $[\![A \mid B]\!]_a$. □

The next lemma shows some important properties of the extension operator. For short Lemma 4.16(1) says that extending a form to the right with the empty form does not change its behaviour. Part 2 says the corresponding for left extension. Part 3 proves that extension is associative. Lemma 4.16(4) confirms that the process $bind\langle\tilde{g},x,\tilde{f}\rangle$ indeed builds up the form $x{\mapsto}F$ as expected. Part 5 shows how to construct the root context for an agent A. The last part shows that an unnecessary form can be garbage collected. In the Piccola calculus, this is expressed by rule *discard zombie*.

Lemma 4.16 *For any form F let P_F be the process that models it.*

1. $(\nu\tilde{g},\tilde{h})ext\langle\tilde{f},\tilde{h},\tilde{g}\rangle \mid empty\langle\tilde{g}\rangle \mid P_F\langle\tilde{h}\rangle \approx P_F\langle\tilde{f}\rangle$
2. $(\nu\tilde{g},\tilde{h})ext\langle\tilde{f},\tilde{g},\tilde{h}\rangle \mid empty\langle\tilde{g}\rangle \mid P_F\langle\tilde{h}\rangle \approx P_F\langle\tilde{f}\rangle$
3. *For $R = P_F\langle\tilde{f}\rangle \mid P_G\langle\tilde{g}\rangle \mid P_H\langle\tilde{h}\rangle$ with $\tilde{f}',\tilde{g}',\tilde{e} \notin fv(R)$ it holds:*
   $$(\nu\tilde{f},\tilde{f}',\tilde{g},\tilde{h})R \mid ext\langle\tilde{f}',\tilde{f},\tilde{g}\rangle \mid ext\langle\tilde{e},\tilde{f}',\tilde{h}\rangle \approx (\nu\tilde{f},\tilde{g},\tilde{g}',\tilde{h})R \mid ext\langle\tilde{g}',\tilde{g},\tilde{h}\rangle \mid ext\langle\tilde{e},\tilde{f},\tilde{g}'\rangle$$
4. $[\![x{\mapsto}F]\!]_a \approx (\nu\tilde{f},\tilde{g})\overline{a}\langle\tilde{g}\rangle \mid bind\langle\tilde{g},x,\tilde{f}\rangle \mid P_F\langle\tilde{f}\rangle$
5. $[\![F;A]\!]_a \approx (\nu\tilde{f})P_F\langle\tilde{f}\rangle \mid [\![A]\!]_a^{\tilde{f}}$
6. $(\nu a)[\![F]\!]_a \approx \boldsymbol{0}$

Proof. 1 – 3 are proved by exhibiting an appropriate bisimulation up-to context and expansion. 4 – 6 are straightforward calculations. The complete proof is in Appendix C.2. □

We now prove correctness of the encoding with respect to structural congruence:

Lemma 4.17 *$A \equiv B$ implies $[\![A]\!]_a \approx [\![B]\!]_a$*

This is proved by structural induction on the derivation of $A \equiv B$. See Appendix C.3 for details. Notice that the translation $[\![A]\!]_a$ corresponds to $[\![\epsilon;A]\!]_a$ thus we assume both agents A and B live in an empty sandbox.

The following lemma states that thread contexts (see Definition 3.10) do not modify the reduction possibilities. In order to state the lemma we first define $[\![.]\!]$ to work on contexts as well. $[\![.]\!]$ is the extension of the definition given in Table 4.4 where a hole is translated to a hole. Thus $[\![.]\!]$ can be seen as a function from Piccola contexts to $L\pi$-contexts. We write $[\![\mathcal{C}[P]]\!]_a$ for the term $([\![\mathcal{C}[]]\!]_a)[P]$, thus we apply $[\![.]\!]$ as usual and in the translated context we replace the hole with process P. Notice that the fresh helper channels that are introduced may not occur in P.

Lemma 4.18 *If $P \xrightarrow{\mu} Q$ then there is a Q' with $\left[\!\left[\widehat{\mathcal{E}}[P]\right]\!\right]_a \gtrsim\xrightarrow{\mu} Q'$ and $\left[\!\left[\widehat{\mathcal{E}}[\mathbf{0}]\right]\!\right]_a \gtrsim \mathbf{0}$.*

Proof. By structural induction over $\widehat{\mathcal{E}}$ and using expansion for the cases $F \cdot \widehat{\mathcal{E}}$ and $F\widehat{\mathcal{E}}$. □

We are now going to prove operational correspondence. This correspondence says that whenever an agent A reduces to B, then the corresponding translation of A can also reduce to a process P which is congruent to the translation of B. It shows that a reduction of an agent A corresponds to one or several reductions of its translated process.

Lemma 4.19 *For any agent $A, B \in \mathcal{A}$: $A \rightarrow B$ implies $[\![A]\!]_a \rightarrow\Rightarrow\approx [\![B]\!]_a$.*

Proof. By Proposition 3.13 and Lemma 4.17 there exists a canonical agent A' such that:

$$[\![A]\!]_a \approx P = [\![A']\!]_a$$

and the process P has the structure:

$$\nu c_1 \ldots c_n . P_{M_1} \mid \ldots \mid P_{M_m} \mid P_{A_1} \mid \ldots \mid P_{A_k}$$

where P_{M_i} are translated messages and P_{A_j} are translated threads (See Definition 3.11). Thus $P_{M_i} = (\nu r)[\![M_i]\!]_r$ and $P_{A_j} = (\nu r)[\![A_j]\!]_r$ for $j < k$ and $P_{A_k} = [\![A_k]\!]_a$. Using the previous lemma and Proposition 3.15 there are two cases to consider:

- one of the threads is an application with a closure or the inspect or the project primitive. We only consider a closure. Now we have (Q is the body of the abstraction):

$$\begin{aligned}[\![(F;\lambda x.A)G]\!]_a &\gtrsim (\nu s, r_1, r_2, \tilde{f}, \tilde{g}, \tilde{e})P_F\langle\tilde{e}\rangle \mid \mathit{fun}\langle\tilde{f}, s\rangle \mid P_G\langle\tilde{g}\rangle \mid \overline{f_i}\langle r_1, r_2\rangle \mid r_1(s).\overline{s}\langle\tilde{g}, a\rangle \mid \\ &\quad !s(\tilde{g}, r).Q \\ &\gtrsim (\nu s, \tilde{g}, \tilde{e})P_F\langle\tilde{e}\rangle \mid P_G\langle\tilde{g}\rangle \mid \overline{s}\langle\tilde{g}, a\rangle \mid !s(\tilde{g}, r).Q \\ &\rightarrow (\nu s, \tilde{g}, \tilde{e})P_F\langle\tilde{e}\rangle \mid P_G\langle\tilde{g}\rangle \mid (\nu\tilde{h}, \tilde{e}')[\![A]\!]_a \mid \mathit{bind}\langle\tilde{h}, x, \tilde{g}\rangle \mid \mathit{ext}\langle\tilde{e}', \tilde{e}, \tilde{h}\rangle \\ &\approx [\![F \cdot x{\mapsto}G; A]\!]_a\end{aligned}$$

- one of the threads is a receiver $\widehat{\mathcal{E}}[c?]$ and one of the messages is cF. For simplicity, we assume there is only one message. We have that:

$$\begin{aligned}(\nu r)[\![cF]\!]_r \mid [\![c?]\!]_a &\gtrsim (\nu\tilde{f})P_F\langle\tilde{f}\rangle \mid \overline{c}\langle\tilde{f}\rangle c(\tilde{f}).\overline{a}\langle\tilde{f}\rangle \\ &\rightarrow (\nu\tilde{f})P_F\langle\tilde{f}\rangle \mid \overline{a}\langle\tilde{f}\rangle \\ &\approx [\![F]\!]_a\end{aligned}$$

  A similar calculation shows the same result for $F; c?$. □

The following lemma states the other direction of the above lemma. Whenever a process $[\![A]\!]_a$ reduces to a process P, it is either the case that P is congruent to $[\![A]\!]_a$ or that the reduction reflects a reduction on the agent side. In the first case, the reduction is an internal bookkeeping action that can be expanded. In the latter case, the agent A reduces to some agent B and $[\![B]\!]_a$ is congruent to P.

Lemma 4.20 *For any closed agent A: $[\![A]\!]_a \rightarrow P$ implies one of the following*

1. *B exists with: $A \equiv B$ and $P \approx [\![B]\!]_a$.*

2. *B exists with: $A \rightarrow B$ and $P \Rightarrow [\![B]\!]_a$.*

Proof. Case 1 is Lemma 4.17. Case 2 is shown by using a contradiction. Assume A is stuck and is written in canonical format. By Proposition 3.15 there are only threads of the following structure:

- $\widehat{\mathcal{E}}[\mathbf{R}]$ or $\widehat{\mathcal{E}}[\lambda x.A]$. This case is not possible since A is closed by assumption.
- $\widehat{\mathcal{E}}[FG]$ where F is a form without a service. If F is a form without a service, then the service selection feature of its encoding yields an error:

$$P_F\langle\tilde{f}\rangle \mid \overline{f_i}\langle p, q\rangle \Rightarrow P_F\langle\tilde{f}\rangle \mid \overline{q}$$

  Thus $[\![FG]\!]_a \approx \mathbf{0}$ and the term P cannot evolve.
- $\widehat{\mathcal{E}}[c?]$ and no message cF is in A. The only possible transition for $\widehat{\mathcal{E}}[c?]$ is an input action along c by Lemma 4.18. Since there are no matching output reductions, $\widehat{\mathcal{E}}[c?]$ is stuck. The same argument can be made for $\widehat{\mathcal{E}}[G; c?]$.

All cases contradict our assumption, therefore there must be a B with the required behaviour. □

Notice that the lemma is only stated for closed agents (see Definition 3.1). The lemma is not true for arbitrary agents. For instance

$$[\![\epsilon \cdot \mathbf{R}]\!]_a \rightarrow (\nu\tilde{h}\tilde{g}\tilde{f})empty\langle\tilde{g}\rangle \mid empty\langle\tilde{g}\rangle \mid ext\langle\tilde{h}, \tilde{g}, \tilde{f}\rangle \mid \overline{a}\langle\tilde{g}\rangle$$

but this process is congruent to $[\![\epsilon]\!]_a$. Thus $[\![\epsilon]\!]_a$ and $[\![\epsilon \cdot \mathbf{R}]\!]_a$ are equivalent. This is not surprising since the translation $[\![A]\!]$ puts the agent A into a context equivalent to the empty form. And in fact it holds $\epsilon; \epsilon \cdot \mathbf{R} \equiv \epsilon$.

Whereas the previous lemmas show the adequacy of reduction we also have to show the adequacy of barbs. The fact that an agent is a barb must also be characterized on its translation.

Lemma 4.21 *For any agent $A \in \mathcal{A}$: $A\downarrow$ implies $[\![A]\!]_a \Downarrow_a$.*

Proof. A is a barb. The translation of a barb is a process of the form:

$$\nu c_1 \ldots c_n . P_{M_1} \mid \ldots \mid P_{M_m} \mid P_{A_1} \mid \ldots \mid P_{A_{k-1}} \mid [\![F]\!]_a$$

where P_{M_i} are translated messages and P_{A_j} are processes of the form $(\nu r)[\![A_j]\!]_r$. By Lemma 4.14 the conclusion holds. □

The following lemma states the converse:

Lemma 4.22 *For any A closed, $[\![A]\!]_a \Downarrow_a$ implies $A \Downarrow$.*

Proof. Let $P\downarrow_a$ such that $[\![A]\!]_a \Rightarrow P$. By Lemma 4.20 there is a B with $P \approx [\![B]\!]_a$ and either $A \equiv B$ or $A \Rightarrow B$. Assume B is written in canonical form. Since $P\downarrow a$ the main agent of B must be a form, thus $B\downarrow$. If the main agent is not a form it is either a thread $\widehat{\mathcal{E}}[FG]$ where F has no service or it is a thread $\widehat{\mathcal{E}}[c?]$ and there is no message cF. In both cases we have $P\not\downarrow_a$. □

We now prove soundness. Soundness means that if the translations of two closed agents are congruent in the $L\pi$-calculus, then the agents are congruent themselves. If the translation is not sound, it cannot serve as an *implementation* of Piccola agents in terms of $L\pi$. The implementation is wrong since it might yield the same behaviour for different agents.

Proposition 4.23 (Soundness) *For closed agents A, B and channel a the congruence $[\![A]\!]_a \approx [\![B]\!]_a$ implies $A \approx B$.*

Proof. Let $\mathcal{S} = \{(A, B)$with A and B closed, and $[\![A]\!]_a \approx [\![B]\!]_b\}$. We show that $\mathcal{S}$ is a barbed bisimulation:

- Assume $A \rightarrow A'$. We show that there is a B' with $A' \approx B'$ and $B \Rightarrow B'$. By Lemma 4.19 we have $[\![A]\!]_a \rightarrow\Rightarrow [\![A']\!]_a$. Since $[\![A]\!]_a \approx [\![B]\!]_a$ there exists a P with $[\![B]\!]_a \Rightarrow P$ and since $\approx$ is an equality we have $P \approx [\![A']\!]_a$. Now there are two cases for $[\![B]\!]_a \Rightarrow P$:
  - $[\![B]\!]_a \equiv P$. In this case we simply take $B' = B$ for the required B'.
  - $[\![B]\!]_a \rightarrow\Rightarrow P$. By Lemma 4.20 we know that there is a B' with $B \rightarrow B'$ or $B' \equiv B$.
- Assume $A \downarrow$. We show that this implies $B \Downarrow$. By Lemma 4.21 we know that $[\![A]\!]_a \Downarrow_a$, which implies $[\![B]\!]_a \Downarrow_a$. By Lemma 4.22 this implies $B \Downarrow$.

Since $\mathcal{S}$ is symmetric the converse for the above holds trivially and $\mathcal{S}$ is a barbed bisimulation. Furthermore, by definition $\mathcal{S}$ is closed for any context. □

Soundness gives a proof technique for showing that certain closed agents are congruent. We will give a few example proofs in the following section.

An encoding is fully abstract if it is sound and complete [Mil75]. Our encoding is not fully abstract since it is not complete. Completeness would mean that if $A \approx B$ then also $[\![A]\!]_a \approx [\![B]\!]_a$. However, this is not the case for our encoding. The reason is that we can construct a context in $L\pi$ that can distinguish agents which are congruent in the Piccola calculus. The reason is that an observer can provide tuples as forms that implement malicious forms.

As an example, consider the agents:

$$\begin{aligned} A &= \mathbf{L}\, c?\ (\mathit{isEmpty}{\mapsto}S \cdot \mathit{isService}{\mapsto}S \cdot \mathit{isLabel}{\mapsto}S) \\ B &= c?; \epsilon \end{aligned}$$

where S is the service $\epsilon; \lambda x.\epsilon$, i.e., a service that always returns the empty form. Both agents make first an input on channel c and then yield the empty form. However, A inspects the received form. Thus an observer can provide a malicious form along c and detect a difference between A and B although they are congruent. An context that distinguishes between A and B provides a form along c and does not implement the inspect facility correctly. Consider the $L\pi$ context:

$$\mathcal{C} = [] \mid (\nu \tilde{f})\overline{c}\langle \tilde{f} \rangle$$

Now $\mathcal{C}[[\![B]\!]_a] \Downarrow_a$ but $\mathcal{C}[[\![A]\!]_a] \not\Downarrow_a$ since the inspection process $\overline{f_s}\langle p, q\rangle$ that appears within the translation of A does not lead to a continuation along neither p nor q.

For our purpose it is sufficient to have a sound encoding of Piccola agents. Since the encoding is compositional, we have a milder form of completeness: $A \approx B$ implies that its translations cannot be distinguished by contexts that are themselves translations of agents. Thus we can use the encoding to derive equalities of Piccola agents. But we cannot use the encoding to show properties of the $L\pi$ calculus, since we cannot arbitrarily switch between Piccola and $L\pi$.

4.6 Proving Laws for the Piccola Calculus

We use the encoding to prove laws for the Piccola calculus. As examples we show some of the laws mentioned at the end of Chapter 3.

Lemma 4.24 *For any forms F, G and label x it holds:*

$$F \cdot x{\mapsto}G; x \approx G$$

Proof. Since both expressions are closed agents we can compare their respective encoding in the π-calculus. We have:

$$\begin{aligned} [\![F \cdot x{\mapsto}G]\!]_a &\gtrsim (\nu \tilde{f}, \tilde{g}, \tilde{h}, \tilde{f}').P_F\langle\tilde{f}\rangle \mid P_G\langle\tilde{g}\rangle \mid bind\langle\tilde{h}, x, \tilde{g}\rangle \mid ext\langle\tilde{f}', \tilde{f}, \tilde{h}\rangle \mid \overline{a}\langle\tilde{f}'\rangle \\ [\![F \cdot x{\mapsto}G; x]\!]_{a'} &\approx [\![F \cdot x{\mapsto}G]\!]_a \mid a(\tilde{f}).\overline{f_p}\langle x, a', r\rangle \\ &\approx P_G\langle\tilde{g}\rangle \mid \overline{a'}\langle\tilde{g}\rangle \\ &\approx [\![G]\!]_{a'} \end{aligned}$$

and the conclusion follows by Proposition 4.23. □

The following law is the Piccola variant of deterministic communication. Since there is exactly one receiver of the message, we can replace

Lemma 4.25 *For any form F and thread context $\widehat{\mathcal{E}}$ it holds*

$$\nu c.cF \mid \widehat{\mathcal{E}}[c?] \approx \nu c.\widehat{\mathcal{E}}[F]$$

Proof. By induction over the thread context. The anchor is given by the expansion lemma of the π-calculus. The process $[\![c?]\!]_a$ has the format $c(\tilde{f}).P$ and the message cF is the process $(\nu\tilde{f})P_F\langle\tilde{f}\rangle \mid \overline{c}\langle\tilde{f}\rangle$ thus $\nu c.cF \mid c? \approx \nu c.F$. The induction step follows from Lemma 4.17. □

4.7 Discussion

In this chapter we have presented a sound encoding of the Piccola calculus into the π calculus. The encoding can be used to propagate properties from the π calculus to Piccola. These properties include laws that specify, for instance, when an agent is blocked on receiving from a channel and the channel is private, we can garbage collect the agent.

The encoding is also useful to reason about invariants of components encoded in Piccola. Consider the case of a reference cell. We can ensure the invariant that there is always one form *stored* in the private channel of the reference cell. We use this property for the encoding of fixed point in Piccola, see Section 5.5 in the following chapter.

Finally, it should be mentioned that working on the encoding gave feedback for defining the semantics of the Piccola calculus. In fact, in an early version of the Piccola calculus we have defined assignments as a binding-scope pair. The value of the assignment was the binding extended with the value of the scope. The scope was evaluated in a extended root context. It was due to the encoding of Piccola into the π calculus that we found the similarities that allowed us to remove assignment from the calculus and see it as a composition of sandbox, binding, and extension expression.

Chapter 5

A Small Composition Language

In this chapter we define the composition language Piccola on top of the calculus defined in Chapter 3. The language sits at the second Piccola composition layer (see Table 2.1). Speaking about "Piccola" we are often lazy and assume that the context makes it clear whether we mean the calculus or the language. The main difference of the Piccola language with respect to other programming languages is that Piccola does not contain any primitive or built-in data-types except forms and channels. The semantics of other values (for instance for arithmetic computation) is defined by external components.

With respect to the Piccola calculus, the language provides the following changes to ease programming:

- Parallel composition, channel restriction as well as sending and receiving along channels are available in the language via *predefined services*. This simplifies the syntactical structure of programs and makes the language more uniform.
- The definition of *recursive* forms is directly supported.
- Piccola provides *user-defined operators* constructed by special characters. Operators are either defined on individual forms or they can be specified for a given context. Infix and prefix operators are useful to support an algebraic notation for plugging components in a composition style.
- Piccola defines syntax for *user-defined collections*. It is more convenient to build collections surrounded by brackets than to assemble them by a purely functional notation.
- Service invocations carry a *dynamic namespace* where the concrete argument form is just one part of it. This makes it possible to introduce dynamically scoped contexts.
- An expression in Piccola is a *sequence* of form expressions and bindings. Bindings not only define a binding, but they also extend the root context for subsequent expressions. This allows the programmer to use bindings as assignments and simplifies the definition of services. Parentheses and indentation group expressions.
- *Nested form bindings* can be simplified to a single binding.
- The language provides syntax for *projections* with a high precedence whereas sandbox expressions have low precedence.

The following diagram helps to structure the rest of this chapter:

$$\begin{tikzcd}
\text{Piccola Language, Terms, Section 5.2} \arrow[d, "\text{Abbreviation, Section 5.3}"'] \\
\text{Simplified Language} \arrow[d, "{[\![.]\!]}\text{-semantics, Section 5.5}"'] \\
\text{Functional agents, Section 5.4} \\[-2em]
\text{and Forms, Chapter 3}
\end{tikzcd}$$

In Section 5.1 we give an introduction to the programming model that is behind Piccola. In Section 5.2 we define the syntax of the language. In Section 5.3 we present a set of syntactical sugar features, for instance to define operators and collections. This leads to a simplified language. In Section 5.4 we define functional Piccola agents. Functional agents are a subset of agents with primitive services to create channels and span new agents. In Section 5.5, the semantics of Piccola is given by a translation of simplified terms to functional agents. In Section 5.6 we define the initial root context containing services to create channels, inspect forms, and spawn off agents. In Section 5.7 we present an alternative encoding of fixed points by using combinators known from the lambda-calculus. In Section 5.8 we discuss how external components are accessed from Piccola. In Section 5.9 we briefly discuss aspects of the design rational for the Piccola language.

5.1 Programming with Explicit Environments

One of the surprising things when programming in a language with explicit environments is that the notion of a variable gets somewhat blurred. The reason is that variables and environments are not only a question of scopes but they are values that can be manipulated. The same happens in meta-programming when we have access to the state of the running program.

To understand programming with explicit environments we contrast the situation with the language C where the term `a = 1` is a statement and an expression. The value of the expression is 1. The effect of the statement is that the value 1 is assigned to the variable `a`. A statement `b = a = 1` assigns the value 1 to `a` and `b`.

In Piccola statements do not exist. The term `a = 1` has the value of the singleton binding $a\mapsto 1$. The flavour of a statement is achieved by extending the current root context with the binding and evaluating subsequent expressions in the extended root context. We say "extending the root" for this process. Consequently, the expression `b = a = 1` denotes the form with a binding $b\mapsto(a\mapsto 1)$. We write Piccola language terms in `typewriter font` and values as agent expressions (see Table 3.1).

We can think of Piccola as a two slot machine. One slot keeps the value to be returned as the value of the expression being evaluated; the other slot stores the current root context. There are three distinguished ways how the value of an expression is used. (1) If the machine encounters a *binding* it extends the current value and the root context, (2) if it encounters a *quoted expression* the root context only is extended, and (3) if it encounters any other *expression* the current value only is extended. All lookups of variables are projections in the current root context. Unless the form expression is completely evaluated, the machine cannot read the actual value. When a new expression is evaluated, we start with the empty form as initial value and inherit the root context from the lexical scope.

As an example consider the following script.

```
a =
    'x = 1
    y = x
b =
    y = 2
    z = y
a
b
```

When we evaluate this script in an initially empty root context we get:

expression	value	root
`a =`	ϵ	ϵ
`  'x = 1` *# quoted*	ϵ	$x{\mapsto}1$
`  y = x` *# binding*	$y{\mapsto}1$	$x{\mapsto}1, y{\mapsto}1$
	$a{\mapsto}(y{\mapsto}1)$	$a{\mapsto}(y{\mapsto}1)$
`b =`	ϵ	$a{\mapsto}(y{\mapsto}1)$
`  y = 2`	$y{\mapsto}2$	$a{\mapsto}(y{\mapsto}1) \cdot y{\mapsto}2$
`  z = y`	$y{\mapsto}2 \cdot z{\mapsto}2$	$a{\mapsto}(y{\mapsto}1) \cdot y{\mapsto}2 \cdot z{\mapsto}2$
	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(y{\mapsto}2 \cdot z{\mapsto}2)$	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(y{\mapsto}2 \cdot z{\mapsto}2)$
`a` *# expression*	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(\ldots) \cdot y{\mapsto}1$	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(y{\mapsto}2 \cdot z{\mapsto}2)$
`b`	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(\ldots) \cdot y{\mapsto}2 \cdot z{\mapsto}2$	$a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(y{\mapsto}2 \cdot z{\mapsto}2)$

Thus the value denoted by the above script is $a{\mapsto}(y{\mapsto}1) \cdot b{\mapsto}(y{\mapsto}2 \cdot z{\mapsto}2) \cdot y{\mapsto}2 \cdot z{\mapsto}2$.

When we evaluate a binding the label is not yet captured by the value. For instance in

```
x = 1
x = x + 1
```

the second binding yields $x{\mapsto}2$ which is also the value of the whole expression. If we need to specify recursive services and forms, we can prefix the binding with the keyword **def**.

5.2 The Language

We now define the syntax of the Piccola language. The language does not contain syntactical primitives for communication along channels, for spawning off new agents, and for hiding labels. These features are made available as predefined services in the initial root context.

5.2.1 Abstract Syntax

The abstract syntax of Piccola is given in Table 5.1. The grammar is a set of productions that describes how form expressions are constructed.

The piccola keywords **def** and **root**, the symbols backslash (\), colon (:), dot (.), round parentheses (()), comma (,), equal (=), and the quote sign (′) are terminal symbols. Nonterminal symbols are shown in *Italic*. Optional parts are written in square brackets [...].

Form ::=	
root	*current namespace*
identifier	*label*
literal	*constant literal*
\ [*Param*] : *Form*	*anonymous service*
Form . *identifier*	*projection*
Form Form	*application*
Form op Form	*infix application*
op Form	*prefix application*
Form , *Form*	*extension*
$op^{\{}$ [*FormList*] $op^{\}}$	*collection*
([*Form*])	*parentheses*
root = *Form* [, *Form*]	*sandbox*
[**def**] *Label* [*Param*] : *Form* [, *Form*]	*service binding*
[**def**] *Label* = *Form* [, *Form*]	*binding*
′ *Form* [, *Form*]	*quote*
FormList ::=	
[*FormList* ,] *Form*	*collection composition*
Param ::=	
identifier [*Param*]	
([*identifier*]) [*Param*]	
Label ::=	
identifier	*simple label*
Label . *identifier*	*nested label*

Table 5.1: Piccola Language Syntax

Precedence Category		Concrete Syntax
9R	prefix	*op Form*
8L	projection	*Form . identifier*
	tight invocation	*Form*(*Form*)
7L	arithmetic high	*Form op Form*
6L	arithmetic low	*Form op Form*
5L	comparison	*Form op Form*
4L	other op	*Form op Form*
3L	invocation	*Form Form*
2R	service	\\[*Param*] : *Form*
	binding	[**def**] *Label* = *Form*
	service binding	[**def**] *Label* [*Param*] : *Form*
	sandbox	**root** = *Form*
	quote	' *Form*
1L	collection composition	*FormList* , *Form*
1R	binding sequence	[**def**] *Label* = *Form* , *Form*
	service binding sequence	[**def**] *Label* [*Param*] : *Form* , *Form*
	sandbox sequence	**root** = *Form* , *Form*
	quote sequence	' *Form* , *Form*
	extension	*Form* , *Form*

Table 5.2: Precedence Rules

The most important class of terms are Piccola form expressions. These expressions evaluate to a form. Constant *literals* are numbers and strings. Strings are enclosed in double quotes (") or in triple douple quotes. The first version interprets escaped character, the second version does not. Normal *identifiers* start with an alphanumeric character and are followed by a sequences of numeric, alphanumeric and underscore characters. Special identifiers start with an underscore or an operator character, are followed by a sequence of alphanumeric, numeric and operator characters, and end with an underscore possibly followed by a sequence of alphanumeric characters. User-defined operators are denoted by special identifiers. The idea is that the underscore is a placeholder for an expression. For instance, the identifier for the + infix operator is `_+_`, its default label `_+_default`. *User-defined operators* are sequences of the characters: `* / + - = < > ! % : ; ~ ^ $ | ? &` and `@`. Alternatively, an operator can be an identifier written in backquotes, like `‘mod‘`. Collections are encloded by tokens $op^{\{}$ and $op^{\}}$ that match. These tokens are sequences of `{` or `[` and `}` or `]`, respectively. They match if their individual characters match in reverse order. For instance, `[{{` matches with `}}]`.

5.2.2 Precedence Rules

The precedence rules for Piccola are given in Table 5.2. Each syntactical category has a precedence and an associativity. For instance the application "`a b`" has precedence 3 and is left-associative. Subterms may only have higher precedence. The precedence of an infix expression is given by the first character of the infix operator. There are four groups of precedence:

	Group	First character
7L	Arithmetic high	`* /`
6L	Arithmetic low	`+ -`
5L	Comparison	`= < > !`
4L	Other	`% : ; ~ ^ $ | ? & @` *identifier*

As an example, the expression "`a b +> c`" is parsed as "`a (b +> c)`" because arithmetic low (defined by the + character) has a precedence of 6 which is higher than the precedence of invocation which in turn is 3. If the precedence is left or right associative, then the left or right subterm may have the same precedence. The expression "`a b c`" is parsed as "`(a b) c`" because invocation is left associative.

Note that infix operators have higher precedence than invocation. Therefore the expression "`a + b`" is read as infix arithmetic expression and not as "`a (+ b)`".

There are two different precedence for invocation: tight invocation (8L) and normal invocation (3L). For tight invocation, the argument must be enclosed in parentheses or collection brackets and must immediately follow the functor with no whitespace in between. For instance the expression "`a b(c)`" is parsed as "`a (b c)`", whereas "`a b (c)`" is parsed as "`(a b) c`". The motivation to distinguish normal and tight invocation is driven by the desire to write code as

```
a().b(c = x).d    instead of  ((a()).b(c = x)).d and
a b c             instead of  a(b)(c).
```

The precedence rules are strict. This means that they do not only rule out ambiguities but they also forbid certain constructs and force the programmer to use parentheses or indentation. For instance the expression "`a b=()`" is syntactically valid when parsed as "`a (b=())`". However, such a parse is not permitted as the expression "`b=()`" has precedence 2 which cannot be a subterm of invocation with precedence 3. Such an expression must be entered using parentheses or indentation.

The associativity induced by a comma is different when the comma appears as top-level operator in an expression sequence or within collection brackets.

Collection. A comma appearing top-level inside collection brackets denotes an expression *FormList, Form* with precedence 1L. For instance, the expression "`[a, b = x, c]`" is parsed as "`[(a, (b = x)), c]`". This means that the collection will contain three elements: the value of "`a`", of "`b = x`", and that of "`c`".

Note that by using parentheses the meaning of the comma changes: The collection "`[a, (b = x, c)]`" contains two elements, namely "`a`" and the term "`b = x, c`".

Sequence. A comma appearing in a sequence of form expressions has precedence 1R. If the left hand side of a comma is a binding, a service binding, a sandbox or a quote expression, then the right-hand side is the *scope* of the left hand side. In these cases the value of the left-hand side extends the current root context for the scope. For instance the expression

```
a = 1, 'b, c
```

is parsed as "`(a = 1, ('b, c))`", i.e., the expression "`'b, c`" is in the scope of the binding "`a = 1`" and the expression "`c`" is in the scope of "`b`".

5.2.3 Indentation

Piccola supports indentation and newlines instead of parentheses and commas to group form expressions. For example the term "`x = f(a = (), b = a), y`" is normally written as

```
x = f
    a = ()
    b = a
y
```

When a line starts at a higher or lower indentation than the previous line, an opening parentheses (indent) or a closing parentheses (dedent) is inserted, respectively. If the new line starts on the same indentation level, a comma is inserted. Inserted dedents may not mix matching parentheses, brackets or other indent-dedent pairs. Therefore, one or multiple dedent tokens are inserted before any closing parentheses, bracket or dedent, if a corresponding indent was inserted but not closed between the matching pairs. The precise rules are as follows. Assume line n has indentation level d. The following line has indentation level d'.

Indent. If $d < d'$ an indent with indentation level d' is inserted unless line n ends with an opening parentheses or bracket or the following line starts with a dot.

Comma. If $d = d'$, a comma is inserted unless line n ends with a comma, an operator, an opening parentheses or bracket, or the following line starts with a dot.

Dedent. If $d > d'$ and the following line does not start with a closing parentheses or bracket then closing dedents to match previously inserted indents are inserted until there is a remaining unmatched opening parentheses, bracket, or an indent with a lower indentation level. A comma is inserted unless the following line starts with a dot.

Closing. Dedents are inserted before any closing parentheses or bracket if there are unmatched indents inserted after the matching parentheses or bracket that gets closed. If dedents are inserted, the next line must not have an indentation level higher than the last inserted dedent.

End. At the end of an input, as many dedents are inserted as there are remaining unmatched indents inserted.

The precedence of invocations with an indented argument have normal precedence (8L). For example

```
a b
    c
```

is parsed as "`(a b) c`".

The indentation restricts the programmers freedom to insert newlines at will. For instance a newline cannot occur after an equals sign unless the value of the binding is indented or put into parentheses. For instance

```
a =
b
c
```

is tokenized as "`a = , b , c`" which is syntactically wrong.

$$[\,\mathbf{def}\,]\ L\ [\,P\,] : T \equiv [\,\mathbf{def}\,]\ L = \backslash[\,P\,] : T \tag{abb-sd}$$
$$\backslash I\ P : T \equiv \backslash I : \backslash P : T \tag{abb-curry1}$$
$$\backslash([\,I\,])\ P : T \equiv \backslash([\,I\,]) : \backslash P : T \tag{abb-curry2}$$
$$\backslash I : T \equiv \backslash(I) : T \tag{abb-paren}$$
$$\backslash : T \equiv \backslash() : T \tag{abb-void}$$
$$[\,\mathbf{def}\,]\ L\,.\,I = T \equiv L = (L\,,\,[\,\mathbf{def}\,]\ I = T) \tag{abb-nest}$$
$$[\,\mathbf{def}\,]I = T_1, T_2 \equiv\ '([\,\mathbf{def}\,]I = T_1), (I = I), T_2 \tag{abb-assign}$$
$$'\,T_1\ [\,,T_2\,] \equiv \mathbf{root} = (\mathbf{root}, T_1)\ [\,,T_2\,] \tag{abb-quote}$$
$$\mathbf{root} = T \equiv \mathbf{root} = T, () \tag{abb-sandbox}$$
$$T_1\ op\ T_2 \equiv ('(x = T_1), op^{infix}\ (y){:}\ (op^{defaultinfix}\ x\ y), x).op^{infix}\ T_2 \tag{abb-infix}$$
$$op\ T \equiv ('(x = T), op^{prefix}\ (){:}\ (op^{defaultprefix}\ x), x).op^{prefix}\ () \tag{abb-prefix}$$
$$op^{\{}\ op^{\}} \equiv op^{\{\}}\ () \tag{coll-empty}$$
$$op^{\{}\ [\,T_1,\,]\ T_2\ op^{\}} \equiv (op^{\{}\ [\,T_1\,]\ op^{\}}\ )\ .\ \mathtt{add}\ T_2 \tag{coll-add}$$

Table 5.3: Piccola Language Abbreviations

Note that no indents are inserted if the previous line ends with an opening bracket. For example the code

```
a = [
  1
  2]
```

is read as "`a = [1, 2]`". This collection has two elements, whereas "`a = [(1, 2)]`" denotes a collection with one element, namely 1 extended with 2.

5.3 Abbreviations

Many of the features of the Piccola language are syntactical sugar. In Table 5.3 the expansion for these features are given. We use T to range over form expressions, P to range over parameter expressions and L to range over label expressions. Amongst other simplifications, the abbreviations define the semantics for user-defined operators and collections.

5.3.1 Services

The rule *abb-sd* allows the programmer to define abstractions and bind them to an identifier in a single expression. For example

```
id x: x
```

is syntactic sugar for: `id = \x: x`. Observe that the name of the service `id` is not visible in the body of its binding. For recursive services we use the **def** keyword.

The rules *abb-curry1* and *abb-curry2* allow the programmer to write curried functions more user friendly. Instead of

```
\l: \r: l + r
```

we usually write

```
\l r: l + r
```

The rules *abb-paren* and *abb-void* allow us to omit parentheses and formal parameters in parameter expressions. For instance in the example

```
if n < 2
  then: 1
  else: n * fac(n - 1)
```

the bindings for label `then` and `else` expand to:

```
if n < 2
  then = \(): 1
  else = \(): n * fac(n - 1)
```

5.3.2 Nested Bindings

The rule *abb-nest* specifies the semantics of bindings with a nested label. A nested binding "`a.b = c`" extends the form denoted by `a` with the binding "`b = c`". This is achieved by writing "`a = (a, b = c)`". This process can be repeated to unfold the complete structure of the nested label.

Observe that an expression `Label.x = T` is only valid when the current root context contains a binding for `Label`. This is due to the fact that the nested binding is translated to a binding where the right-hand side value is an extension of `Label` with a normal binding. For instance the following code is invalid

```
a = ()                       # a will be the empty form
a.b.c = ...                  # wrong!
```

since the form `a.b` is not defined.

The rule *abb-nest* defines an inner fixed point when used with the **def** keyword. The term "`def f.b = c`" is equivalent to

```
f =
    f
    def b = c
```

whereas an outer fixed point would be:

```
def f =
    f                        # f used while being defined. Wrong!
    b = c
```

Such a code is illegal since it would denote an infinite form (refer to Section 5.5.2 for more details on fixed points).

5.3.3 Assignment

The rule *abb-assign* allows us to rewrite binding assignments with a nonempty scope. The behaviour of an assignment "`l = a, b`" is that the resulting value will contain a binding $l{\mapsto}a$ (where a is the value of "`a`") extended with the value of "`b`". The term "`b`" is evaluated in a context which contains $l{\mapsto}a$. This is achieved by writing

```
'(l = a)
(l = l)
b
```

Note that the quoted expression evaluates "`a`" and extends the root context. The following example illustrates the difference between assignment and extension:

```
a =
    l = 1               # assignment
    println l           # prints 1
    (l = 2)             # extension, does not change the root environment
    println l           # still prints 1
println a               # prints (l = 2)
```

5.3.4 Quoted Expressions

We extend the root context with the value of an expression "`a`" by:

```
root = (root, a)
...
```

Since such constructs are very frequent we have a special notation using quotes. By rule *abb-quote* the above code is abbreviated as:

```
'a
...
```

The quote construct is often used for local definitions. Its bindings are not exported.

The rule *abb-sandbox* defines the empty form as the scope for sandbox expressions without scope. The value of a quoted expressions is the empty form. For instance:

```
x = 'a
```

is syntactic sugar for "`x = (root = (root, a), ())`". The value of the whole expression is the binding $x{\mapsto}\epsilon$.

These rules explain the behaviour of the following idiom of using two quotes in Piccola. For instance, "`''extern(), ...`" is an abbreviation for

```
root =
    root
    (root = (root, extern()), ())
...
```

This code evaluates `extern()` and extends the root context with the empty form. This means that the result of `extern()` is not used, the application is evaluated for its side effect only. We also use this idiom if the application is known to return the empty form to make the code more self-documenting.

5.3.5 User Defined Operators

The infix expression "`a + b`" is syntactic sugar for

```
(
    '(x = a)
    (_+_ y: DefaultOp._+_default x y)          # default +
    x
)._+_ b
```

The behaviour of this term is as follows. Assume the expression "`a`" evaluates to a form *a* that contains a binding _+_$\mapsto S$. In this case the projection on the last line denotes the service *S* and the expression is equivalent to

```
a._+_ b
```

The infix operator is dispatched on its left-hand expression.

Now, assume *a* has no binding for the label `_+_`. In this case the projection sees the service defined as default and the infix-expression is equivalent to

```
DefaultOp._+_default a b
```

The order of evaluation is in both cases the same: First the left-hand expression is evaluated then the right-hand expression is evaluated and finally the service of the operator is applied. We use the label `DefaultOp` to contain global defaults for user-defined operators.

A similar expansion works for prefix operators. The term "`+a`" behaves as "`a.+_()`" when the form *a* contains the label `+_` and as "`DefaultOp.+_default a`" otherwise.

5.3.6 Collections

The semantics of user-defined collections in Piccola is specified by the rules *coll-empty* and *coll-add*. They work as follows. For each user defined collection there is a global factory in `DefaultOp`. The code "`x = { }`" is syntactic sugar for

```
x = DefaultOp.{_}()
```

It should be noted that `DefaultOp.{_}` is not a collection itself, but a factory to create new (empty) collections. Individual elements are added to such a collection using its `add` service. The term "`{a, b, c}`" is syntactic sugar for

```
DefaultOp.{_}().add(a).add(b).add(c)
```

Observe that a collection, i.e., the form returned by the factory must contain a service `add` which in turn returns the collection with the added element. In Section 6.1 we will see an example of an external component that is wrapped to fit the collection notation.

Recall from Section 5.2.2 that the comma separates the elements of the collection. As such, the bindings do not behave like assignments. For instance

```
a = 1                # Assignment
x = [[
  a = 2              # inside collection
  b = a]]
```

A, B, C	$::=$	$\mathbf{L}$	*inspect*	$\mid$	$\mathbf{new}$	*new Channel*
	$\mid$	$\mathbf{run}$	*run*	$\mid$	$hide_x$	*hide*
	$\mid$	ϵ	*empty form*	$\mid$	$\mathbf{R}$	*(static) root*
	$\mid$	$A; B$	*sandbox*	$\mid$	$A \cdot B$	*extension*
	$\mid$	x	*variable*	$\mid$	$x \mapsto A$	*binding*
	$\mid$	$\lambda x.A$	*abstraction*	$\mid$	AB	*application*

Table 5.4: Functional Piccola Agents

$$
\begin{array}{rclrcl}
[\![\mathbf{new}]\!]^f & = & \epsilon; \lambda x.\nu c.send \mapsto c \cdot receive \mapsto (\lambda y.c?) & [\![\mathbf{run}]\!]^f & = & \epsilon; \lambda x.(x() \mid \epsilon) \\
[\![\mathbf{L}]\!]^f & = & \mathbf{L} & [\![hide_x]\!]^f & = & hide_x \\
[\![\epsilon]\!]^f & = & \epsilon & [\![\mathbf{R}]\!]^f & = & \mathbf{R} \\
[\![A; B]\!]^f & = & [\![A]\!]^f; [\![B]\!]^f & [\![A \cdot B]\!]^f & = & [\![A]\!]^f \cdot [\![B]\!]^f \\
[\![AB]\!]^f & = & [\![A]\!]^f [\![B]\!]^f & [\![x]\!]^f & = & x \\
[\![x \mapsto A]\!]^f & = & x \mapsto [\![A]\!]^f & [\![\lambda x.A]\!]^f & = & \lambda x.[\![A]\!]^f
\end{array}
$$

Table 5.5: Embedding functional agents

is syntactic sugar for

```
    a = 1
    x = DefaultOp.[[_]]().add(a = 2).add(b = a)
```

The root context is the same for both elements that are added to the new collection. The collection x contains the bindings $a \mapsto 2$ and $b \mapsto 1$.

5.4 Functional Piccola Agents

The abbreviations rules of Section 5.3 allow us to rewrite Piccola terms into terms of a simpler language. With respect to the syntax of Table 5.1, we do not have to deal with user defined operators and collections, with curried services, quotes, assignments, and nested labels.

In order to give the semantics of the Piccola language we use an intermediate representation of terms called *functional Agents* see Table 5.4. We use the letters A, B, C to range over functional agents. They are called agents because they can be trivially embedded into agent expressions of the Piccola calculus. They are called functional because the functional subset of agents is considered only. The concurrent facet of Piccola is reified by the primitive services **run** and **new**. Finally, to simplify reading and writing the functional agents, we have singleton bindings as agents. To omit writing parentheses, binding has stronger precedence than application.

The embedding of functional Piccola agents A into the Piccola calculus is given by the semantical function $[\![.]\!]^f$ as in Table 5.5. The interpretation $[\![A]\!]^f$ is compositional. Primitive services are encoded appropriately. We normally omit the writing of $[\![A]\!]^f$ and identify functional and normal agents. This is why we use the same meta-variables A for both. The primitives **run** and **new** can then be considered syntactic sugar for agents.

Invoking the service **new** returns a form with two services *send* and *receive*. The two services share a restricted channel c. Applying the *send* service on a form F sends the value F along the channel. The result of the application *send* F is the empty form. This is due to the structural rule *emit*:

$$cF \equiv cF \mid \epsilon$$

The *receive* service receives a value from c. Invoking *receive* blocks unless a value is sent along c. Arbitrary form is returned when multiple values are sent along the channel.

The channel c is restricted. As a consequence, the following term is equivalent to the null agent:

$$[\![\mathbf{new}(); \mathit{receive}()]\!]^f \approx \mathbf{0}$$

since no agent ever gets access to c.

Invoking the primitive service **run** with an argument x yields in the creation of a new running agent executing $x()$. The empty form ϵ is returned by **run** x.

5.5 Semantics

The differences between functional Piccola agents and the simplified Piccola language are the fixed-point construct and the dynamic namespace. In this section we give the denotational encoding of (simplified) Piccola terms into agents. The semantics of Piccola is given by a mapping $[\![.]\!]$ from language terms to agents. The agent A corresponding to a Form expression T is:

$$A = \left[\!\!\left[[\![\mathtt{T}]\!]^f \right]\!\!\right]$$

or simply $A = [\![\mathtt{T}]\!]$.

The function $[\![.]\!]$ is recursively defined for all Piccola language terms, see Table 5.6. The semantics of fixed points (by the **def**) requires a reference cell *Fix* and a helper function $\mathit{lookup}[x].A$. The fixed-point cell *Fix* stores the value of the fixed point and $\mathit{lookup}[x].A$ is the agent A where all references x are replaced by lookup-calls to get the *value* of the fixed point. Observe that the range of $[\![.]\!]$ consists of functional agents only. Thus it is enough to define $\mathit{lookup}[x].A$ only for functional agents A.

In the rest of this section we discuss the definition of dynamic namespaces and of fixed point in more detail.

5.5.1 Dynamic Namespace

The dynamic namespace is a form that is passed on service invocations from the caller to the callee. The form is bound with the label *dynamic*. An important part of the dynamic namespace is the actual argument. By rule *sem inv* we convert all actual arguments T into $\mathit{dynamic} \cdot \mathit{arg}\mapsto[\![\mathtt{T}]\!]$. Thus the client always passes its dynamic context together with the actual value bound by `arg` to the server.

Contrary, any service definition has to conform to this protocol. We translate service definitions (rule *sem abs* and rule *sem void*) into abstractions that expect the dynamic context. When the service contains a formal argument, we extend the root context with an appropriate binding for the actual argument.

$$\llbracket \textbf{root} \rrbracket = \mathbf{R} \quad \text{(sem root)}$$
$$\llbracket T, T \rrbracket = \llbracket T \rrbracket \cdot \llbracket T \rrbracket \quad \text{(sem ext)}$$
$$\llbracket \textbf{root} = T_1, T_2 \rrbracket = \llbracket T_1 \rrbracket ; \llbracket T_2 \rrbracket \quad \text{(sem sandbox)}$$
$$\llbracket I = T \rrbracket = I {\mapsto} \llbracket T \rrbracket \quad \text{(sem bind)}$$
$$\llbracket T_1\ T_2 \rrbracket = \llbracket T_1 \rrbracket (\mathit{dynamic} \cdot \mathit{arg} {\mapsto} \llbracket T_2 \rrbracket) \quad \text{(sem inv)}$$
$$\llbracket \backslash ()\colon T \rrbracket = \lambda\ \mathit{dynamic}. \llbracket T \rrbracket \quad \text{(sem void)}$$
$$\llbracket \backslash (I)\colon \mathrm{T} \rrbracket = \lambda\ \mathit{dynamic}.(\mathbf{R} \cdot I {\mapsto} (\mathit{dynamic}; \mathit{arg}); \llbracket T \rrbracket) \quad \text{(sem abs)}$$
$$\llbracket \textbf{def}\ I = T \rrbracket = \mathbf{R} \cdot I {\mapsto} \mathit{Fix}; (I; \mathit{set})(I {\mapsto} \mathit{lookup}[I].\llbracket T \rrbracket) \quad \text{(sem def)}$$

where *Fix* is

$$\mathit{Fix} = \begin{array}{r} \textbf{new}(); \mathit{send}() \cdot \mathit{set} {\mapsto} \lambda x.(\mathbf{R} \cdot d {\mapsto} \mathit{receive}(); (\mathit{send}\ x) \cdot x) \cdot \\ \lambda x.(\mathbf{R} \cdot d {\mapsto} \mathit{receive}(); (\mathit{send}\ d) \cdot d) \end{array}$$

and $\mathit{lookup}[x].A$ is inductively defined for any expression as follows. We assume that $x \neq y$ in the tagged equations (*).

$$\begin{array}{rlrll}
\mathit{lookup}[x].x = & x(); x & \mathit{lookup}[x].y = & y & (*) \\
\mathit{lookup}[x].\lambda x.A = & \lambda x.A & \mathit{lookup}[x].\lambda y.A = & \lambda y.(\mathbf{R} \cdot x {\mapsto} \mathit{lookup}[x].x; A) & (*) \\
\mathit{lookup}[x].\epsilon = & \epsilon & \mathit{lookup}[x].\mathbf{L} = & \mathbf{L} & \\
\mathit{lookup}[x].\textbf{new} = & \textbf{new} & \mathit{lookup}[x].\textbf{run} = & \textbf{run} & \\
\mathit{lookup}[x].\mathit{hide}_y = & \mathit{hide}_y & \mathit{lookup}[x].A \cdot B = & (\mathit{lookup}[x].A) \cdot (\mathit{lookup}[x].B) & \\
\mathit{lookup}[x].(y {\mapsto} A) = & y {\mapsto} (\mathit{lookup}[x].A) & \mathit{lookup}[x].(AB) = & (\mathit{lookup}[x].A)\ (\mathit{lookup}[x].B) & \\
\mathit{lookup}[x].(A; B) = & r {\mapsto} \mathit{lookup}[x].A \cdot x {\mapsto} x; & \mathit{lookup}[x].\mathbf{R} = & \mathit{hide}_x \mathbf{R} & \\
 & r \cdot x {\mapsto} \lambda d.x() \cdot r; & & & \\
 & \mathit{lookup}[x].B & & &
\end{array}$$

Table 5.6: Translating simplified Piccola terms to expressions

Services with no formal parameter may access the concrete argument by looking at the value of `dynamic.arg`. For instance, the following script prints out the number 5:

```
f: println dynamic.arg          # the service f prints its argument
f 5                             # invoke f
```

The nested form `dynamic.arg` is not forwarded when a service is invoked with the empty form as argument. This is due to the fact that the argument binding is overwritten by every invocation.

Observe that the primitive services must be consistent with this protocol. We have to wrap those services accordingly (see Section 5.6). For instance in

$$\mathbf{new}(); mySend{\mapsto}\lambda d.send(d; arg); mySend(arg{\mapsto}F) \approx \nu c.(cF \mid \epsilon)$$

we use *mySend* as the sender service to send the value F along the newly created channel.

5.5.2 Fixed Points

Forms can be thought of as finite trees. It is not possible do specify a form which contains itself as a nested form. However, we can specify a form that has a service containing itself. Examples are recursive services and methods in objects that refer to *self*. In the following we explain the semantics of **def** expressions that is used to simplify the definition of fixed points. We expect that such fixed points permit the following:

Mutual recursion. Fixed points may be used for writing mutually recursive services.

Object encodings. The fixed point construct must be useful for directly specifying an object model that supports inheritance.

These are pragmatic requirements. It may be possible that a fixed point is needed to meet certain other criterion. In this case fixed points may be constructed manually by using channels. In Section 5.7.3 we give an example how this might be done. In Section 7.8 we present mixin-composition to demonstrate the usefulness of the fixed point to model inheritance. We now explain the behaviour of *Fix* and then the transformation $lookup[x].A$.

Fixed-point Cell

The fixed point cell *Fix* creates a local channel c. Initially we send the empty form ϵ along the channel. This is done by the expression $send()$. The value of *Fix* consists of two services: A setter method available by *set* and a getter method. The body of the *set* is as follows. We first receive a form from the local channel c and bind it to a dummy variable d. Then we send the new value along the channel and return the same value. The getter service works as follows: We receive a form d from c and send d along the channel and return it.

The behaviour of *Fix* is that of a reference cell initialized with the empty form. We can set it to a new value. Invoking the getter service always returns the most recently set value. There is always one value written on the local channel c. This is due to the fact that receiving and sending occur only in pairs after the channel is initialized.

The following expression illustrates the protocol of *Fix*:

$y \mapsto Fix;$	initialize cell and bind it to y
$a \mapsto y() \cdot$	the value of a is the empty form
$b \mapsto (y; set) x \cdot$	initialze the cell with x, the value of b is x
$c \mapsto y()$	the value of c will be x

We use this cell to store the value of the fixed point. The idea is to set the cell exactly once with the result of the fixed point. Inside the fixed point we use the getter service and do not change the value of the fixed point after it has been set.

Using the Fixed Point

The agent $lookup[x].A$ denotes the agent A where all occurrences of the variable x are replaced by a call to the getter service of the cell. The only complicating factor is the appropriate treatment of sandbox expressions inside A.

Consider the agent of a **def** binding:

$$[\![\underline{\texttt{def}}\ \texttt{x = T}]\!] = \mathbf{R} \cdot x \mapsto Fix; (x; set)(x \mapsto lookup[x].A)$$

where $A = [\![\texttt{T}]\!]$. Thus $lookup[x].A$ is evaluated in a context given by $\mathbf{R}$ extended with the binding $x \mapsto Fix$. We set the *Fix* cell with the binding $x \mapsto F$ where F is the value of the agent $lookup[x].A$. Thus the value of the **def** binding is $x \mapsto F$.

What is the behaviour of agent $lookup[x].A$? It has the same behaviour as A but looks up x in the cell instead of in the root context. The predicate $lookup[x]$ is transparent for any agent except variables, the root context, abstractions and the sandbox expression. For instance, the binding $b \mapsto \epsilon$ remains the same:

$$lookup[x].(b \mapsto \epsilon) = b \mapsto \epsilon$$

For any other agent, $lookup[x].A$ is as follows.

Variables. If $x \neq y$ we have $lookup[x].y = y$. The semantics of any y appearing in an agent A is not changed when we put the agent into $lookup[x].A$.

We have $lookup[x].x = x(); x$. Thus the value of x is constructed by projecting onto x in the result of the getter service of the *Fix* cell. If the cell is initialized, we will get the form F where F the value of $lookup[x].A$. If the cell is not initialized, $x()$ will return the empty form and the projection onto x blocks or raises an exception, respectively.

For instance consider the term "`def x = x`". It holds that:

$$\begin{aligned}[\![\underline{\texttt{def}}\ \texttt{x = x}]\!] &= \mathbf{R} \cdot x \mapsto Fix; (x; set)(x \mapsto lookup[x].[\![x]\!]) \\ &= \mathbf{R} \cdot x \mapsto Fix; (x; set)(x \mapsto (x(); x))\end{aligned}$$

since the fixed point cell is initially empty we have $x() \approx \epsilon$

$$\approx \mathbf{R} \cdot x \mapsto Fix; (x; set)(x \mapsto (\epsilon; x))$$

Thus we cannot evaluate "`def x = x`" since it contains an illegal lookup $\epsilon; x$. Such behaviour was intended since such a (recursive) form would be the infinite form $x \mapsto x \mapsto x \mapsto \cdots$.

Root context. We have $lookup[x].\mathbf{R} = hide_x\mathbf{R}$. Thus if we explicitly refer to the current root context we get it without access to the *Fix* cell. This prevents us from accidently or maliciously accessing the cell of the fixed point.

We should keep in mind that we cannot store the root context in a binding and access the fixed point from there. The following example illustrates this:

```
def x =
    myroot = root        # myroot binds a form without x
    self: x
a = x.self()             # OK! yields x
b = x.myroot.x           # wrong since x.myroot has no x!
```

Services. It holds that $lookup[x].\lambda x.A = \lambda x.A$. Within the body of the abstraction, x binds the concrete argument as usual. As such, the inductive definition of $lookup[.]$ is not applied inside the body of the abstraction. For instance it holds that:

$$\begin{aligned} [\![\underline{\texttt{def}}\ \texttt{x = \textbackslash x.x}]\!] &= \mathbf{R}\cdot x{\mapsto}Fix;(x;set)(x{\mapsto}lookup[x].\lambda x.x) \\ &= \mathbf{R}\cdot x{\mapsto}Fix;(x;set)(x{\mapsto}\lambda x.x) \\ &\approx \mathbf{R}\cdot x{\mapsto}Fix;x{\mapsto}\lambda x.x \\ &\approx x{\mapsto}\lambda x.x \\ &= [\![\texttt{x = \textbackslash x.x}]\!] \end{aligned}$$

The first congruence comes from evaluating the *set* method of the fixed point: it returns the argument. Since the fixed point cell is not used anymore, we can discard it.

The reader should keep in mind that service definitions correspond to services with *dynamic* as formal argument. The case just discussed only occurs in artificial examples like

```
def dynamic =
    f: dynamic           # returns the dynamic context
```

where we define a fixed point for the identifier `dynamic`. To our experience, such code never appears, since `dynamic` denotes the dynamic context and is not an ordinary identifier.

In most of the cases, the formal argument is different from the fixed point variable. Consider the recursive service "`def x: x`". It holds that (writing y for *dynamic*):

$$\begin{aligned} [\![\underline{\texttt{def}}\ \texttt{x:}\ \ \texttt{x}]\!] &= \mathbf{R}\cdot x{\mapsto}Fix;(x;set)(x{\mapsto}lookup[x].\lambda y.x) \\ &= \mathbf{R}\cdot x{\mapsto}Fix;(x;set)(x{\mapsto}\lambda y.\mathbf{R}\cdot x{\mapsto}(x();x);x) \end{aligned}$$

We write this agent in canonical format with the closure $S = \epsilon;\lambda y.(\mathbf{R}\cdot d{\mapsto}c?;cd\cdot d;x)$:

$$\approx \nu c.(c(x{\mapsto}S) \mid x{\mapsto}S)$$

One can think of this expression as if the binding $x{\mapsto}S$ is stored in the channel c. The channel c represents the state and the service S contains the getter method of the fixed point cell. When we interact with $x{\mapsto}S$ by projection on x and invoke the service we get S back again.

Sandbox. The sandbox expression is the most complex one. The problem is that we do not know if the expression A in $A; B$ evaluates to a form where x is bound or not. The expression

$$\begin{aligned} lookup[x].(A; B) = {} & r{\mapsto}(lookup[x].A) \cdot x{\mapsto}x; \\ & r \cdot x{\mapsto}\lambda d.x() \cdot r; \\ & lookup[x].B \end{aligned}$$

can be read in three steps :

1. First, the context $r{\mapsto}lookup[x].A \cdot x{\mapsto}x$ evaluates to a form with two labels r and x. The binding for r contains the new root environment $lookup[x].A$. The binding for x contains the fixed point cell.
2. Then, the context $r \cdot x{\mapsto}\lambda d.x() \cdot r$ evaluates to r extended with x bound to a new service. This new service fakes the fixed point getter method. On invocation, the original content of the cell $x()$ extended with r is returned.
3. Finally, the expression $lookup[x].B$ is evaluated in the root context constructed in the second step.

The central thing happens in the body of the faked getter service $\lambda d.x() \cdot r$ in the second step. Assume the agent B refers to x, thus $lookup[x].B$ will contain the term $x(); x$. If r contains a binding for x, say $x{\mapsto}F$ then x in B will evaluate to F. If, however, r does not contain a binding for x, the client sees the content of the fixed point cell. Provided the fixed point cell is initialized, this content will contain a binding $x{\mapsto}F'$ for some form F'. The term $x(); x$ evaluates either to F or F'. If r does not contain x and the cell is not initialized, $x(); x$ evaluates to $\epsilon; x$ which means that the fixed point cannot be used because it is not set yet.

We present two examples, one where A overwrites x and one where x is not overwritten. First consider the term:

```
def x =
    x = ()
    self: x
```

We show that `x.self()` evaluates to the empty form. We have:

$$\begin{aligned} A &= [\![\texttt{x = (), self: x}]\!] \\ &= [\![\texttt{'(x = ()), (x = x), (self: x)}]\!] \\ &= \mathbf{R} \cdot x{\mapsto}\epsilon; x{\mapsto}x \cdot self{\mapsto}\lambda y.x \end{aligned}$$

We abbreviate the dynamic namespace by y. It is of no concern here anyhow. Now we have:

$$\begin{aligned} lookup[x].A = \big(& r{\mapsto}(lookup[x].(\mathbf{R} \cdot x{\mapsto}\epsilon)) \cdot x{\mapsto}x; \\ & r \cdot x{\mapsto}(\lambda d.x() \cdot r); \\ & lookup[x].(x{\mapsto}x \cdot self{\mapsto}\lambda y.x)\big) \end{aligned}$$

by using $hide_x A \cdot x{\mapsto}B \;\approx\; A \cdot x{\mapsto}B$ we get $lookup[x].(\mathbf{R} \cdot x{\mapsto}\epsilon \approx \mathbf{R} \cdot x{\mapsto}\epsilon)$ and thus:

$$\begin{aligned} \approx \big(& r{\mapsto}(\mathbf{R} \cdot x{\mapsto}\epsilon) \cdot x{\mapsto}x; \\ & r \cdot x{\mapsto}(\lambda d.x() \cdot r); \\ & x{\mapsto}(x(); x) \cdot self{\mapsto}\lambda y.\mathbf{R} \cdot x = (x(); x); x\big) \end{aligned}$$

$lookup[x].A$ is evaluated in a context where $x() \approx F$ for some F:

$$\begin{aligned}&\approx \big(\mathbf{R} \cdot x{\mapsto}(\lambda d.F \cdot \mathbf{R} \cdot x{\mapsto}\epsilon);\\ &\qquad x{\mapsto}(x();x) \cdot \mathit{self}{\mapsto}\lambda y.x();x\big)\\ &\approx x{\mapsto}\epsilon \cdot \mathit{self}{\mapsto}\lambda y.\epsilon\end{aligned}$$

Since "`def x = ...`" evaluates $lookup[x].A$ in contexts where $x()$ denotes a form (either the empty form or the value of the fixed point) we conclude that the above **def** keyword was not necessary at all. The label x is overwritten immediately, and `x.self()` returns the empty form.

Consider now the following program where the binding does not overwrite x:

```
def x =
    z = ()
    self: x
```

We have:

$$\begin{aligned}A = [\![\texttt{z = (), self: x}]\!] &= [\![\texttt{'(z = ()), (z = z), (self: x)}]\!]\\ &= \mathbf{R} \cdot (z{\mapsto}\epsilon); z{\mapsto}z \cdot \mathit{self}{\mapsto}\lambda y.x\end{aligned}$$

Like before we calculate:

$$\begin{aligned}lookup[x].A &= \big(r{\mapsto}(hide_x\mathbf{R} \cdot z{\mapsto}\epsilon) \cdot z{\mapsto}z;\\ &\qquad r \cdot x{\mapsto}(\lambda d.x() \cdot r);\\ &\qquad z{\mapsto}z \cdot \mathit{self}{\mapsto}\lambda y.x();x\big)\\ &\approx \big(\mathbf{R} \cdot z{\mapsto}\epsilon \cdot x{\mapsto}(\lambda d.x() \cdot hide_x\mathbf{R} \cdot z{\mapsto}\epsilon);\\ &\qquad z{\mapsto}z \cdot \mathit{self}{\mapsto}\lambda y.x();x\big)\\ &\approx z{\mapsto}\epsilon \cdot \mathit{self}{\mapsto}\lambda y.(x() \cdot hide_x\mathbf{R} \cdot z{\mapsto}\epsilon);x\\ &\approx z{\mapsto}\epsilon \cdot \mathit{self}{\mapsto}\lambda y.x();x\end{aligned}$$

The last simplification is due to the fact that $hide_x\mathbf{R}$ and $z{\mapsto}\epsilon$ do not contain a binding for x. The statement "`def x = ...`" denotes a form F where z is bound to the empty form and *self* is a service that returns F as expected.

A consequence of the sandbox semantics within fixed points is that a user cannot hide the fixed point being defined:

```
def x =
   root = ()        # root is now the empty form extended with x bound to the faked getter
   self: x
```

Such code is valid and the service `x.self` returns the fixed point x when invoked.

Finally, it is also worthwhile to consider the expression

$$\mathbf{R}; x$$

When put in a fixed point, we have:

$$\begin{aligned}
lookup[x].(\mathbf{R}; x) &= r{\mapsto}lookup[x].\mathbf{R} \cdot x{\mapsto}x; r \cdot x{\mapsto}(\lambda d.x() \cdot r); lookup[x].x \\
&= r{\mapsto}hide_x\mathbf{R} \cdot x{\mapsto}x; r \cdot x{\mapsto}(\lambda d.x() \cdot r); x(); x \\
&\approx r{\mapsto}hide_x\mathbf{R} \cdot x{\mapsto}x; (\lambda d.x() \cdot r)(); x \\
&\approx r{\mapsto}hide_x\mathbf{R} \cdot x{\mapsto}x; x() \cdot r; x \\
&\equiv r{\mapsto}hide_x\mathbf{R} \cdot x{\mapsto}x; x(); x \\
&\approx lookup[x].x
\end{aligned}$$

This example gives us confidence that the definition of $lookup[.]$ and of the sandbox operator in particular makes sense. Although explicit references to the root context hide the fixed point cell, the sandbox operator re-establishes it again.

5.6 Initial Root

When programming in Piccola, we assume that there are predefined services that give access to the Piccola primitives.

The initial root context where every Piccola program is executed contains these services. Let

$$C = \begin{array}{l}
newChannel{\mapsto}(\lambda d.\mathbf{new}; \mathbf{R} \cdot send{\mapsto}\lambda d.send(d; arg)) \\
\cdot\ run{\mapsto}(\lambda d.\mathbf{run}(d; arg; do)) \\
\cdot\ inspect{\mapsto}\lambda d.\lambda e.L(d; arg)(\\
\qquad isEmpty{\mapsto}(\lambda x.(e; arg; isEmpty)(e \cdot arg{\mapsto}\epsilon)) \\
\qquad \cdot isService{\mapsto}(\lambda x.(e; arg; isService)(e \cdot arg{\mapsto}\epsilon)) \\
\qquad \cdot isLabel{\mapsto}(\lambda l.(e; arg; isService)(e \cdot arg{\mapsto}(\\
\qquad\qquad project{\mapsto}(\lambda d.(l; project)(d; arg)) \\
\qquad\qquad \cdot hide{\mapsto}(\lambda d.(l; hide)(d; arg)) \\
\qquad\qquad \cdot bind{\mapsto}(\lambda d.(l; bind)(d; arg))))))
\end{array}$$

The behaviour of a Piccola script is given by $C; [\![\texttt{script}]\!]$.

The initial context wraps the built-in services so that they comply to the protocol of the dynamic namespace. There is no label-hide primitive since first-class labels can be encoded within Piccola using inspect as we have seen in Section 3.4.4.

Giving access to the primitives via predefinied services allows the programmer to overwrite these services.

This concludes the syntax and semantics of the Piccola language.

5.7 More on Fixed Points

In this section we present a purely functional encoding of fixed points and compare it with our encoding of **def** for recursive forms. We also give an example how to encode fixed points directly by using channels.

5.7.1 A Lazy Fixed-Point Combinator

In Section 3.3 we presented a fixed-point combinator in the Piccola calculus. Here is how this combinator is written in the Piccola language:

```
fix = \f: (\x: \a: f (x x) a) (\x: \a: f (x x) a)
```

or written more compactly as:

```
fix f:
    'self x a: f (x x) a
    self self
```

As an example we encode the factorial function as follows:

```
rfact fact n:                 # Assume the factorial to be the first arg
    if n < 2
        then: 1
        else: n * fact(n - 1)
fact = fix rfact              # Make the fixed point
println fact 5                # Use it!
```

To avoid cluttering the namespace with the non-recursive `rfact`, we can also write:

```
fact = fix (\fact n:
    if n < 2
        then: 1
        else: n * fact(n - 1))
```

5.7.2 Comparing the Lazy Combinator and Def

The fixed-point combinator has the same behaviour as our encoding of fixed points using channels for recursive services. Assume, a Piccola term `T` which is the body of a recursive service. The term uses `s` for itself and `a` for the argument. Now consider the term "`fix (\s a: T)`" in the context G. Let $A = [\![\texttt{T}]\!]$. We have:

$$G; [\![\texttt{fix (\textbackslash s a: T)}]\!] = \mathit{fix}(G; \lambda sa.A)$$

with $S = (\epsilon; \lambda xa.G; \lambda sa.A)(xx)a$ this expression is equivalent to

$$\begin{aligned} &\approx SS \\ &\approx x{\mapsto}S; \lambda a.(G; \lambda sa.A)(xx)a \end{aligned}$$

This service only interacts with the environment is when it is invoked with an argument F. In this case we have:

$$\begin{aligned} (SS)F &\approx x{\mapsto}S \cdot a{\mapsto}F; (G; \lambda sa.A)(xx)a \\ &\approx G \cdot s{\mapsto}SS \cdot a{\mapsto}F; A \end{aligned}$$

Thus any invocation of the recursive service leads to the agent A in a sandbox consisting of the original context G extended with s and the argument a.

Let us consider how the same term `T` behaves beneath a **def** binding and put into the same context G.

$$G; [\![(\underline{\texttt{def}}\ \texttt{s} = \texttt{\textbackslash a.T).s}]\!] = (G \cdot s{\mapsto}\mathit{Fix}; (s; \mathit{set})(s{\mapsto}\lambda a.\mathbf{R} \cdot s{\mapsto}(s(); s); A)); s$$

Let the service $S = G \cdot s{\mapsto}Fix_c; \lambda a.\mathbf{R} \cdot s{\mapsto}(s(); s); A$. This service looks up s and then executes A. The channel c is created by evaluating Fix. Fix_c is the form with the getter and setter method of the fixed-point cell that give access to c, i.e., $Fix_c = (set{\mapsto}(\epsilon; \lambda x.(\mathbf{R} \cdot d{\mapsto}c?; cx \mid x)) \cdot (\epsilon; \lambda x.(\mathbf{R} \cdot d{\mapsto}c?; cd \mid d)))$. With S the above term is equivalent to

$$\approx \nu c.c(s{\mapsto}S) \mid S$$

As before, this service is applied to F and it holds that:

$$\begin{aligned}(\nu c.c(s{\mapsto}S) \mid S)F &\equiv \nu c.c(s{\mapsto}S) \mid SF \\ &\approx \nu c.c(s{\mapsto}S) \mid G \cdot s{\mapsto}Fix_c \cdot a{\mapsto}F \cdot s{\mapsto}(s(); s); A \\ &\approx \nu c.c(s{\mapsto}S) \mid G \cdot a{\mapsto}F \cdot s{\mapsto}S; A\end{aligned}$$

Thus the **def** fixed point leads to the agent A that is executed inside the sandbox consisting of G extended with a and s. The additional message $c(..)$ that is floating around is important for two reasons: it stores S for further invocations and it cannot be accessed directly from inside A other than by invoking s. Thus both terms reduce to A inside the context G extended with s and a and have thus equivalent behaviour.

In the rest of this section we show the difference between the functional fixed point combinator and the **def** encoding. If we want to construct a fixed point for several bindings, we have to convert those bindings into an abstraction taking a dummy argument. The lazy fixed-point combinator invokes the dummy abstraction each time the fixed point is unfolded, i.e., accessed. If all our bindings were abstractions, the **def** encoding and the functional encoding are bisimilar.

However, there is a difference if we have side-effects within the dummy abstraction. This is the case, for instance, when we use fixed points for the self reference of objects. The following example illustrates the problem. It contains a factory `newPoint` to create simple point objects with an `x` and `y` part stored in different variables (see Appendix E).

```
newPoint I:                                  # initial argument I
    fix (\self dummy:
        myAsString:
            "(x = " + *(self().x) +          # pretty print
            ", y = " + *(self().y) + ")"
        x = newVar I.x
        y = newVar I.y
    ) ()
```

Self is encoded as a constant function. We can create a new point and print its value:

```
p = newPoint(x = 4, y = 5)
println p                                    # Prints: (x = 4, y = 5)
```

However, when we change the value of one variable, this change is not reflected as expected:

```
p.x <- 15                                    # change the x component to 15
println p                                    # Oops, prints: (x = 4, y = 5)
```

The reason is that each time we call `self()` a new fixed point containing two fresh variables `x` and `y` is created. The variables are initialized with the values passed on the initial creation.

With the **def** encoding, this problem does of course not occur since the fixed-point code is exactly executed once and the result is stored in the fixed-point cell.

5.7.3 Fixed Point Combinator using Channels

In this section, we give a simple and direct way to encode fixed points using channels. Consider the following service `self` bound in `x` that always returns itself:

```
x =
    'ch = newReadChannel()   # Non destructive read from the channel
    ''ch.send                # store the following form (in ch)
        self = ch.read       # self reads the value of the channel
        a = 7
    ch.read()                # return the stored value
```

The code works as follows: The local channel `ch` stores the form. It has a non-destructive `read`, i.e., this service receives a form `r` from the local channel, sends the form back along the same channel and returns the form `r`. The service `read` blocks when the channel is empty, i.e., there is no value sent along it yet. The channel is initialized by the `ch.send` service invocation. The form sent along the channel has two bindings: one is the service `self` which is the same as the `read` service. The other is the binding `a = 7`. The service `ch.read` is invoked and the value returned by this invocation is bound to the label `x`.

The form `x` has two labels: `a` and `self`. The form `self` is the same as returned by a call to `x.self()`, which is the same form as `x.self().self()` which is again the same. There is no access to the local channel `ch`. Thus we cannot change the state of the channel from the outside, as always the same form is written on it.

5.8 External Components

Piccola is a pure composition language. There are no predefined primitives for any computation. Instead, every computation is performed by external or host components. In this section we explain how host components are accessed in general and in particular how Java objects are used in JPiccola. JPiccola is the implementation of Piccola in Java.

Peer Forms. The general schema works as follows. Assume we have two systems, Piccola on one side and external components on the other side. We assume that we can access the external component as a form. We call this form the *peer form*. We denote the peer form for any external component c by $p(c)$. All services provided by the external component are services of the peer form. When we invoke from Piccola a service in a peer form $p(c)$, the provided service of c will be invoked and the argument (which is a form) must be converted to an external component. Therefore we also need a function e to convert every form F into an external component $e(F)$. If the form F is a peer form $p(c)$ then converting this form back to a component must be the original external component, written $e(p(c)) = c$. If the form is not peer form the behaviour is language specific (see below).

In order to integrate external components seamlessly we need to adapt them inside Piccola to a given style. Thus we want to extend their interface with certain bindings. The difficulty is that we still want to be able to *identify* the external component with the adapted Piccola form [Höl93]. Assume we have an external component that gets wrapped by Piccola. When we send this component back to another external service, the external service must not see the adapted, but the original, i.e., the unwrapped component.

Extensibility of forms solves this problem. Let us define two functions *up* and *down* [DM98]. Those two functions describe the effect of *up*-ing an external component into a form, and *down*-ing a form into an external component. The idea is that we keep the peer form in a distinct label `peer` in order to prevent unexpected overwriting. Let

$$Up(c) \overset{\text{def}}{=} p(c) \cdot peer{\mapsto}p(c) \tag{5.1}$$

$$Down(F) \overset{\text{def}}{=} e((peer{\mapsto}F \cdot F); peer) \tag{5.2}$$

By this schema it holds:

$$Down(Up(c) \cdot F) = o$$

provided F denotes a form without the label `peer`. Thus we can modify any up-ed component. As long as we keep the `peer` binding, we can invoke an external service with this form and the external service receives the original component.

The general scheme is the specification for any implementation of Piccola. It is also used in SPiccola [Sch01], the Piccola implementation in Squeak [IKM+97].

Java–Piccola Bridge. We now explain how in particular Java objects can be accessed from within JPiccola. We assume that components written in yet another language, for instance C++ objects or Corba Components, are accessed either directly from Java or via Java's native interface. If these components are accessible from Java, they are also accessible from JPiccola. As explained above, a peer form gives access to the provided services of an external component. If this component is a Java object, the form has services providing access to all public methods of it. Host objects are instantiated via peer classes. *Peer classes* give access to all static methods and fields of a class and they contain a factory service to create peer objects. Peer classes are created in JPiccola with the predefined service `Host.class` with the name of the class as argument. For instance

```
    Host.class "java.lang.System"
```

creates the peer class for the Java System class.

Method dispatch works as follows. Assume we invoke a service `ser` on a peer object o with the argument form F. The following steps are used to find the appropriate message to send to the object in Java. We first construct a form-tuple from the argument, then we create an tuple of Java objects, and finally we associate a type-tuple in order to resolve overloaded methods.

1. If the form F has bindings `val1`, `val2` up to `val`n, then we create the tuple

   $$(F.val1, F.val2, ..., F.valn)$$

   If the form has no binding `val1` we create the 1-tuple (F). If the form is the empty form, we create the empty tuple $()$.

2. Next we convert the form tuple into a tuple of objects $(o_1, ..., o_n)$ by applying *Down* on each component of the tuple.

3. Finally, we construct a tuple of types in order to resolve overloaded methods. For each Java object o_i of the tuple, the type t_i is the class of the corresponding object. If however the form F has a binding `type`i, then the type t_i is *F.typei*.

The Java method that corresponds to the service has the signature `ser`$(t_1, t_2, ...t_n)$. If such a method does not exist, an exception is thrown.

Primitive data-types of Java are handled by their associated classes, e.g., the type `int` is an object of instance `java.lang.Integer`. The JPiccola system offers static methods to do the arithmetic for such values since the Java language framework does not provide methods that do arithmetic operations.

Dispatch for the constructor in peer classes happens by the same schema like method dispatch.

Finally, externalizing a form that is not a peer form, returns the implementation object that is used inside JPiccola to model forms. This allows us to reify the Piccola runtime environment. We have not explored what can be achieved by accessing and modifying the runtime environment from Piccola as this would make Piccola a reflective system.

Protecting Forms. Host components and their services can be separated into two categories: clients and containers. *Client services* use the arguments passed as required services. For client services it is important that the adaption from Piccola does not change the identity. This is ensured by using the `peer` binding.

In contrast, *container services* store their arguments without invoking any service on them. Typically, containers have other services to retrieve stored elements. As an example consider the following `list` which is a host object of type `ArrayList`. We use the Java methods `void add(int index, Object element)` to store an element and `Object get(int index)` to get the element back. We store the adapted peer object `obj`:

```
obj = Host.class("java.lang.Object").new()      # create a new object
obj.tag = "anAdaptedObject"                     # adapt from Piccola
list = Host.class("java.util.ArrayList").new() # cerate a collection
''list.add(val1 = 1, val2 = obj)                # add adapted object
obj = list.get 1                                # get element back
''println obj.tag                               # fails! tag not defined
```

When we add the form `obj` we down it and thus add the Java object (which does not contain `tag`). When we (later) retrieve this element the adaption is lost.

We protect forms from loosing their adaption by putting them into a `peer` binding. We define the service `protect` as follows and change the above code to protect the added element:

```
protect Form: peer = Form
''list.add(val1 = 1, val2 = protect obj)        # protect the form
```

When we pass the form `obj` down to Java, the value of the `peer` binding, which is not an upped object, is added. The adaption is not lost. In Section 6.1 we will see an example of how an `ArrayList` of Java is adapted by a wrapper.

5.9 Language Design

Piccola is a composition language based on the idea of forms. Adhering to our principle of generalization we basically only define form-expressions in the syntax. In this section we explain the most important language decisions we took.

Quotes. In earlier versions of Piccola we had a keyword **return** to denote the form to be returned. The example below defines a channel that contains an additional non-destructive `read` service. To the left, it is written in Piccola and to the right in the old syntax using return.

```
newReadChannel:
    'ch = newChannel()
    ch
    read:
        'r = ch.receive()
        ''ch.send r
        r
```

```
newReadChannel:
    ch = newChannel()
    return
        ch
        read:
            r = ch.receive()
            ch.send r
            return r
```

We found two drawbacks with the **return** keyword. First, too many **return** keywords confuse the reader, especially, when we use them in nested forms, which is syntactically correct. Second, **return** was not doing what users expected when it was used, for example inside an if-then-else expression. In fact, **return** does not have the semantics of blocks that return to the home-context of the block, as in Smalltalk. In Section 7.10.1 we will give such an encoding of blocks.

It turned out that **return** was used only to keep binding local. However, local bindings can also be modeled by simply extending the current root with these bindings. This is exactly what can be achieved with quotes. The language with the return-keyword is described and used in earlier papers about Piccola [AN00, AKN00, ALSN01, AN01].

Indentation. We have adopted the use of indentation for structuring code from Haskell and Python. It turns out that indentation is helpful when the expressions are not too deeply nested. In fact, we often refactored code when it was too much indented and found useful high-level abstractions through the refactoring. Deep indentation indicates that too many things are going on at the same place. In Chapter 8 we see this deep indentation when we compose glue code wrappers with their respective components (see Figure 8.14).

When a form expression has many bindings they all have to be on the same indentation level. Writing and reading such code sometimes is awkward. We expect that a composition environment could alleviate this problem to a certain degree. The goal would be that each form expression should fit on a single page. Longer forms are broken into pieces and managed by the composition environment.

User Defined Operators. Operator overloading is common in many object-oriented and functional languages. The idea is that the service associated with, for instance, the plus in $a + b$ is not globally determined but specified by the types of a and b. Python uses the following schema to dispatch the expression $A + B$: If object A understands the message `__add__`, this message is sent with *B* as argument. Otherwise, if object *B* understands the message `__radd__` (the right-hand side operator) we send this message with *A* as argument. Otherwise a exception is thrown [Lut96].

We use the left-hand expression in infix terms do the dispatch. However, Piccola additionally supports global operators. This allows us, for instance, to define form equality as a global operator. Forms can define their own equality by defining `_==_`. The global form equality checks if all bindings are mutually equal.

Piccola can also model Python's dispatch. The following definition delegates plus calls to the right-hand side.

```
DefaultOp._+_default L R: R.radd L
```

Thus, when the left-hand side in a plus-expression does not contain a binding for `_+_`, the call is handled by the right-hand side. For instance, the expression "`() + A`" becomes "`A.radd()`" since the empty form has no label `_+_`.

With the translation for user-defined collections, Piccola allows the collections associated with brackets to be changed. In contrast, languages like Prolog or Haskell provide syntactic sugar for lists, but the concrete implementation for lists cannot be changed.

Special Form Abstractions. Lisp supports so-called special form abstractions [Pit80]. The idea of these abstractions is that the order of evaluation can be defined as part of the abstraction. The abstraction receives the unevaluated expressions as argument. As an example this allows the programmer to define an if-then-else abstraction as a ternary function. The first argument is a boolean value and the other two are the unevaluated then- and else-branch. If the boolean is true we evaluate the then-branch otherwise the else-branch. With standard eager evaluation, such an abstraction cannot be defined since both branches would be evaluated before invoking `if`.

In Piccola all arguments are passed by value. A similar effect to special form abstractions is achieved by using service definitions instead of nested forms. As an example, booleans have a service `select`:

```
true = (select X: X.true)
false = (select X: X.false)
```

We define the global service `if`:

```
if Bool Cases:
    'Cases = (then: (), else: (), Cases)        # provide defaults
    Bool.select(true = Cases.then, false = Cases.else)()
```

Invoking `if` we pass a boolean and a `Cases` form. The boolean selects either the `then` or the `else` service and invokes it. The result of invoking `if` is the value returned by the case branch. For instance:

```
if true
    then: println "then branch"
    else: println "else branch"
```

prints out "`then branch`". The abstraction `if` creates the illusion of a keyword due to indentation and the fact that we create an ad-hoc form containing the then and the else branch.

5.10 Discussion

We have added a layer of expressive power on top of the Piccola calculus. This expressive power is needed to efficiently use Piccola as a composition language. Many of the features are syntactical sugar for Piccola. For instance we have introduced an abbreviatation for user-defined infix operators that support polymorphism in their left component. Contrarily, a user may also want to define *global* operators, like generic form comparison. The possibility to define the semantics of Piccola operators as syntactic sugar gives us confidence that the expressive power of the calculus is appropriate. Of course, an efficient implementation of Piccola may choose to treat infix operators as primitive instead of expanding them according to the abbreviation rule *abb-infix*.

The semantics of the language contains dynamic namespaces and fixed points in addition to the Piccola calculus. The definition of fixed points is based on channels. We demonstrate how to encode mutual recursive services using a purely functional sub-language, i.e., without using channels and explicit agents and we have shown that both encodings are equivalent for service definitions.

An alternative approach that avoids the complexity of the $lookup[.]$ operator is to define recursive forms as primitive in the calculus and give an observational characterization. We have not chosen this approach since it may violate our assumption that forms are finite trees.

We argue that finitary forms have a closer relation with interfaces and environments used in component based programming. By assuming finite forms we can specify generic glue wrappers that iterate over forms. With recursive forms, applying generic glue wrappers may lead to nontermination. Further work is needed to investigate whether one can define a type system that guarantees the necessary finiteness of forms while still enabling recursive values.

Another approach would be to give up the expressiveness of the inspect operator and replace it with a general wrapper. The semantics of this wrapper can be thought of as a map service over forms. In fact, from our programming experience it seems enough to have first class labels, a comparison operator for those labels, and a map function. Our proposal with the inspection operator is more powerful as it allows us to encode the needed composition abstraction. However, it may turn out that a smaller set of composition abstractions would be sufficient for defining composition abstractions. In that case, one could drop inspect, replace it with weaker operators that would not be based on finitary assumptions anymore.

Chapter 6

Partial Evaluation

In this chapter we present a partial evaluation algorithm for Piccola. This algorithm uses the fact that forms are immutable. We replace references to forms by the forms referred to. We can then specialize projections and replace applications of referentially transparent services by their results. However, most services in Piccola are not referentially transparent and cannot be inlined since that would change the order in which side-effects are executed. We need to separate the referentially transparent part from the non-transparent part in order to replace an application with its result and to ensure that the order in which the side-effects are evaluated is preserved.

The algorithm separates functional Piccola agents into *side-effect terms* and *lazy forms expressions*. Side-effect terms contain applications that may cause side-effects and projections that may be undefined. For side-effect terms the order of evaluation is important. In contrast, lazy forms are referentially transparent. As subexpressions they contain deferred projections, bindings, and hidden forms. Dropping unnecessary subexpressions does not change the semantics of the lazy form. We call these expressions lazy since the bindings can be evaluated on demand.

This chapter is structured as follows. In Section 6.1 we present a wrapper that adapts host components. We consider when this wrapper is used and motivate the need for the partial evaluator. In Section 6.2 we give an overview of the algorithm, which is formally defined in Section 6.3. In Section 6.4 we prove correctness and termination of the algorithm. In Section 6.5 we extend the algorithm. We consider services that have a side-effect and services that just introduce new state. Applying *new-state* services can be deferred but such applications are not referentially transparent. We improve the results of specialization by treating new-state services differently than side-effects. In Section 6.6 we show how to extend the algorithm with constant folding. In Section 6.7 we describe possible enhancements to the algorithm and issues when implementing it.

6.1 A typical Example

In Figure 6.1 (part of) the adapter for Java lists is given. The service `wrapList` adapts a form `list` that gives access to a Java object of `java.util.ArrayList` so that it can be used with Piccola's collection notation. In particular, a collection must support the `add` service that adds an element and returns the collection in order to be used with the bracket notation (see Section 5.3).

```
def wrapList list:
    peer = list.peer                     # keep peer binding
    add X:                               # add X and return wrapped list
        ''list.add(protect X)            # protect X when downing the form
        wrapList list
    isEmpty: wrapBoolean(list.isEmpty())
    size: wrapNumber(list.size())        # return the size
    clone: wrapList(list.clone())
    _+_ Elem: clone().add Elem           # clone and add to the close
    forEach X:                           # iterate over all elements
        'is = wrapIterator list.iterator()
        'def loop:
            if is.hasNext()
                then:
                    X.do is.next()       # and invoke X.do
                    loop()
        ''loop()
    reduce F Init:                       # reduce by using F
        'res = newVar Init
        ''forEach
            do Elem: res <- (F (*res) Elem)
        *res

DefaultOp.[_]: wrapList                  # make a new list
    Host.class("java.util.ArrayList").new()
```

Figure 6.1: Wrapper for a List

Some services defined by the adapter have a corresponding Java method. For instance `size` and `isEmpty` call the corresponding method and wrap the results as Piccola numbers or boolean, respectively. The service `add` uses the underlying Java method to add an element and returns the wrapped list. Other services are implemented by helper routines, for instance the service `forEach` creates an iterator for the object and iterates over the elements. Observe that there are dependencies between the defined services. For instance the service `+` uses `clone` and the service `reduce` depends on `forEach`.

In order to motivate the partial evaluation, let us consider what happens when we evaluate the expression "`[1,2].forEach(do: ...)`". Recall that this expression is syntactic sugar for "`DefaultOp.[_]().add(1).add(2).forEach(do: ...)`". The following actions are triggered when we evaluate the expression:

1. The host object for a new empty list is wrapped by `wrapList`.
2. The adapted service `add` is invoked. This leads to a call on the host list.
3. The list is wrapped again as a result of the `add` service.
4. The same is done for adding the element 2.
5. The adapted service `forEach` is invoked. This leads to the creation of an iterator on the host list.

Now the iterator visits all stored elements in the list. The `wrapList` wrapper is invoked several times, but only one service it defines is used each time. The following code has the same behaviour but uses no wrapper:

```
'list = Host.class("java.util.ArrayList").new()  # new list
''list.add (peer = 1)                             # protect inlined
''list.add (peer = 2)
'iterator = list.iterator()
...                                               # use the iterator
```

Here we directly call the associated methods of the external component. Of course, the second code example runs more efficiently.

Such situations occur frequently in Piccola. Adapting components normally does not cause a side-effect and we would like to inline such adapters. In the following we present a partial evaluation algorithm that inlines referentially transparent services and separates the side-effect.

6.2 Overview

Partial evaluation [AMY97, JGS93, CD93] is a program transformation technique which, given a program and parts of its arguments, produces a specialized program with respect to those arguments.

Partially evaluating programs that may contain side-effects has to take into account the evaluation strategy. Because partial evaluators perform non-standard eager evaluation, we need to know which services have side-effects. In pure, functional languages the property of referential transparency assures that any data structure created at partial evaluation time has the same value throughout the specialization. With side-effects this assumption does not hold anymore.

Before presenting the algorithm in detail, we give an informal account of the main idea. We separate each service s into two services s_p and s_r. The first service, s_p, is the side-effect part of the service. When we apply s_p on a form F, the side-effects of sF are evaluated. We refer to the result of s_pF as the side-effect. The service s_r is referentially transparent. It takes the side-effect and the argument F and returns the value of sF. Thus the service s is split into s_p and s_r such that the following holds:

$$sF = s_r(s_pF)F$$

As an example, consider the service `wrapRec`:

```
wrapRec ch:
    received = ch.receive()         # side-effect
    value = wrap received           # wrap is referentially transparent
    channel = ch                    # return ch as part of the result
```

The service receives a value from the channel `ch` and wraps the received form using a service for which we assume that it is referentially transparent. The services `wrapRec`$_p$ and `wrapRec`$_s$ are now:

```
wrapRec_p ch:  ch.receive()
wrapRec_r side ch:
    received = side
    value = wrapped side
    channel = ch
```

The trick is that we can now defer invocation of the referentially transparent service. Assume we use the result of an invocation of `wrapRec` and project on the `received` label. In that case, the invocation of `wrapped` is not necessary anymore. The code

```
a = (wrapRec ch).received
```

is the same as "`a = wrapRec`$_\mathtt{r}$ (`wrapRec`$_\mathtt{p}$ `ch`) `ch`" which is the same as

```
a = ch.receive()
```

when we inline the referentially transparent service.

For the algorithm we not only split services but any functional agent (see Section 5.4). Furthermore, we do not represent the lazy part as curried services. Instead the root context consists of the side-effects bound by unique labels.

For simplicity, we first do not consider the hide primitive and treat it as a service with a side-effect. In Section 6.5 we show how to handle hide more effectively.

The partial evaluation algorithm *partial* : $\mathcal{A} \to \mathcal{A}$ is expressed in two steps. First an agent A is split into a side-effect term and a lazy form expression. Then, the side-effect term and the lazy form are combined back into a specialized agent. The set of side-effect terms is denoted by $\mathcal{P}$ and ranged over by P. The set of lazy forms $\mathcal{R}$ ranged over by R. Some helper predicates are defined over lazy forms and side-effects. In that case we use Q to range over of $\mathcal{P} \cup \mathcal{R}$. The grammar for lazy forms and side-effect terms is given in Table 6.1. We adopt the same precedence as for functional agents: Projection is stronger than binding which is stronger than application.

P	$::=$	ϵ	*empty form*	$\mid$	$x{\mapsto}P$	*nested side-effect*
	$\mid$	$P \cdot P$	*extension*	$\mid$	$x{\mapsto}R.x$	*projection*
	$\mid$	$x{\mapsto}RR$	*side-effect application*			
R	$::=$	ϵ	*empty form*	$\mid$	x	*variable*
	$\mid$	$R \cdot R$	*extension*	$\mid$	$x{\mapsto}R$	*binding*
	$\mid$	$R.x$	*projection*	$\mid$	$\lambda x.P \star R$	*lazy abstraction*
	$\mid$	$\mathbf{side}(A)$	*side-effect service*			

Table 6.1: Lazy Forms

The two functions of the algorithm are *split* and *combine*:

- The function $split : (\mathcal{A} \times \mathcal{R}) \to (\mathcal{P} \times \mathcal{R})$ separates a functional agent into a side-effect and a lazy form. We split the agent A in the context given by the lazy form R' and get a side-effect term P and a lazy form R, written $split(A, R') = (P, R)$.
- The function $combine(\mathcal{P} \times \mathcal{R}) \to \mathcal{A}$ combines the side-effect and the lazy form back into a functional agent.

The partial evaluation algorithm $partial : \mathcal{A} \to \mathcal{A}$ is defined as:

$$partial(A) = combine(split(A, \epsilon)) \tag{6.1}$$

Observe that we assume the empty form as initial context for specializing an agent A.

Lazy form expressions are referentially transparent. As we will see, they contain unevaluated projections that are guaranteed to succeed. Evaluating lazy forms can be deferred. Lazy forms contain references to side-effects or to formal parameters. Lazy abstractions $\lambda x.P \star R$ contain their side-effect P and referentially transparent result R.

Side-effect services are arbitrary agents A. Partial evaluation does not specialize them. The primitive services **new**, **run**, and **L** are side-effect services. Side-effect terms contain applications and projections that may fail. Side-effect terms have a specific structure. Atomic side-effects are bound by unique labels, side-effect terms can be nested and sequentially be composed. In $P_1 \cdot P_2$ we may refer to side-effects of P_1 from P_2.

6.3 The Algorithm

We present and discuss the functions *split* and *combine* in detail. We assume that $\cdot$ is associative and ϵ is the neutral element. This allows us to reduce the number of defining equations. For instance, when defining projection in Table 6.6 we write $project(R \cdot x{\mapsto}R_1, x) = R_1$ assuming that we can rewrite any form with several bindings into a form extended with a single binding.

We need some helper predicates for the free and bound variables and we define substitution.

The set of free variables in a lazy or side-effect term, $fv(Q)$ is defined in Table 6.2. Note that the definition of free labels of an ordinary Piccola agent is meaningless as the free labels of $\mathbf{R}$ are undefined. For lazy forms and side-effects, a recursive definition can be given since

$$\begin{aligned} fv(\epsilon) &= \emptyset & fv(x) &= \{x\} \\ fv(R_1 R_2) &= fv(R_1) \cup fv(R_2) & fv(R.x) &= fv(R) \\ fv(Q_1 \cdot Q_2) &= fv(Q_1) \cup fv(Q_2) & fv(x \mapsto Q) &= fv(Q) \\ fv(\lambda x.P \star R) &= (fv(P) \cup fv(R)) \backslash (labels(P) \cup \{x\}) & fv(\mathbf{side}(A)) &= \emptyset \end{aligned}$$

$$\begin{aligned} labels(\epsilon) &= \emptyset & labels(x) &= \emptyset \\ labels(x \mapsto Q) &= \{x\} & labels(R.x) &= \emptyset \\ labels(Q_1 \cdot Q_2) &= labels(Q_1) \cup labels(Q_2) & labels(\mathbf{side}(A)) &= \emptyset \\ labels(R_1 R_2) &= \emptyset & labels(\lambda x.P \star R) &= \emptyset \end{aligned}$$

Table 6.2: Free variables and defined labels

sandbox expression are inlined and lazy forms do not contain **R**. The interesting case is the free variables for abstractions $\lambda x.P \star Q$. They are constructed by taking the free variables of P and R and by removing x and the labels that are defined by P. This definition reflects the fact that R will be evaluated in a context defined by P as we will see.

The predicate $labels(Q)$ denotes the set of labels that are bound by Q. The label of a binding $x \mapsto Q$ is the set $\{x\}$. The set of labels of an extension is the union of the subexpressions. We are conservative with the set of labels when the labels cannot be inferred straightforward. For instance, the set of labels of any application or projection is empty.

The expression $Q[x/R]$ denotes the expression Q where all free x are replaced by R. Substitution is defined in Table 6.3. Note that there is no special definition for the side-effect $y \mapsto R_1 R_2$. This case is defined by the binding and by the application, thus $(y \mapsto R_1 R_2)[x/R] = y \mapsto (R_1 R_2)[x/R] = R_1[x/R]\ R_2[x/R]$. Note that $R[x/R]' \in \mathcal{R}$ and $P[x/R] \in \mathcal{P}$. This means that a substitution on a lazy form denotes a lazy form, and a substitution on a side-effect term denotes a side-effect term. As usual we replace bound variables to avoid name capture [HS86].

Notice that substitution on a projection is defined in terms of the helper predicate *project* which is defined later.

6.3.1 Combining Side-effects and Lazy Forms

The function *combine* gives a denotational semantics to pairs of side-effects and lazy forms. It does so by translating them to Piccola agents. The function *combine* is defined as:

$$combine(P, R) = combine'(P); embed(R) \tag{6.2}$$

A side-effect and a lazy form are combined to a sandbox expression where the root context is the combined side-effect and the value is the embedded lazy form. The functions *embed* and *combine*$'$ are given in Table 6.4. The embedding is compositional except for abstractions that respect the special nature of lazy closures. Since an abstraction $\lambda x.P \star R$ itself contains a side-effect part and a lazy form value, the embedding is $\lambda x.combine(P, R)$.

The function $combine'(P)$ translates a side-effect into a functional agent. It replaces the sequential composition operator of the side-effect with a sandbox. Nested side-effects, ap-

$$
\begin{aligned}
\epsilon[x/R] &= \epsilon \\
(P \cdot Q)[x/R] &= P[x/R] \cdot Q[x/R] \\
x[x/R] &= R \\
y[x/R] &= y && \text{where } x \neq y \\
(y{\mapsto}Q)[x/R] &= y{\mapsto}Q[x/R] \\
(R_1R_2)[x/R] &= R_1[x/R]\ R_2[x/R] \\
\mathbf{side}(A)[x/R] &= \mathbf{side}(A) \\
(\lambda x.P \star R_1)[x/R] &= \lambda x.P \star R_1 \\
(\lambda y.P \star R_1)[x/R] &= \lambda y.P[x/R] \star R_1[x/R] && \text{where } x \neq y\text{, and} \\
&&& \quad y \notin \mathit{fv}(R) \text{ or } x \notin \mathit{fv}(P, R_1) \\
(\lambda y.P \star R_1)[x/R] &= \lambda z.P[x/z][x/R] \star R_1[x/z][x/R] && \text{where } x \neq y \text{ and} \\
&&& \quad y \in \mathit{fv}(R) \text{ and } x \in \mathit{fv}(P, R_1) \\
(R_1.y)[x/R] &= \mathit{project}(R_1[x/R], y)
\end{aligned}
$$

Table 6.3: Substitution

$$
\begin{aligned}
\mathit{embed}(\epsilon) &= \epsilon & \mathit{embed}(x) &= x \\
\mathit{embed}(R_1 \cdot R_2) &= \mathit{embed}(R_1) \cdot \mathit{embed}(R_2) & \mathit{embed}(\mathbf{side}(A)) &= A \\
\mathit{embed}(x{\mapsto}R) &= x{\mapsto}\mathit{embed}(R) & \mathit{embed}(P.x) &= \mathit{embed}(P); x \\
\mathit{embed}(\lambda x.P \star R) &= \lambda x.\mathit{combine}(P, R) \\
\\
\mathit{combine}'(P_1 \cdot P_2) &= \mathit{combine}'(P_1); \mathit{combine}'(P_2) & \mathit{combine}'(\epsilon) &= \mathbf{R} \\
\mathit{combine}'(x{\mapsto}R_1R_2) &= \mathbf{R} \cdot x{\mapsto}\mathit{embed}(R_1)\mathit{embed}(R_2) & \mathit{combine}'(x{\mapsto}R.x) &= \mathbf{R} \cdot x{\mapsto}\mathit{embed}(R.x) \\
\mathit{combine}'(x{\mapsto}P) &= \mathbf{R} \cdot x{\mapsto}\mathit{combine}'(P)
\end{aligned}
$$

Table 6.4: Embedding side-effects and lazy terms

plications and projections are combined to extensions of **R** with the embedded expression. Recall from the introduction that the root context **R** will contain the side-effects.

The following agent A illustrates how *combine* works. The agent defines two services f and g that both contain a side-effect by calling unknown services c and d respectively. Furthermore, g calls f.

$$\begin{aligned} A = {} & \mathbf{R} \cdot f{\mapsto}(\lambda x.\mathit{resultf}{\mapsto}c(\mathit{argf}{\mapsto}x)); \\ & \mathbf{R} \cdot g{\mapsto}(\lambda y.\mathit{resultg}{\mapsto}d(f y)); \\ & a{\mapsto}g() \end{aligned}$$

We expect that the side-effect of g must be the composition of the side-effects of f and d and that the argument passed to d is the resulting value of $f y$. In the following subsection we will see that splitting A in the context $c{\mapsto}c \cdot d{\mapsto}d$ yields a side-effect P and a lazy form R as follows:

$$\begin{aligned} P &= y_4{\mapsto}(y_2{\mapsto}y_1{\mapsto}c(\mathit{argf}{\mapsto}\epsilon) \cdot y_3{\mapsto}d(\mathit{resultf}{\mapsto}y_2.y_1)) \\ R &= a{\mapsto}\mathit{resultg}{\mapsto}y_4.y_3 \end{aligned}$$

Observe that the side-effect y_4 is a nested side-effect that consists of y_2 and y_3. The right-hand side of y_3 uses the side-effect $y_2.y_1$ to refer to the side-effect of $c(\mathit{argf}{\mapsto}\epsilon)$. When combining this tuple into a functional agent, we have to make sure that side-effect identifiers are correctly bound. This is achieved by transforming any side-effect binding into an extension of the root context with this binding:

$$\begin{aligned} \mathit{combine}(P, R) = {} & \mathbf{R} \cdot y_4{\mapsto}(\\ & \qquad \mathbf{R} \cdot y_2{\mapsto}(\mathbf{R} \cdot y_1{\mapsto}c(\mathit{argf}{\mapsto}\epsilon)); \\ & \qquad \mathbf{R} \cdot y_3{\mapsto}d(\mathit{resultf}{\mapsto}(y_2; y_1))); \\ & a{\mapsto}\mathit{resultg}{\mapsto}(y_4; y_3) \end{aligned}$$

We can think of the combination of side-effect terms as if each side-effect binding was a quoted binding:

```
'y4 =
    'y2 =
        'y1 = c(argf = ())
        root
    'y3 = d(resultf = y2.y1)
    root
a = resultg = y4.y3
```

6.3.2 Separating Side-effects

We discuss splitting of agents which is the heart of the specialization algorithm. The function $\mathit{split}(A, R)$ is defined in Table 6.5. The first set of definitions (6.3 – 6.5) split primitive services into their corresponding lazy forms. The side-effect part is empty since the construction of a service has no side-effect. The lazy forms are the primitive services marked as side-effect services.

$$split(\mathbf{L}, R') = (\epsilon, \mathbf{side}(\mathbf{L})) \tag{6.3}$$
$$split(\mathbf{new}, R') = (\epsilon, \mathbf{side}(\mathbf{new})) \tag{6.4}$$
$$split(\mathbf{run}, R') = (\epsilon, \mathbf{side}(\mathbf{run})) \tag{6.5}$$
$$split(hide_x, R') = (\epsilon, \mathbf{side}(hide_x)) \tag{6.6}$$
$$split(\epsilon, R') = (\epsilon, \epsilon) \tag{6.7}$$
$$split(\mathbf{R}, R') = (\epsilon, R') \tag{6.8}$$
$$split(x{\mapsto}A, R') = (P, x{\mapsto}R) \qquad \text{where } split(A, R') = (P, R) \tag{6.9}$$
$$split(A; B, R') = (P_1 \cdot P_2, R_2) \qquad \text{where } split(A, R') = (P_1, R_1) \text{ and } split(B, R_1) = (P_2, R_2) \tag{6.10}$$
$$split(A \cdot B, R') = (P_1 \cdot P_2, R_1 \cdot R_2) \qquad \text{where } split(A, R') = (P_1, R_1) \text{ and } split(B, R') = (P_2, R_2) \tag{6.11}$$
$$split(\lambda x.A, R') = (\epsilon, \lambda x.P \star R) \qquad \text{where } split(A, R' \cdot x{\mapsto}x) = (P, R) \tag{6.12}$$

and

$$split(x, R') = \begin{cases} (\epsilon, project(R', x)) & \text{if } x \in labels(R') \\ (y{\mapsto}project(R', x), y) & \text{otherwise} \end{cases} \tag{6.13}$$

$$split(AB, R') = \begin{cases} (P_1 \cdot P_2, R_3[x/R_2]) & \text{if } service(R_1) = \lambda x.\epsilon \star R_3 \\ (P_1 \cdot P_2 \cdot y{\mapsto}P_3[x/R_2], \\ \quad nest(R_3, y, P_3)[x/R_2]) & \text{if } service(R_1) = \lambda x.P_3 \star R_3 \text{ and } P_3 \neq \epsilon \\ (P_1 \cdot P_2 \cdot y{\mapsto}service(R_1)R_2, y) & \text{otherwise} \end{cases} \tag{6.14}$$

where $(P_1, R_1) = split(A, R')$, $(P_2, R_2) = split(B, R')$ and y denotes a unique identifier.

Table 6.5: Split Function

The side-effect of evaluating the empty form (6.7) is the empty form and the result is the empty form. The result of evaluating **R** (6.8) is the current context R' and the side-effect is empty.

Splitting a binding (6.9) works as follows: We first split A which yields a side-effect P and a result R. The side-effects are propagated and the resulting lazy form is the lazy binding $x{\mapsto}R$.

Sandbox expressions $A; B$ are split into side-effect terms and lazy forms as follows (6.10): First the agent A is split in the context R'. This yields a side-effect P_1 and an intermediate context R_1. Then, expression B is split in the intermediate context R_1 into a side-effect P_2 and a lazy form R_2. The side-effects are composed to $P_1 \cdot P_2$. The resulting lazy form is R_2. Observe that R_1 does not occur in the final result. The equation 6.11 for splitting $A \cdot B$ is similar. The difference is that we evaluate both subterms in the same root context R' and that the resulting lazy form is the extension of the respective lazy forms.

Specializing an abstraction $\lambda x.A$ (6.12) yields no side-effects and splits the body of the abstraction in the context R' extended with a binding $x{\mapsto}x$. This extension reflects the fact that the root context for A will contain a binding x. Observe that we do not infer anything about what binding x contains.

Evaluation of x (6.13) is done by projecting x in the current root context R'. There are two distinct cases. If it is known that R' has a binding for x, i.e., $x \in \mathit{labels}(R')$, projection is guaranteed to succeed and has no side-effects. In the other case — if we cannot ensure that R will contain a binding for x — the projection is part of the side-effect. The projected value $\mathit{project}(R, x)$ is bound by a unique identifier y. The lazy form is the variable y. This case reflects the fact that looking up a variable in a context where it is not defined cannot be reduced further or raises an exception which is a side-effect.

The helper predicate $\mathit{project} : (\mathcal{R} \times \mathcal{L}) \to \mathcal{Q}$ denotes the value bound by a label. If the projection can be performed at specialize time, we do the actual lookup. If the value of the projection is not known, an unevaluated projection is returned. For instance $\mathit{project}(R_1 \cdot x{\mapsto}R_2, x) = R_2$ and $\mathit{project}(y, x) = y.x$. The term $\mathit{project}(R, x)$ is recursively defined on R in Table 6.6. If R is the empty form then projection on it will never succeed. In that case we might warn that a type error will occur at runtime. Formally, we define *error* to be the projection $\epsilon.x$.

If the form is extended to its right with a binding $x{\mapsto}R$, projection on x returns R. This is the important case that simplifies a projection expression. If the form is an extension with a service or an extension with a binding with a different label, projection proceeds recursively. In any other case, projection cannot be determined at specialize time and $\mathit{project}(R, x)$ denotes the projection $R.x$. Note that $R.x$ is a lazy form if $x \in \mathit{labels}(R)$, for instance $(x{\mapsto}\epsilon \cdot y).x \in \mathcal{R}$.

Before we discuss the case of an application, we present a few examples to get an idea how *split* separates functional agents. For instance:

$$\mathit{split}(\lambda x.x, \epsilon) = (\epsilon, \lambda x.\epsilon \star x) \tag{6.15}$$

$$\mathit{split}(\lambda x.(y{\mapsto}x; y), \epsilon) = (\epsilon, \lambda x.\epsilon \star x) \tag{6.16}$$

$$\mathit{split}(\lambda x.(y{\mapsto}\epsilon \cdot x; y), \epsilon) = (\epsilon, \lambda x.\epsilon \star (y{\mapsto}\epsilon \cdot x).y) \tag{6.17}$$

$$\mathit{split}(\lambda x.(x; y), \epsilon) = (\epsilon, \lambda x.y_1{\mapsto}x.y \star y_1) \tag{6.18}$$

The first two abstractions are identity services. Splitting those services removes the sandbox expression in the second example. Both abstractions have an empty side-effect. For the

$$\begin{aligned}
project(\epsilon, x) &= \text{error} \equiv \epsilon.x \\
project(R \cdot x{\mapsto}R_1, x) &= R_1 \\
project(R \cdot y{\mapsto}R_1, x) &= project(R, x) && \text{if } x \neq y \\
project(R \cdot (\lambda y.P \star R_1), x) &= project(R, x) \\
project(R \cdot \mathbf{side}(A), x) &= project(R, x) \\
project(R, x) &= R.x && \text{otherwise}
\end{aligned}$$

Table 6.6: Projection

$$\begin{aligned}
service(\epsilon) &= \text{error} \\
service(R \cdot \lambda x.P \star R_1) &= \lambda x.P \star R_1 \\
service(R \cdot \mathbf{side}(A)) &= \mathbf{side}(A) \\
service(R \cdot x{\mapsto}R_1) &= service(R) \\
service(R, x) &= R && \text{otherwise}
\end{aligned}$$

Table 6.7: Service Selection

third example, the side-effect is also empty. We return the value bound by y in x. If x does not contain the required binding, the empty form is returned as default value. Thus, the projection on y will never fail. In contrast, applying the service of equation 6.18 on a form that does not contain a binding for y raises an exception.

Let us now consider the most interesting case of specializing an application (equation 6.14 on Table 6.5). First — as is with extension — we split the agents A and B in the context of R'. This gives us two side-effects P_1 and P_2 and two lazy forms R_1 and R_2, respectively. The side-effects are composed in the right order, first P_1 then P_2.

The predicate $service : \mathcal{R} \to \mathcal{R}$ is used to determine the service of a lazy form. The term $service(R)$ denotes the service bound in a term R. For instance, $service(y{\mapsto}R_1 \cdot \lambda x.P \star R)$ is the abstraction $\lambda x.P \star R$. Determining the service that is associated with a lazy form R is similar to *project*. The recursive definition is given in Table 6.7. If R is the empty form we raise an error. If the lazy form is an extension of any form R extended with a service, service selection yields this service. If the lazy form is an extension with a binding, service selection recursively proceeds. In any other case, we cannot determine the abstraction and $service(R)$ denotes R.

Now, there are three possibilities depending on the functor of the application: The functor may be a referentially transparent service, it may be unknown, or it may be a service containing side-effects.

1. The functor is a referentially transparent service $\lambda x.\epsilon \star R_3$, i.e., its side-effect is empty. We inline the application by replacing it with the body R_3 where all x are replaced by the concrete argument R_2. The lazy form is $R_3[x/R_2]$.

As an example consider the service of example 6.17 which we invoke with the binding $(y{\mapsto}a)$. We split the application in a context that binds a:

$$\begin{aligned} split((\lambda x.\underbrace{(y{\mapsto}\epsilon \cdot x; y))}_{R_3}\underbrace{(y{\mapsto}a)}_{R_2}, a{\mapsto}a) &= (\epsilon, ((y{\mapsto}\epsilon \cdot x).y)[x/y{\mapsto}a]) \\ &= (\epsilon, a) \end{aligned}$$

The *split* function has specialized the application.

2. When the functor of the application cannot be determined at specialization time we proceed as follows. Consider the application $f()$ where we know nothing about f. We have:
$$split(f(), f{\mapsto}f) \;=\; (y_1{\mapsto}f(), y_1)$$
Since we know nothing about the functor, we put the application into the side-effect and bind it to a unique label y_1. The lazy-form of the application refers to the side effect y_1. Splitting the application explicitly states that $f()$ may contain a side-effect.

3. We consider the last case which is the most important one. We use an example to explain what happens. Assume the application $f()$ appears within an abstraction where f is the passed argument. We have:
$$split(\lambda f.f(), \epsilon) \;=\; (\epsilon, \lambda f.y_1{\mapsto}f() \star y_1) \tag{6.19}$$
Now we apply this abstraction on a form F. In the side effect and the lazy form we have to replace the variable f with the concrete argument F. The substitution yields $y_1{\mapsto}F()$ and y_1.

We need to ensure that we can refer to the result of this application even if we invoke the same abstraction several times. For that we nest the side-effects for each application with a unique label. Consequently, the lazy form has to lookup the result in the nested form by using a projection. Using such a unique label y_2, the side effect of the above application is $y_2{\mapsto}(y_1{\mapsto}F())$ and the lazy form $y_2.y_1$.

The function *nest* does the nesting of side-effects. The term $nest(R, x, P)$ is R where all y that are defined in P are replaced by the projection $x.y$. For instance, if P is the side-effect $y_1{\mapsto}P' \cdot y_2{\mapsto}P''$ then $nest(R, x, P) = R[y_1/x.y_1][y_2/x.y_2]$. Since the labels in P are distinct, the order of the substitution does not matter. The predicate *nest* is recursively defined on P:

$$\begin{aligned} nest(R, x, \epsilon) &= R \\ nest(R, x, P \cdot y{\mapsto}Q) &= nest(R[x/x.y], x, P) \end{aligned}$$

Consider the following agent A. It defines a service f which calls a service g. The service f is applied twice, once on the empty form and once on the form u.

$$A \;=\; \mathbf{R} \cdot f{\mapsto}\lambda x.(c{\mapsto}(\underbrace{g(a{\mapsto}x)}_{y_1})); a{\mapsto}\underbrace{f()}_{y_2} \cdot b{\mapsto}\underbrace{fu}_{y_3}$$

The agent A contains three static applications. We associate unique identifier $y_{1...3}$ with each invocation. Let r be the initial context that contains the bindings for the unknown forms $r = g{\mapsto}g \cdot u{\mapsto}u$. Splitting the three applications yields:

$$\begin{aligned} split(g(a{\mapsto}x), r \cdot x{\mapsto}x) &= (y_1{\mapsto}g(a{\mapsto}x), c{\mapsto}y_1) \\ split(f(), r \cdot f{\mapsto}(...)) &= (y_2{\mapsto}(y_1{\mapsto}g(a{\mapsto}\epsilon)), c{\mapsto}y_2.y_1) \\ split(fu, r \cdot f{\mapsto}(...)) &= (y_3{\mapsto}(y_1{\mapsto}g(a{\mapsto}u)), c{\mapsto}y_3.y_1) \end{aligned}$$

which gives

$$\begin{aligned} split(A, r) = (&y_2{\mapsto}(y_1{\mapsto}g(a{\mapsto}\epsilon)) \cdot y_3{\mapsto}(y_1{\mapsto}g(a{\mapsto}u)), \\ &a{\mapsto}(c{\mapsto}y_2.y_1) \cdot b{\mapsto}(c{\mapsto}y_3.y_1)) \end{aligned}$$

The partial evaluation inlined f and binds the side-effects to y_2 and y_3, respectively.

Observe that the nesting of side-effects ensures that we can access the side-effects from within the lazy form expression. If we apply the partial evaluation algorithm twice on the above expression, the nested side effects and projection are specialized:

$$split(combine(split(A, r)), r) = (y_1{\mapsto}g(a{\mapsto}\epsilon) \cdot y_2{\mapsto}g(a{\mapsto}u), a{\mapsto}(c{\mapsto}y_1) \cdot b{\mapsto}(c{\mapsto}y_2))$$

However, applying *split* twice does not linearize all nested side-effects since recursive service applications would introduce new nested side-effects at each specialization step.

This concludes the predicate *split* and the partial evaluation algorithm.

6.4 Correctness

We show that the partial evaluation algorithm is correct and terminates for all expressions. While termination is straightforward to show, correctness requires a bit of work. The important definition is that of referential transparency.

Termination. We can readily verify by structural induction on the domains for $\mathcal{A}, \mathcal{P}$ and $\mathcal{R}$ that the algorithm terminates. The important aspect for termination is the definition of the substitution given in Table 6.3. Consider the application xR_1 where we replace x with a user defined abstraction. For example

$$(xR_1)[x/\lambda z.P \star R] = (\lambda z.P \star R)R_1[x/\lambda z.P \star R]$$

It might be tempting to define the result of such a substitution as the result of splitting the application, as we have done for projection.

However, as the following example shows, this may lead to an infinite loop during the specialization process. Consider the term xx where we substitute the service $\lambda y.y_1{\mapsto}yy \star y_1$ for x. When we split the substitute term, the process loops since the substitute contains another instance of the same expression.

Correctness. This property specifies that any closed agent is behaviourally equivalent to its specialized agent. This means:

$$partial(A) \approx A \qquad \text{for } A \text{ closed} \tag{6.20}$$

In order to prove this equation, we show by induction over A that for all functional agents A and lazy form expressions R, the following holds

$$combine(split(A, R)) \approx embed(R); A \tag{6.21}$$

Then, equation 6.20 is a special case of equation 6.21 where R is the empty form. However, in order to prove the induction steps for this equation we need a stronger property, namely that for all A and R, there are two agents A_1 and A_2 such that:

$$combine(split(A, R)) \approx A_1; A_2$$

and all free labels in A_2 are defined by A_1 and A_2 does not contain any applications which cause side-effects or undefined projections. Whenever A_1 reduces to a barb with value F, there exists a form value G such that the expression $F; A_2$ is equivalent to G. The formal definition of this property is that A_2 is referentially transparent in A_1.

Definition 6.1 *A Piccola agent B is* referentially transparent *in an agent A, if for any agent C and names $\tilde{c}$ with $\nu\tilde{c}.C \mid A \Downarrow$, written as canonical agent:*

$$\nu\tilde{c}.(C \mid A) \;\Rightarrow\; \nu\tilde{c}'.(M_1 \mid \ldots \mid M_n \mid A_1 \mid \ldots \mid A_{k-1} \mid F)$$

there exists a form G such that:

$$F; B \approx G$$

The fact that B is referentially transparent within A is written $A \vdash B$. $\epsilon \vdash B$ is written as $\vdash B$.

Referential transparency formalizes the idea behind lazy forms. Whenever A is reduced to a barb with value F, the agent $F; B$ is equivalent to a form G. In other words, when A reduces to F then $A; B$ reduces to G. This notion rules out the possibility of B containing a side-effect. It also guarantees that all required labels of B are defined by A.

The word *all* in the above definition is important. It is not enough to find an equivalent G just for some possible reductions. For instance $c(x \mapsto \epsilon) \mid c() \mid c?; x \rightarrow c() \mid x \mapsto (); x \approx c() \mid \epsilon$. But $c(x \mapsto \epsilon) \mid c() \mid c? \not\vdash x$.

Obviously, all forms are referentially transparent, thus $\vdash F$ for any form F. We can prove by induction on A that $split(A, R)$ generates tuples that are referentially transparent. If $split(A, R) = (P, R)$ then $combine'(P) \vdash embed(R)$. The formal proof is in Appendix D.

6.5 New-state Services

In this section we present an enhancement to the partial evaluation algorithm. The enhancement adds special splitting rules for *new-state* services. A new-state service is not referentially transparent but it can be deferred, since its evaluation has no direct side-effect. Invoking the new-state service returns a different result each time it is invoked. Examples of new-state services are **new** and **L**.

Consider the program

```
a =
    'ch = newChannel()    # neither used nor returned
    b = 7
```

This program has the same behaviour as "`a = b = 7`" since the channel is never used. In the rest of this section we extend the partial evaluation algorithm so that it is capable to perform specializations of this kind.

We first extend the grammar for lazy forms with new-state services:

$$R ::= \ldots \mid \mathbf{newState}(A)$$

and change the predicates *embed* and *split* as follows:

$$\begin{aligned} \mathit{embed}(\mathbf{newState}(A)) &= A \\ \mathit{split}(\mathbf{L}, R) &= (\epsilon, \mathbf{newState}(\mathbf{L})) \\ \mathit{split}(\mathbf{new}, R) &= (\epsilon, \mathbf{newState}(\mathbf{new})) \end{aligned}$$

We extend the notions of free variables and substitution accordingly.

The equation for splitting applications, equation 6.14 on Table 6.5, does not need to be changed. We split applications with new-state services in the same way as unknown functors. For instance:

$$\mathit{split}(\mathbf{new}(), R) = (y {\mapsto} \mathbf{newState}(\mathbf{new})(), y)$$

The enhanced partial evaluation algorithm $\mathit{partial}' : \mathcal{A} \rightarrow \mathcal{A}$ is then defined as:

$$\mathit{partial}'(A) = \mathit{combine}(\mathit{strip}(\mathit{split}(A, \epsilon))) \tag{6.22}$$

The function *strip* removes applications of new-state services that are not needed. In order to explain *strip*, consider the splitting of the following agent A:

$$\begin{aligned} A = {} & \mathbf{R} \cdot a {\mapsto} \mathbf{new}(); \\ & \mathbf{R} \cdot b {\mapsto} \mathbf{new}(); \\ & \mathit{result} {\mapsto} (a; \mathit{send})() \end{aligned}$$

Evaluating A creates two fresh channels by using the new-state service **new** and binds them to a and b, respectively. It returns the result of invoking the *send* service of the channel a.

Splitting A yields:

$$\begin{aligned} \mathit{split}(A) = {} & y_1 {\mapsto} \mathbf{newState}(\mathbf{new})() \cdot \\ & y_2 {\mapsto} \mathbf{newState}(\mathbf{new})() \cdot \\ & y_3 {\mapsto} y_1.\mathit{send}(), \\ & \mathit{result} {\mapsto} y_3 \end{aligned}$$

Observe that we never refer to y_2 in the expression, which will always have the form $y {\mapsto} \mathbf{newState}(A) R$.

The function $\mathit{strip} : (\mathcal{P} \times \mathcal{R}) \rightarrow (\mathcal{P} \times \mathcal{R})$ is defined in Table 6.8. The technical difficulty is to ensure proper working of *strip* with nested side-effects. For that purpose the function *strip* uses a helper predicate strip' which carries an additional argument Q denoting the scope of

$$
\begin{aligned}
strip(P,R) &= strip'(P,R), R\\[4pt]
strip' &: \mathcal{P} \times \mathcal{Q} \to \mathcal{P}\\
strip'(\epsilon, Q) &= \epsilon\\
strip'(P \cdot x{\mapsto}R_1R_2), Q) &= strip'(P, Q \cdot R_1 \cdot R_2) \cdot x{\mapsto}R_1R_2\\
strip'(P \cdot x{\mapsto}R.y), Q) &= strip'(P, Q \cdot R) \cdot x{\mapsto}R.y\\
strip'(P \cdot x{\mapsto}P'), Q) &= strip'(P, P' \cdot Q) \cdot x{\mapsto}strip'(P', undo(Q,x))\\
strip'(P \cdot x{\mapsto}\mathbf{newState}(A)R, Q) &= \begin{cases} strip'(P, Q \cdot R) \cdot x{\mapsto}\mathbf{newState}(A)R & \text{if } x \in fv(Q)\\ strip'(P,Q) & \text{otherwise}\end{cases}\\[4pt]
undo &: \mathcal{Q} \times \mathcal{L} \to \mathcal{Q}\\
undo(x.y, x) &= y\\
undo(y, x) &= y \qquad\qquad\qquad\qquad \text{if } x \neq y\\
undo(\epsilon, x) &= \epsilon\\
undo(R.y, x) &= undo(R,x).y \qquad\qquad \text{if } R \neq x\\
undo(R_1R_2, x) &= undo(R_1,x)\ undo(R_2,x)\\
undo(R_1 \cdot R_2, x) &= undo(R_1,x) \cdot undo(R_2,x)\\
undo(y{\mapsto}R, x) &= y{\mapsto}undo(R,x)\\
undo(\lambda y.P \star R, x) &= \lambda y.undo(P,x) \star undo(R,x)\\
undo(\mathbf{newState}(A), x) &= \mathbf{newState}(A)\\
undo(\mathbf{side}(A), x) &= \mathbf{side}(A)
\end{aligned}
$$

Table 6.8: Stripping new-state services

the side-effect. The function *strip*$'$ is recursively defined for side-effect terms. The scope Q is the term where the effect might be looked up. Consider the stripping of a side-effect extended with a binding $x{\mapsto}\mathbf{newState}(A)R$. If the scope Q does not contain x free, then we can drop the binding and continue recursively. If the scope refers to x, then we must keep the binding and continue recursively. In this latter case, we have to add R to the scope where side-effects may be used.

The purpose of *undo* is to undo nesting of side-effects by replacing $x.y$ by y when traversing a nested binding $x{\mapsto}Q$. The predicate *undo* is transparent except for projection, i.e., $\mathit{undo}(x.y, x) = y$. In this case the projection $x.y$ is replaced by y since we have to undo the nesting of the label x.

As expected, *strip* removes the binding y_2 in our example:

$$\begin{aligned} \mathit{strip}(\mathit{split}(A)) = \;& y_1{\mapsto}\mathbf{newState}(\mathbf{new})()\cdot \\ & y_3{\mapsto}y_1.\mathit{send}(), \\ & \mathit{result}{\mapsto}y_3 \end{aligned}$$

6.6 Constant Folding

Constant folding is an optimisation technique that replaces a call of a function with known arguments with the result of that call. In this section we demonstrate this technique by presenting an extension to the partial evaluation algorithm. We show how to improve it with respect to deterministic applications of form inspection. An application of $\mathbf{L}$ is deterministic, if the argument is the empty form, a single service, or a binding. In these cases we know the result of the application and can replace the application with its result. Other constant applications like arithmetic or string operations can be integrated into the partial evaluation framework in a similar way.

In order to inline the result of, say $\mathbf{L}(x{\mapsto}R)$, we need to extend the grammar for lazy forms in Table 6.1 with the service hide_x for hiding a label x. Consequently we extend the grammar for lazy forms with a variant where a label is hidden: $\mathbf{hidden}(R, x)$ denotes the form R with the label x hidden. This is necessary in order to be able to write the result of $\mathbf{L}(x{\mapsto}R)$ as a lazy form expression. When R contains a variable we cannot evaluate the hiding of a label at specialization time. The semantics of projection and service selection from a hidden form is implemented by the helper functions *project* and *service* which have to be extended accordingly.

We extend the domains of lazy forms with hidden forms

$$R ::= \ldots \mid \mathbf{hidden}(R, x)$$

and adjust the notion of free variables and substitution accordingly:

$$\begin{aligned} \mathit{fv}(\mathbf{hidden}(R, x)) &= \mathit{fv}(R) \\ \mathit{labels}(\mathbf{hidden}(R, x)) &= \mathit{labels}(R)\backslash\{x\} \\ \mathbf{hidden}(R, y)[x/R'] &= \mathbf{hidden}(R[x/R'], y) \end{aligned}$$

Label hiding is transparent for the free labels and for substitution. The predicate *labels* respects the semantics of label hiding and removes x from the set of provided labels.

The helper predicates *project* and *service* are extended in the obvious way:

$$\begin{aligned}
project(\textbf{hidden}(R, x), x) &= \text{error} \\
project(\textbf{hidden}(R, y), x) &= project(R, x) && \text{if } x \neq y \\
service(\textbf{hidden}(R, x)) &= service(R)
\end{aligned}$$

Looking up a label x in a form where x is hidden yields an error. In any other case (if the hidden label is not the same as the projected label) projection and service selection proceeds recursively.

The function *embed* is extended to include hidden forms:

$$embed(\textbf{hidden}(R, x)) = hide_x\ embed(R)$$

The predicate *split* is modified in order to consider the primitive hide service. We add a new case to equation 6.14 in Table 6.5:

$$split(AB, R) = (P_1 \cdot P_2, \textbf{hidden}(R_2, x)) \qquad \text{if } service(R_1) = hide_x \tag{6.23}$$

where $(P_1, R_1) = split(A, R)$ and $(P_2, R_2) = split(B, R)$

Finally, we provide more cases for applications of inspect, depending on the value inspected:

$$split(AB, R) = \begin{cases} (P_1 \cdot P_2, \lambda x.split((x; isEmpty)\epsilon, x{\mapsto}x)) & \text{if } R_2 = \epsilon \\ (P_1 \cdot P_2, \lambda x.split((x; isService)\epsilon, x{\mapsto}x)) & \text{if } R_2 \text{ is a service} \\ (P_1 \cdot P_2, \lambda x.split((x; isLabel)E(y), x{\mapsto}x)) & \text{if } R_2 = (y{\mapsto}R_3) \text{ for any } R_3 \text{ and } y \\ (P_1 \cdot P_2, y = \textbf{newState}(\mathbf{L})R_2, y) & \text{otherwise} \end{cases} \tag{6.24}$$

where $(P_1, R_1) = split(A, R), (P_2, R_2) = split(B, R)$ and $service(R_1) = \mathbf{L}$. The first-class label $E(y)$ is the expression:

$$project{\mapsto}(\epsilon; \lambda x.(x; x)) \cdot hide{\mapsto}hide_x \cdot bind{\mapsto}(\epsilon; \lambda x.x{\mapsto}x)$$

Observe that we define the service for projection since $x{\mapsto}$ is not a functional agent (see Table 5.4).

With this extension to the partial evaluation algorithm we achieve better results when working with first class labels.

6.7 Discussion

We conclude the chapter and describe further applications of the partial evaluation algorithm and issues related to its implementation that are open for future work. The presentation of the algorithm in this chapter focussed on its correctness proof. We have omitted the aspect of code duplication, efficient implementation of the specialized code, code annotations for inlining, and using the information in a composition environment.

Code duplication and Efficiency. We have not discussed the effect of code duplication that is relevant in the context of partial evaluation. Assume a referentially transparent service that requires a lot of computation, like calculating the factorial. When we have an invocation of this service, say $s()$ this invocation may get duplicated during specialization.

Instead of duplicating, we can turn the application into a once function. A once function gets evaluated at most once — hence its name [Mey92]. When a once function is evaluated the first time, its result is cached and reused by later invocations. For more details on the implementation we refer the reader to [Sch01].

Using once functions also makes the special treatment of new-state services superfluous. In fact, once functions already defer the invocation as long as possible (and may omit them if not needed) and cache the result. Thus we can add referential transparent service applications to the grammar of lazy forms. Their implementation as once-functions guarantees that the resulting expression is correctly specialized.

In the implementation we may not implement *combine* as presented here. In fact, instead of taking unique labels for each application that may contain a side-effect and putting them into the root context, it is more appropriate to use arrays of a data-structure for forms and offset values to store the result of side-effect invocations and use the same (pre-calculated) offsets to fetch those values back.

Annotations. In the presentation of the algorithm in this chapter we inline any application. However, inlining duplicates the code. In order to keep storage consumption low we need a way to decide which services to inline and which not. Furthermore, we need annotations for host services to indicate whether they are referentially transparent or whether they are new-state services.

More work is required to see how one can optimize communication along channels, i.e., by inferring that a channel always contains the same value.

Composition environment. The partial evaluation algorithm can be used as a simple type checker for Piccola expressions. Although the associated type system is not complete, i.e., there may still be some runtime type errors, it can detect many mistakes that occur during programming.

Here is an example how to define the interface of a service. Define *newChannel* as

$$\begin{aligned}
&\mathit{newChannel}\mapsto\epsilon;\lambda x.\\
&\quad(\mathbf{new}();\\
&\quad\mathit{send}\mapsto\lambda x.(\mathit{send}(x);\epsilon)\\
&\quad\mathit{receive}\mapsto\lambda x.\mathit{receive}())
\end{aligned}$$

Any application of *newChannel* can then be specialized to:

- a form with exactly two services *send* and *receive*,
- applying the *send* service denotes the empty form.

Such an annotation can be considered as a type-declaration for the `newChannel` service.

We expect that the information provided by the algorithm should be explored in the context of an integrated composition environment. For instance, consider the following code fragment:

```
'size = newVar 17           # create a local variable
'initialize()               # call initialize and extend root
size.get()                  # which size to use?
```

The problem here is as follows. If `initialize()` returns a form containing a binding for `size`, we will refer to this binding in the last line. However, if in all contexts the service `initialize()` does not return a binding containing `size`, it is guaranteed that `size` on the last line refers to the local variable. If we know the implementation of `initialize` we can often give this guarantee. In fact, the partial evaluation algorithm infers the value of `size` in the last line. An integrated composition environment can refer to the value of that label. Such situations often occur when we use, for instance, wrappers that add default bindings.

Summary. We have presented a partial evaluation algorithm for Piccola. The algorithm separates expressions into side-effect terms and referentially transparent forms. We use this optimisation technique to remove the overhead associated with wrappers.

A key feature of the algorithm is that it removes the explicit namespaces of Piccola. The specialized code only depends on an initial root context. The same applies for the dynamic context that is passed around.

We have used the semantics of Piccola to prove correctness of the optimisation. However, more work is needed to know when to apply the optimisation to improve overall performance with respect to the code size.

The algorithm as presented here duplicates terms and as such costly computation. In a real implementation, duplication must be prohibited and links used to decrease the generated code size. Furthermore, we should cache the results and eliminate duplicated computation using once functions [Mey92].

Chapter 7

Composition Styles in Piccola

In the previous chapters we have defined the composition language Piccola. In this and the next chapter we see Piccola at work. The purpose of this chapter is to demonstrate the expressive power of Piccola and of forms in particular to model composition abstractions as first-class entities. We present composition and coordination abstractions that cannot be defined as pure functions. This validates our claim that agents, channels and forms support the definition of such abstractions. We also define composition styles as a declarative and high-level notion to express composition. The key requirement to implement styles is that we can express connectors as first-class abstractions. The two kinds of composition abstractions correspond to Piccola's composition layers (see Table 2.1) for library abstractions and composition styles, respectively.

An architectural style defines a vocabulary of components, connectors and rules governing composition. A composition style does the same, but not only at the architectural but also at the implementation level. A composition style defines a set of plug-compatible components, connectors to compose them and rules governing the composition. Not all architectural styles naturally map to a set of components-types and connectors. For instance we cannot define a generic connector for layered architectures.

We advocate the use of *component algebras* to capture the notion of a composition style. The component-types are the sorts, the connectors are the operators, and the rules governing the connection are represented by the signature of the algebra. The signature of a composition style defines a domain specific language. Defining and presenting a composition style as a grammar helps to communicate the expressiveness and constraints of the style.

This chapter is organized in three parts: First we define the notion of a composition style, then we present a longer example implementing a non-trivial composition style, and finally we present how inheritance and coordination abstractions are defined using forms, agents and channels. In Section 7.1 we contrast the low-level wiring view with the plugging view of composition abstractions. We illustrate the difference with the well understood style for push-flow filters in Section 7.2. We give two approaches to implement this style in Section 7.3. While one approach uses higher-order functions , the other approach uses explicit wires. In Section 7.4 we show that event notification can be implemented by using specific wires. This section starts our longer example that culminates in the implementation of a GUI-style in Piccola. By wrapping the GUI framework of Java into a domain specific language. In Section 7.5 we define an enhancement to the push-stream style by adding multiplexers, the merging of streams, and distributive filters. We combine this merge-stream style with a style for GUI composition in Section 7.6 and Section 7.7. This example validates our that forms support the definition of extensible composition abstractions. In Section 7.8 we define and show a style for mixin composition. Finally, in Section 7.9 we show how to make an aspect a first-class abstraction and in Section 7.10 we present how to encode control and coordination abstraction within Piccola.

7.1 Plugging versus Wiring

Software components are black-box abstractions that not only provide but also require services in order to function correctly. Building an application from components should then be a simple matter of wiring components, i.e., of connecting provided to required services. So — what is the problem?

The problem is that wiring is an inherently low-level activity that can lead to configuration errors. It does not scale up well. Wires are the gotos of component based programming. It is more natural to plug components. A plug is a set of unconnected wires. The wires of a plug are connected to a compatible socket in a single step. Composite components hide their connected interfaces and thus yield a new composite component.

We argue that a composition style can most naturally be captured by the signature of a many-sorted algebra [Wir90]. The component types are the sorts of the algebra. The operators are the connectors, and the rules are expressed by the signature. Let S be a set whose elements are called sorts. A signature Σ is a set of *operation symbols* σ of an arity $s_1, s_2, ..., s_n$ where $s_i \in S$ for $i \leq n$ and a rank s. An operation symbol is written $\sigma(s_1, s_2, ..., s_n) \rightarrow s$. If $n = 0$ then σ is called a constant symbol. A many-sorted algebra consists of a signature plus an *interpretation* of the signature into a set of elements, called the carrier. The interpretation associates each σ of the signature an actual operation on the set of the algebra.

Consider the following Unix script.

```
ls | grep test
```

The plugging-view declaratively says that the output of `ls` is further processed by `grep`. We are most of the time not so much concerned about the fact that there are two concurrent processes connected via buffer. The terms “`ls`” and “`grep test`” are constants of the algebra, the pipe symbol corresponds to the actual operation that connects two filters and denotes a new element of the algebra, i.e., a filter. In contrast, in a wiring view, this expression describes two processes “`ls`” and “`grep test`”. The output stream of “`ls`” is the input stream of the grep process. The input stream of the whole expression is the input stream of “`ls`”. The output stream of the `grep` process is the output stream of the whole script.

The plugging view is what is eminent from the script.

- Scripts are high-level specifications that make the composition of components explicit.
- Scripts are constraint by the grammar. Only well defined expressions are allowed. For instance, we cannot connect files in a Unix shell script with the pipe symbol.

The script makes it easier to reason about properties of the resulting configuration and it simplifies program understanding as the complexity of the underlying wires is hidden.

A signature is essentially a grammar for a domain specific language. The elements of the algebra are the actual components that are describable in the grammar. In fact, when

	provided services		*required services*	
Source			`push E:`	push element downstream
			`close:`	signal end of stream
Filter	`push E:`	push element downstream	`push E:`	push element downstream
	`close:`	signal end of stream	`close:`	signal end of stream
Sink	`push E:`	push element downstream		
	`close:`	signal end of stream		

Table 7.1: Provided and required services for the push-stream style

defining a composition style we actually design a specialized language. Such languages are also called little languages [Ben86].

In the following section we will present a simple composition style and contrast its wiring view with its plugging view.

7.2 A Push-Flow Style

In a pipe-and-filter architecture, each component has a set of inputs and a set of outputs. A component reads data on its inputs and produces output. We call these components *filters*. The connectors of this style connect an output of one filter to an input of another filter. They are called *pipes* [AAG93, SG96]. The rules for the style specify that filters are independent entities and may not share state with other filters. They may not know the identity of their respective upstream and downstream filters. A variant of the pipe-and-filter architecture is when each filter has at most one input and one output port. Components with no input port are called *sources*, components with no output port are called *sinks*. Filters in the narrow sense have one input and one output port. Such components can only be assembled in a linear sequence called a *pipeline*.

There are many examples of successful pipe-and-filter architectures: The best known are UNIX-processes connected by pipes and scripted by shell scripts [KP84][1]. Other usages are in compiling technology [ASU86], or distributed programming [BMR$^+$96, KBH$^+$01].

We use the pipe-and-filter style to explain the difference between wiring and plugging. One can classify filters whether they provide or require services for their input or output ports. In Unix, a filter process requires both services from the environment. We use a passive push filter providing an upstream `push` service and requiring a downstream `push` service. We also have a `close` operation to tear the stream down, flushing cached values and releasing allocated sources.

Sources, filters and sinks are distinguished by the different services they provide and require as shown in Table 7.1. Basically, filters and sinks provide `push` and `close` services to upstream components which use them to push data and to signal the end of the stream. Sources and filters require `push` and `close` services from downstream components to which they are connected (Figure 7.1).

[1]Most UNIX environments degenerate from the pure style. For instance, the "`ls`" command often behaves differently depending on whether the output stream is bound to a process or to the standard output. In the latter case, the output is formatted to multiple columns.

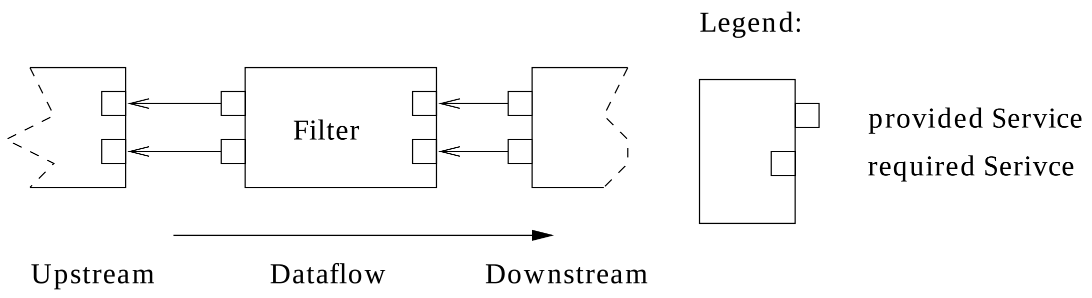

Figure 7.1: Wiring Streams

Arity	$\rightarrow$ *Rank*
Filter >> *Filter*	$\rightarrow$ *Filter*
Filter >> *Sink*	$\rightarrow$ *Sink*
Source >> *Filter*	$\rightarrow$ *Source*
Source >> *Sink*	$\rightarrow$ *Void*

Table 7.2: Signature for the Push-flow Style

We wire such components together by binding the provided services of a sink to the corresponding required services of a filter. Using a binding-oriented notation, as for instance in Darwin [EP93], this is written as:

```
bind aSink.push -- aFilter.push
bind aSink.close -- aFilter.close
```

This approach has the following limitations: First it does not scale up, since we may only wire one connection at the time. Second, it is error-prone as we might forget some bindings. Finally, the composite is not a component, i.e., a first-class value of the language.

A component algebra for the push-stream style is defined as follows. The sorts are *Filter*, *Source*, *Sink*, and *Void*. The signature consists of four overloaded operation symbols, see Table 7.2.

The operation symbol *Filter* >> *Sink* $\rightarrow$ *Sink* specifies that a composition of a filter with a sink denotes a sink. The operation symbol *Source* >> *Sink* $\rightarrow$ *Void* says that the connection of a source and a sink denotes a component of sort *Void*. As there are no operations defined on this sort, we cannot further compose it.

Properties of the algebra are imposed by equational specifications. For our style, we want that the >> -operator is associative, thus $(F_1 >> F_2) >> F_3$ denotes a component with the same behaviour as $F_1 >> (F_2 >> F_3)$.

The signature of a style is a level of abstraction above the notion of provided and required services. Instead of wiring provided and required services of filters at a low level, we have defined a little language to compose streams. In this language we can compose streams without paying attention to the individual services of the components. The high-level operators of the stream-language ensure that the services are bound correctly. The signature forbids bad configurations like composing two sinks.

An implementation of this style must provide the carrier and the operations to conform to the signature of Table 7.2. The components we have characterized with provided and required services are made consistent with the signature by associating with the >>-plug in F_1 >> F_2 the operation that wires the provided services of F_2 with the required services of F_1 and denotes the component with the unbound wires of both F_1 and F_2.

The signature hides the following implementation details:

- It abstracts the fact that the required `push` and `close` services of the left-hand side filter are bound by the provided services of the other filter.

- It abstracts the fact that the required and provided services of the composite filter are the unbound wires of the individual components.

Note that these details are far from being unimportant. In fact, in Chapter 8 we will see that these details may cause compositional mismatch. We will demonstrate how Piccola can be used to detect and repair these mismatches by using glue code.

The signature is an level above the provided and required services. It is possible to evolve the underlying component algebra without modifying scripts that use the style, for instance it might be necessary to add an initialize service that is connected when wiring. In later sections, we evolve the style be extending its composition behaviour, i.e. by defining richer wiring semantics for the same plugs.

7.3 Implementing Styles

We demonstrate how to evolve a wiring based implementation of components into a component algebra of the desired style. Remember that the design rational for Piccola is to be a generic scripting language. We want to encapsulate a set of components into a little language with the grammar given by the signature of the composition style. We want to provide a shallow embedding of this little language into Piccola, instead of implementing it as a new language from scratch.

There are two ways to perform a wiring: the *functional* and the *first-class wire* way. The functional way works by invoking services. When we invoke an abstraction or a service, we wire the required services of the component to the argument and we receive the provided services as result. In this approach composition means functional composition. The second approach uses first-class wires. A wire can be connected or unconnected. If attached to a provided service, invoking the wire invokes the connected service. Invoking an unconnected wire may block, do nothing, or raise an exception depending on the type of the wire.

In order to present these two approaches in detail, we first discuss the difference of component factories and instances.

7.3.1 Component Factories and Instances

When we speak about components we are often not so much concerned whether we work with instances or the factories to create them. With objects and classes the distinction is much more important. Classes are used to instantiate objects. In object-oriented languages, classes are often not first-class citizen. A composition language must support the creation of component factories as well as their instantiation.

In order to implement an algebra we have to know whether the elements of the algebra are instances or factories. Assuming that the elements are instances and we connect a filter and a sink, the composite denotes a new instance of a sink according to the signature. The underlying instances have changed their state from unconnected to connected. We cannot wire their sockets with other components, unless we first detach them.

In contrast, assume we have factories for sinks and filters. A composition defines a new composite factory. We can use the same factories over and over again.

At last, it is also useful to compose factories and instances. In the following subsection, we define the composition of a filter factory with a sink instance to be a new sink instance.

7.3.2 The Functional Way

We present an implementation of the push-stream style defined above. For our style, a sink does not have any required services. A *sink instance* is a form with a `push` and a `close` service. A *filter factory* is a form with an `apply` service. Invoking `apply` with a sink instantiates the filter and returns the provided services of a filter, i.e., a new sink. A *source factory* is modeled as a service `bindSink` that requires a sink and returns a *Void* component type. We use the empty form for void types, since no further composition is necessary for such types in the style.

Below is the definition of an output sink which prints all pushed data-elements and does noting on close, a filter that counts all pushed elements and prefixes each with its counter, and a source which emits “`Hello`” and “`World`”.

```
Stdout =
    push S: println S           # use println service
    close: ()                   # do nothing when the stream gets closed

CountFilter =
    apply Sink:
        'c = newCounter()       # instantiate local counter
        Sink
        push S: Sink.push
            asString(c.inc()) + ": " + S

HelloWorldSource =
    bindSink Sink:
        ''Sink.push "Hello"
        ''Sink.push "World"
        ''Sink.close()
```

Observe that we use form extension to construct the new sink in the filter `CountFilter`. The form returned by `apply` is the form `Sink` extended with a binding for `push`. The new binding overwrites the existing binding for `push` in the sink. The service `close` is inherited.

These components are composed using functional composition. The following script prints out the lines “`1: Hello`” and “`2: World`”.

```
Counted = CountFilter.apply Stdout
HelloWorldSource.bindSink Counted
```

Observe that a sink is an instance, while filters and sources are factories. We can use the filter several times, and get a new counter each time it is applied to a sink.

We wrap these components to support the style according to the signature of Table 7.2. The difficulty is that the >>-operator is overloaded. Since Piccola is dynamically typed, we cannot depend on the static type to resolve the overloading like in statically typed languages. We present two approaches to implement overloaded operators, the first uses run-time type checks, the second uses double dispatch.

Type checks. We use explicit type checks for the dispatch. As an example, we add the _>>_ binding to a filter. When invoked, we check whether the argument is a sink or another filter. It is a filter if it has a service `apply`. The following recursive service `addPlugToFilter` adds the _>>_ binding to a filter. We use it to wrap the `CountFilter` of above:

```
applyLabel = label(apply = ())            # first-class label
def addPlugToFilter Filter:
    Filter
    _>>_ Other:
        if (applyLabel.exists Other)      # explicit type check
            then: addPlugToFilter
                apply Sink: Filter.apply(Other.apply Sink)
            else: Filter.apply Other

CountFilter1 = addPlugToFilter CountFilter
```

The component `CountFilter1` can now be used according to the signature. For instance, `CountFilter1 >> Stdout` denotes a new sink component that counts pushed data-elements. Adaption of the other component types is similar.

Performing run-time checks is considered bad practice since it makes the code brittle for changes. Therefore, we present a better approach in the next paragraph.

Double Dispatch. By using double dispatch we can avoid the run-time type checks [Bec97, Ing86]. Double dispatch works by delegating a _>>_ call to a service of the right-hand side argument. The delegate service contains the type of of the left-hand side as part of its name.

Thus, the protocol for our three component types must look like:

```
aSink =
    close: ...                            # as before
    push S: ...                           # as before
    prefixSource Source: Source.bindSink(close = close, push = push)
    prefixFilter Filter: Filter.apply(close = close, push = push)

aFilter =
    apply Sink: ...                       # as before
    _>>_ Other: Other.prefixFilter (apply = apply)
    prefixSource Source:                  # return new composite source
    prefixFilter Filter:                  # return new composite filter

aSource =
    bindSink Sink: ...                    # as before
    _>>_ Other: Other.prefixSource (bindSink = bindSink)
```

```
asSink Sink:
    prefixFilter Filter: Filter.apply Sink
    prefixSource Source: Source.bindSink Sink   # Source >> Sink
    Sink

def asFilter Filter:
    apply Sink: asSink
        Filter.apply Sink                        # Filter >> Sink
    prefixFilter FilterL: asFilter
        apply Sink:                              # Filter >> Filter
            FilterL.apply(apply Sink)
    _>>_ R: R.prefixFilter (apply = apply)
    prefixSource Source: asSource                # Source >> Filter
        bindSink Sink: Source.bindSink(Filter.apply Sink)
    Filter
    apply = apply

asSource Source:
    _>>_ R: R.prefixSource Source
    Source
```

Figure 7.2: Double dispatch for Push-stream style

For instance, assume we compose the filter with the sink by writing "`aFilter >> aSink`". In that case, the service `aSink.prefixFilter(apply = ...)` gets called. The sink invokes `apply` and passes itself, i.e., the form that consists of the `push` and `close` service to the filter. Similar, we call the service `bindSink` on the source when the sink is prefixed with a source.

Instead of manually adding this double dispatch protocol to each component, which would be clumsy, we can use the genericity of forms and define *wrappers* that add it to any given component. The wrappers are given in Figure 7.2. The wrapper `asSink` adds the needed `prefixSource` and `prefixFilter` bindings to a bare sink. A bare sink is a form that provides the `close` and `push` services but does not necessarily contain bindings to support the composition style. The wrapper `asFilter` adds the >>-plug and the needed `prefix`-bindings to a bare filter, and `asSource` adds the >>-plug to a bare source.

When we say *add* a binding to, for instance, a sink this can mean two different things. Consider the wrapper `asSink`. It returns the form

```
prefixFilter Filter: Filter.apply Sink
prefixSource Source: Source.bindSink Sink
Sink
```

If `Sink` does not have a binding `prefixFilter` or `prefixSource`, these two bindings are provided as a *default*. If `Sink` contains bindings with these names the bindings in `Sink` overwrite the default bindings. Default bindings support (later) evolution of the style. In Section 7.7 we will make use of this possibility.

In contrast the following wrapper:

```
asSink1 Sink:
    Sink
    prefixFilter Filter: Filter.apply Sink
    prefixSource Source: Source.bindSink Sink
```

overwrites existing bindings for the two `prefix`-services. If the form `Sink` contains bindings with these names, we see the ones defined in the `asSink1` wrapper. If neither `prefixFilter` nor `prefixSource` are defined, then both `asSink`-wrappers behave the same.

In object-oriented modeling, an overwriting wrapper is analogous to a subclass that overwrites `prefixFilter`. The defaulting wrapper corresponds to a superclass that provides default methods for `prefixFilter` and `prefixSource` which can be specialized in subclasses.

Observe that the wrapper `asFilter` defines default bindings to the passed `Filter` and overrides `apply`. The reason for this is that we want to extend the protocol later on, but we also want to ensure that `apply` gets modified by the generic wrapper. The new `apply` service uses `asSink` to ensure that the returned sink from the original `apply` service fits the double-dispatch protocol and thus the composition style.

We can implement the style with three wrappers, one for each sort in the composition style. We do not need to modify the underlying components. Note that no wrapper is needed for the *Void* sort.

Connecting Filters is Associative. We have imposed that `>>` is associative for any component types. By using the fact that beta-reduction is a valid law (see Section 3.8) we inline functional applications. We assume that F_i are components wrapped with one of the above wrappers, and that their bare component does not contain bindings that interfere with the double-dispatch protocol. For instance, a bare sink may not contain `prefixSource` or `prefixFilter` bindings.

First we consider *instances*. The term F_1 >> F_2 >> F_3 is an instance provided F_3 is sink. To be a valid term, F_1 is either a source or a filter, F_2 must be a filter in any case. Assume F_1 is a source. Then the expression F_1 >> F_2 >> F_3 is

$$F_1\texttt{.bindSink}(F_2\texttt{.apply}\ F_3)$$

independent of the order of parentheses. If F_1 is a filter, then the whole expression becomes

$$\texttt{asSink}(F_1\texttt{.apply}(F_2\texttt{.apply}\ F_3))$$

and is again independent of how we put parentheses.

Now, consider the case where we have *factories*. In that case, F_3 must be a filter. Again, we have two possibilities depending on the type of F_1. If F_1 is a filter, then the filter denoted by F_1 >> F_2 >> F_3 is:

```
asFilter(apply Sink:  F1.apply(F2.apply(F3.apply Sink))
```

and if F_1 is a source, the composite source is:

```
asSource(bindSink Sink:  F1.bindSink(F2.apply(F3.apply Sink))
```

and both composite are independent of the parentheses. The operator is `>>` is associative and our implementation fulfills the laws imposed by the specified composition style.

In Section 7.5 we will reuse these wrappers for a style which supports the merging of sources. There, the precondition that the bare filters may not contain bindings that interfere with the protocol does not hold and the filters are not associative. If we need to ensure associativity we must change the wrappers so that they overwrite bindings.

7.3.3 First-Class Wiring

For the push-stream style we could use functions to bind the required services. Two necessary conditions must hold to do so:

- The topology of the architecture is not changed at runtime. Thus we do not need to dynamically re-bind any wiring.
- We can give an order in which the *instances* are created. In our style, we start with a sink that provides `push` and `close` and we then apply the filters to this stream and finally bind the resulting sink to a source.

However, these conditions are not always met. When the functional approach is not appropriate to establish the wirings, we use first-class wires. This approach allows us to incrementally wire instances or to change the wiring at runtime.

A simple first-class wire allows us to use the required service bound to it. It works analogous to a future: invoking a connected wire invokes the service it has been bound to; invoking a free wire delays the client until the connection is established. An implementation of a wire is based on a single channel:

```
newWire:
    'ch = newReadChannel()              # A channel with a non-destructive read
    bind = ch.send                      # Bind a service
    \Argument: ch.read() Argument       # Invoke service
```

The channel will store the provided service. This implementation is not safe since multiple bindings are allowed, but it is enough for explanatory purposes. We assume that `bind` is called at most once. In Chapter 8 we will use generic glue to formalize and enforce this requirement.

In Piccola, we use forms as interfaces to components. However, a form does only expose a set of services as bindings that are provided by the component. We propose the use of the following convention how a form can give access to the required services of a *component instance*: the provided services of a component are the bindings in the form, the required service are available in a nested form bound by `required`. The task of a connector is to create a new form that contains the unbound wires of both components and hides the connected services.

The script in Figure 7.3 defines a push-stream source and a counter filter following this convention. A source does not provide any services (see Table 7.1), the value of the form `mySource` contains the single binding `required` with two free wires available in the nested form. Note that invoking the service `run` returns the empty form, and the quoted `required` statement makes the required services available in the local context. The wires bound in the nested `required` form need to be connected in order for the source to operate properly.

```
'requiredSink:                     # service to create the wires
    push = newWire()               # required push
    close = newWire()              # required close
mySource =
    required = requiredSink()
    'required                      # make required namespace available
    ''run
        do:                        # run main agent
            push "Hello"
            close ()
myFilter =
    required = requiredSink()
    'required
    'c = newCounter()
    push S: push (asString c.inc()) + ": " + S
    close: close()
```

Figure 7.3: Pushfilters with first-class wires

What is the behaviour of the `mySource` instance? The agent representing it is specified as a `do` block and passed to `run`. It calls `required.push` to invoke the required push service. But this service blocks, unless the wiring is established. Thus, the agent representing the behaviour of `mySource` blocks until a sink or a filter is connected to it. Once connected, the agent writes "`Hello`" to the stream and closes the stream.

Note that we use the fact that the set of required services is available as a nested form. In the main agent of the component we extend the current namespace with all the required services and then use them as if they were defined as normal services in the context.

The filter `myFilter` provides a push service that, when invoked, uses the required `push` to forward any data-element and to prefix it with the current count.

Wiring `mySource` to the filter `myFilter` means binding the provided services of the filter to the required services of the source:

```
mySource.required.push.bind myFilter.push
mySource.required.close.bind myFilter.close
```

In order to abstract from the low-level wiring we extend any source or filter instance by adding the >>-plug. The adding is done by the following `addInstancePlug` wrapper:

```
def addInstancePlug Source:
    Source
    _>>_ Right:                    # define the >> connector
        'Source.required.push.bind Right.push
        'Source.required.push.close Right.close
        if (isEmpty (required = (), Right).required)
            then: ()
            else: addInstancePlug Right
```

The >>-plug performs the wiring and returns the composite. If the `Right` component does not require any services, i.e., it is a sink, the composite is the empty form. Otherwise, `Right` is a filter and we wrap the composite in turn so that it contains the >>-plug.

The following composition connects `mySource` with `myFilter`:

```
connected = addInstancePlug(mySource) >> myFilter
```

Thanks to the uniformity of the operator, we do not need operator overloading in the case of explicit wiring. In fact, the >>-operator always needs to wire the provided services of the right-hand side with the required services of the left hand side and returns the extension of the provided and required services with the connected wires removed. If we design a grammar and that uses double dispatch in combination with explicit wires, this indicates that the grammar is too overloaded and we should introduce more operation symbols. However, we do not see a technical reason to forbid the use of explicit wires and double dispatch for one style.

When using first-class wires, we naturally connect instances. This is manifested in the above example since there is only a single `counter`. In the functional approach, the factory service `apply` was a natural hook to allocate and initialize the counter for the new filter instance. With the wire approach we introduce factories explicitly.

In order to lift from *instance connectors* to component *factory connectors* we use functional abstraction. In our example, we convert `mySource` and `myFilter` into abstractions that create an instance when invoked. Consequently, the >>-plug builds up a new factory which, when invoked, instantiates both components and connects the instances appropriately.

```
def addPlug MyFactory:
    MyFactory
    _>>_ OtherFactory: addPlug
        new:
            'me = MyFactory.new()
            'other = (required = (), OtherFactory.new())
            ''me.required.close.bind other.close
            ''me.required.push.bind other.push
            me                              # my provided services
            required = other.required       # required of the other component

HelloSource = addPlug(new: ... )            # as before
CounterFilter = addPlug(new: ... )
```

Notice that we use `new` to instantiate filters. The wrapper `addPlug` adds the >>-plug to a source- or a filter-factory. Invoking the >>-plug creates a new composite factory. When we instantiate the composite factory, we instantiate both sub-instances, i.e., `me` and `other` and wire their services.

The factories are composed and used as follows:

```
(HelloSource >> CounterFilter >> output).new()
```

Using the explicit wiring approach, we can instantiate components in arbitrary order as illustrated by the following diagram. The upper part composes factories, the lower part composes instances. Composition takes place from left to right.

$$\begin{tikzcd}
\texttt{Factory1} >> \texttt{Factory2} \arrow[r] \arrow[d] & \texttt{CompositeFactory} \arrow[d] \\
\texttt{Instance1} >> \texttt{Instance2} \arrow[r] & \texttt{CompositeInstance}
\end{tikzcd}$$

This diagram commutes if the connectors for instances, i.e., `addInstancePlug` and for factories perform the same wiring. In our case, the following expressions denote the same composite source instance:

```
(HelloSource >> CountedFilter).new()
addInstancePlug(HelloSource.new()) >> CountedFilter.new()
```

7.3.4 Discussion

We have presented two ways to implement first-class connectors in Piccola. The two approaches have different strength and weaknesses:

- The functional view separates component instances and factories. A factory is a function that takes a set of provided services, instantiates and returns a new component. Creating composites corresponds to functional composition.
- Not all topologies can be decomposed into a functional wiring. We can't express loops and feedback with the functional way. First-class wires are more expressive.
- With first-class wires, we naturally plug instances. However, we can lift the plugs to factories.
- First-class wiring supports dynamic re-binding.

The reader should note that it is also possible to combine both approaches. An example will be given in Section 7.5.

We summarize how the definition of first-class connectors rely on the features of Piccola:

- We use **higher-order** services for functional composition.
- First-class wires are **unified functions and forms**. Invoking a first-class wire invokes the attached service. Nested bindings support attaching the provided service. Richer models of wires offer more services, e.g., to detach wires.
- The implementation of wires uses **channels** to represent the state of the wire, i.e., whether it is connected or not.
- **User-defined operators** capture the algebraic flavour of the component algebra. Notice that operators are syntactic sugar of the Piccola language.

In an object-oriented language we use objects as component instances. We can implement signatures by message sending and operator-overloading. However, we cannot smoothly move to factories, since classes are often not first-class elements of the language and they do not support higher-order functional composition.

To overcome this drawback, composition has to be implemented by object-oriented composition techniques. For instance, we can consider a subclass of a sink as a filter. The filter overwrites the corresponding `push` method and uses `super.push()` to push data-elements downstream. The disadvantage of subclasses is that they are static. A more flexible approach is supported by mixins. A filter is a mixin that eventually is composed with a sink. In both cases however, the inheritance hierarchy obscures the data-flow of the stream. The architecture is less explicit in the code compared with a script that uses the algebraic composition style.

7.4 Event Wiring

In the previous section we used a simple wire that supports the connection of a single service. Event notification [BCTW96] is a richer variant of first-class wiring. Using the event notification schema, one or more participants transmit and receive messages in response to events. An event might be: 'the user pressed a button'. The participant that transmits the message is called the *informer*. The component that receives the event is called a *listener*. The listener registers itself on a registrar, and the informer notifies the registrar when the event occurred. The registrar forwards the event notification to all registered listeners. This pattern of interaction is known as the observer or publish-subscribe pattern and the registrar facility is also referred to as a bus [GHJV95].

Wiring is the process of registering a listener to an informer. While the listener provides a service to be called by the informer as a response to the event, the informer requires a service. But different to the push-stream style, the informer is also operational when no or several services are wired.

In many frameworks for GUI composition the informer behaves differently depending on the number of registered listeners. For instance Microsoft's Foundation class library MFC[Kru97] grays out buttons and menu-items when no listener is attached. The Java Beans model [Mor97] supports an upper bound (often equals to one) to the number of registered listeners.

A first-class wire is a registrar component. It is created by:

```
newRegistrar:
    'listeners = []
    bind Listener:  ''listeners.add Listener
    \Event:
        listeners.forEach(do Listener: Listener Event)
```

We assume that `[]` returns a new mutable list which has an `add` and a `forEach` service. When the event is raised it is forwarded to all registered listeners in turn.

As an example, an observable point is created by:

```
newObservablePoint X:
    required = changed = newRegistrar()     # required service
    'x = newVar (x = 0, X).x                # Defaults to 0
    getX = x.get
    setX nx:
        ''x <- nx                           # change x element
        ''required.changed (arg = nx, event = "X Changed") # notify event
    ...
```

We wire a listener to a point by binding a service to the required `changed` service. The `newObservablePoint` is a factory for observable points.

In the following sections we will use this wiring and encapsulate it inside a GUI composition style. In Section 7.9 we will present a wrapper that adds the observable aspect to arbitrary instances.

7.5 A Merge-Push Style

In this section we extend the push-stream style with a new filter sort *DistFilter* and with an operator to merge sources. We will use these special filters for our GUI-composition style in Section 7.7. This section demonstrates that the connectors for the push-style are truly extensible. As we will show, unifying defaults and overwriting as form extension is the key enabling feature for extensibility.

We add the following operators to the signature of the push-flow style in Table 7.2:

$$\begin{array}{ll} \texttt{newMultiplexer}\;() & \rightarrow \mathit{Source} \\ \texttt{merge}\;\mathit{Source}\;\mathit{Source} & \rightarrow \mathit{Source} \\ \texttt{distributive}\;\;\mathit{Filter} & \rightarrow \mathit{DistFilter} \end{array}$$

We want that distributive filters distribute over merged sources. When we compose such a filter with a merged source, it gets composed with the sub-sources and the corresponding composite sources are merged again:

$$\begin{array}{l} (\,\texttt{merge}\;\mathit{Soure}_1\;\mathit{Source}_2) >> \mathit{DistFilter} = \\ \qquad \texttt{merge}\;(\mathit{Soure}_1 >> \mathit{DistFilter})\;(\mathit{Soure}_2 >> \mathit{DistFilter}) \end{array}$$

Note that this law does not hold for arbitrary filters. Assume a filter that counts each element. In the first case, the filter knows the total sum of all element pushed so far, in the second case, both filters only know how many data elements were pushed from the individual source. If the filter is specified by a referentially transparent service however, then the above equation is valid. In this case, the filter has no state and it cannot have a notion of history.

We present the implementation of these operators.

Multiplexers. Recall from Section 7.2 that a push source does not provide any services and that it requires `push` and `close`. We cannot create an instance of a source from scratch without providing the required services in the functional way. We can, however, create a stream multiplexer that uses explicit wirings. A multiplexer *provides* `push` and `close` — thus it is a sink; it also allows sinks to be connected — thus it is also a source.

When we use the multiplexer as a sink and no other sinks are attached to it, the multiplexer multiplexes to nowhere and looses all data-elements.

The service `newMultiplexer` on Figure 7.4 creates multiplexers. A multiplexer provides `push` and `close` as discussed. We use the `asSink` wrapper of Figure 7.2 to add the double dispatch protocol for sinks. We implement the protocol for sources, i.e., the >>-plug manually.

```
newMultiplexer X: asSink
    X
    close = newRegistrar()
    push = newRegistrar()
    bindSink Sink:
        close.bind Sink.close
        push.bind Sink.push
    _>>_ R: R.prefixSource (X, bindSink = bindSink)
```

Figure 7.4: A Multiplexer for Push-Streams

Distributive Filters. Recall that a filter has the service `prefixSource` needed for the double dispatch. This service is called when a filter is composed with a source. A distributive filter however must behave differently when composed with a source. Therefore, we add an additional binding to the filter that provides this specific behaviour:

```
def distributive Filter: asFilter
    Filter                      # inherit Filter operations
    prefixMergedSource X:       # compose with a merged source
        'Filter = distributive Filter
        merge (X.s1 >> Filter) (X.s2 >> Filter)
```

A distributive filter inherits the provided services of its bare filter and adds the service `prefixMergedSource`. When calling this service, we distribute the filter and attach it to both sources and merge the resulting two sources. We apply `distributive` recursively in case one of the sources `X.s1` or `X.s2` is also merged.

Merging sources is implemented by calling `prefixMergedSource` on distributive filters and `prefixSource` on normal filters and on sinks.

```
merge s1 s2:
    bindSink Stream:            # wire both streams
        s1.bindSink Stream
        s2.bindSink Stream
    _>>_ R:
        'default: R.prefixSource(bindSink = bindSink)
        (prefixMergedSource = default, R).prefixMergedSource
            s1 = s1
            s2 = s2
```

The `>>`-plug works as follows. If the component `R` does have a `prefixMergedSource` service, we call this service and pass both sources `s1` and `s2` as arguments. Otherwise, we use the default and call `prefixSource` by passing the `bindSink` service so that the sink or the filter can attach itself to the source.

Discussion. We have added a new sort *DistFilter* to the stream signature. Basically, we did not need to adapt the double-dispatch protocol. Changing the type of filters and streams would imply changing the wrappers `asFilter` and `asSink` given in Figure 7.2. The extensibility of forms allows us to add additional bindings to a filter and therefore assume default bindings if these bindings are absent. We can evolve the push-stream style without adapting existing components. The connectors implemented for the original push-style are extensible and can be reused without change.

Contrast the situation using object-oriented modeling: we would experience a type-reuse problem. On one hand, distributive filters are a subclass of filters, since they have additional methods. On the other hand, filters must also provide `prefixMergedSource` so that ordinary sources can prefix the filter. This means that we either have to modify the filter class or that we have to type cast in the merging source code. Furthermore, note that extending the interface of the filters with `prefixMergedSource` makes the interface richer leading to a proliferation of messages understood.

The solution offered by forms is that the merge service can provide a default if the plugged filter does not have a `prefixMergedSource`. We use form extension to specify this default.

7.6 GUI Composition

There are many languages that include expressions for doing graphical layout like PIC, or TeX. These languages support the composition paradigm of shapes which are composed into bigger shapes. Two or more shapes are composed to yield a larger shape in a single operation.

There are also many object-oriented frameworks that contain classes to define a graphical layout. The paradigm offered, for instance by GUI frameworks like Java Swing, however is more procedural and wiring based. The user creates a container element and then adds individual elements to the container. This paradigm makes it hard to visualize the final layout from the code. The wiring paradigm is supported by visual GUI builders. An empty area is created and then individual elements are dragged into the container. While the wiring approach may be suitable for visual environments, a composition style is more appropriate for the code.

This and the next section demonstrates how to wrap an existing object-oriented framework into a little language. We wrap parts of the Java GUI framework into a little language. This validates that Piccola is a generic scripting language: we can script the host components from the high-level scripting view defined by our imposed style. The little language itself will be a combination of an extended push-stream style for the event handling and a GUI composition style to be presented in this section. We define the GUI composition style in two steps. We first present a simple variant that only does layout. Then we combine the layout part with the merge-stream style.

Although we focus on some particular aspects of the Java GUI framework, similar ideas apply when dealing with other frameworks for different domains. The reason we choose the GUI framework of Java is because we assume that many are familiar with it, it is a non-trivial framework, and finally, we think that wrapping the framework as a little language makes the framework more accessible to novel-users. The little language is declarative whereas the object-oriented framework only provides procedural abstractions to do the wiring.

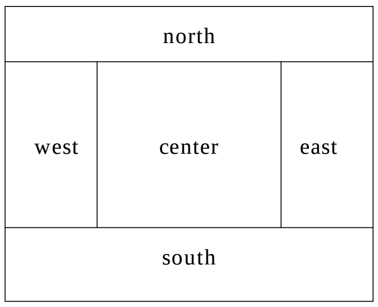

Figure 7.5: Regions of a Borderlayout

7.6.1 Simple GUI Layout

A family of layouts is specified by a so-called layout manager. The manager determines the location and the size of the contained components depending on the available space. For instance, a flow layout arranges components in a left-to-right, top-to-bottom flow, much like lines of text in a paragraph. A border layout lays out at most five components according to the five predefined regions north, south, east, west and center as shown in Figure 7.5. If the container is stretched, the height of the north and south and the width of the west and east region remains whereas the other areas are stretched equally. Complex layouts are implemented by nesting containers.

We focus on these two layouts and omit the other layouts of the Java framework for simplicity reasons. We define the signature for the layout as follows:

Arity	$\rightarrow$ *Rank*
`flow` [*Gui*, . . .]	$\rightarrow$ *Gui*
`border`(*region* = *Gui*, . . .)	$\rightarrow$ *Gui*

where *region* is any of `north, south, center, west` or `east`. The sorts are *Gui* for a GUI component, lists thereof, written `[`*Gui*`, ...]`, and configuration maps that associate regions to *Guis*, written `(north =` *Gui*`, center =` *Gui*`, ...)`.

We have deliberately chosen the signature so that implementing it by forms is straightforward. Clearly, a configuration map is a form where *Gui* elements are bound by the labels `north`, `south`, etc. For lists of *Guis* we use the available list defined in the core library in Piccola.

The GUI composition style is implemented by two connectors `flow` and `border`. The service `flow` takes a list of GUI components and returns a new flow layout component containing the elements. It creates the container, does the wiring, i.e., the adding of the components, and returns the container as new composite. In Java, the container is an instance of `java.awt.Panel` with the appropriate layout manager. The service `border` lays out the passed GUI components according to the regions specified into a border layout

Figure 7.6 compares an expression using the composition style with the explicit wiring paradigm. Both expressions define the same layout: three buttons on top, a text-area in the middle, and a status bar on the bottom, see Figure 7.7. While the component `plugged` uses the style, the component `wired` does the wiring manually, looking very similar to a raw Java implementation.

```
plugged = border
    north = flow [
        newButton(Label = "New", Name = "newButton")
        newButton(Label = "Run", Name = "runButton")
        newButton(Label = "Quit", Name = "quitButton")]
    center = newTextArea(Name = "inputArea")
    south = newTextField(Name = "statusBar")

wired =
    'panel1 = newFlowPanel()      # creates the panel and sets the manager
    ''panel1.add newButton(Label = "New", Name = "newButton")
    ''panel1.add newButton(Label = "Run", Name = "runButton")
    ''panel1.add newButton(Label = "Quit", Name = "quitButton")
    'panel2 = newBorderPanel()
    ''panel2.addCenter newTextArea(Name = "inputArea")
    ''panel2.addSouth newTextField(Name = "statusBar")
    ''panel2.addNorth panel1
    panel2
```

Figure 7.6: Plugging a GUI

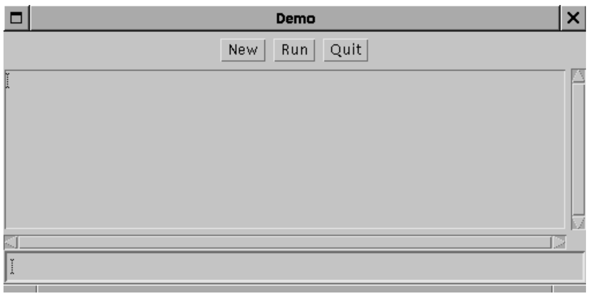

Figure 7.7: The Generated Layout

The differences are as follows:

- The layout of the plugged composite is specified as a single, declarative expression. In contrast, the code for adding the components for the `wired` composite is split into several statements. This is dangerous as things might be forgotten or elements added twice. For instance, in the Java framework, when we add several components to the same area in a border layout only the most recent one is visible in the composite.

- The wiring paradigm allows us to change the GUI at runtime, for instance, by replacing a button. The plugged expression does not support the change of the GUI at runtime. Of course this is possible, if we give access to the external Java components that implement the layout. However, changing a GUI at runtime is considered bad practice [Joh00].

- An expression of the little GUI language gives an idea of how the final output will look. Guessing the layout from the wired composite requires more analysis.

7.6.2 Using Default Arguments

We extend the GUI composition style to support the specification of additional properties for the individual sub-layouts. The extension validates our claim that generalizing arguments into forms supports extensible composition abstractions. A service assuming additional parameters can specify them as defaults or by overriding.

In our case we would like to change the default gap between the buttons. In the explicitly wired code we can set the horizontal gap by setting the property `Hgap` of the layout-manager for `panel1` as follows:

```
panel1.getLayout().setHgap 10
```

Here, the service `getLayout` returns the layout manager object associated with the panel. A flow layout manager accepts the method `setHgap` to set the horizontal gap.

In order to support properties we extend the signature of the GUI composition style. The properties are specified as an additional parameter to the `flow` and `border` services. We add a sort *P* for these properties. The new signature is:

Arity	$\rightarrow$ *Rank*
`flow( [` *Gui*, ... `] ,property` $=$ *P*)	$\rightarrow$ *Gui*
`border`(*region* = *Gui*, ... ,`property` $=$ *P*)	$\rightarrow$ *Gui*
propertyName = *Value*, ...	$\rightarrow$ *P*

where *propertyName* is a valid property for the layout manager and *Value* is its corresponding value. Often these values are strings and integers.

Handling these properties relies on the fact that arguments are encoded as forms. We define a *default* argument for the service `flow`. If the argument contains a nested form called `properties` we set these properties on the underlying layout manager. The default value for `properties` is the empty form, i.e., all properties should have their default value. Below is a fragment of Piccola code to set the properties of the local panel used in the service `flow`. The service is invoked by passing a list of *Gui* elements. The form `list` might contain a binding for `properties` as explained. The complete service `flow` is presented later in Figure 7.8.

```
'panel = newPanel()
''setProperties
    panel.getLayout()
    properties = (properties = (), List).properties # default is empty
```

The service `newPanel` creates a host object instance of the Java class `java.awt.Panel` and wraps it using the Piccola bridge. We set the properties in `panel.getLayout()` using the helper service `setProperties`.

7.6.3 Using First-class Labels

The service `setProperties` uses form inspection and first-class labels to call a different setter service on the underlying component for each binding in the form `property`. If we encounter a binding `Hgap = 10`, then the corresponding property must be set in the component, i.e., in the layout manager by calling the service `setHgap`.

We use a dispatch form to associate the setter services with a given property, or more precisely, with its name.

```
dispatchSet =
    Hgap = label(setHgap = ())
    Alignment = label(setAlignment = ())
    ...
```

The `setProperties` service iterates over the property form, looks up the setter service name in the dispatch form and invokes the service on the underlying component.

```
setProperties component: forEachLabel          # Iterate over all labels
    form = component.properties                # in the form
    do Label:                                  # and do
        setter = Label.project dispatchSet     # e.g., setter = setHgap
        value = Label.project component.properties  # the value to be bound
        setter.project component value         # invoke setter with the value
```

The reader familiar with the Java Beans model knows the convention how to encode properties of Beans. For each property `Prop` supported by a Java Bean, there is a setter method `setProp(Value)` and a getter method `getProp()` in the corresponding Bean class. If there is no setter method, then the property is read-only[2]. With the `dispatchSet` we can effectively follow this convention since there are only finitely many properties in the GUI framework defined. We do not need a service to create a label from a string.

Additional checks ensure that only valid properties for the underlying component are specified. If illegal properties are specified we can raise an exception or ignore them [BG97]. For our example, only the properties `Alignment`, `Vgap` and `Hgap` are valid for flow layouts, and only `Vgap` and `Hgap` are valid for border layouts.

[2]In the Java Beans Model, this is called a design pattern. In fact, for each Bean there is a corresponding Bean-Info class that defines the properties and the setter and getter methods. If a user sticks to the Java naming convention, the Bean-Info class can be automatically generated [Mor97].

7.7 Combining Styles

In this section we combine the merge-stream style and the GUI composition style into a little language for specifying graphical user interface dialogs. Due to form extension we can combine components of different styles into a coherent one. The key idea is to add the component of one style as a nested form to a component of another style.

We associate with each *Gui* a nested form `events` which is the source of events of the corresponding *Gui* element. When composing GUI elements we also merge the corresponding event sources. At the end a composite GUI, like, for instance the one defined in Figure 7.6 is a single event source of the merge-stream style. We can attach filters and sinks to this source and specify the behaviour of the GUI element in a high-level and declarative way.

As an example, we compose the plugged element into a dialog and define the behaviour of this dialog:

```
dialog = newFrame                 # puts the component into a frame
    properties =
        Title = "Demo"
        Name = "Demo"
    component = plugged
''dialog.events >> WindowClosing >>
                 newPushStream(\X: X.getSource().dispose())
''dialog.events >> Action >> debugOut
```

The first stream disposes the frame when the window is closing. The second stream prints a debug output for all action events generated by the dialog. Observe that the behaviour and layout of the dialog is specified declaratively using the high-level composition styles defined in the previous sections.

While this gives a little language to define graphical user interfaces, the mapping to the concrete Java framework is not ideal. There is a big overhead if we register a listener for all events a component can raise and push all these events into the corresponding event source — maybe only to filter out those events later in the downstream.

Distributive filters overcome this bottleneck. They can ”walk up” to the individual stream sources. We can compose the filter and the source and we remove unnecessary source-filter compositions. This is possible if the filter decides that it will remove all events generated by that component. Furthermore, instead of attaching the filter to the event source, we generate a listener object and attach it directly to the underlying Java object. Ideally, we fetch only those events that are actually needed, i.e., that have a sink attached. This speeds up the resulting application as if the events were manually low-level wired.

This section is structured as follows. In Section 7.7.1 we show how to merge GUI elements and their associated event-sources. In Section 7.7.2 we present the technical details necessary to create Java listeners and add them to Java GUI components. In Section 7.7.3 we give a coordination abstraction to adapt services as once-services. In Section 7.7.4 we give another example demonstrating the extensibility of the >>-plug. Finally, in Section 7.7.5 we summarize the lessons learned from the implementation of the GUI-stream combination.

7.7.1 Composing GUI elements

Having defined the merge operator for sources, we can use it when composing GUI elements. For that purpose, the connectors of the GUI composition style (e.g. `flow`) do not only

```
flow X:
    'panel = newPanel()
    ''setProperties                  # set specified properties
        panel.getLayout()
        properties = (properties = (), X).properties
    'wire events Component:
        ''panel.add(Component, type = "java.awt.Component")
        merge events Component.events # return a new merged push-source
    panel
    events = X.reduce wire panel.events
```

Figure 7.8: Creating a Flow Layout

layout the GUI components, but also reduce all event sources to a single source by merging them. The code for `flow` is given in Figure 7.8

We assume that the elements of the list passed to `flow` are GUI components and have a nested form `events` that is the source for events raised in the component. The GUI component are laid out and a new source is created by merging the individual `event` sources.

The function `reduce` is often used in functional programming. Reducing a list applies a binary service to each element in the list. The binary service is invoked once for each element of the list. The first argument is the result of the previous call, the second argument is the current list component. For instance, we can use `reduce` to compute the sum of a list:

```
println ([1, 2, 3].reduce (\x y: x + y) 0)     # prints 6
```

7.7.2 Adding Listeners

The Piccola-Java bridge defines a set of event-type listeners classes. These classes implement the necessary event type listener interface and forward the event method to Piccola. For instance, an action listener is created from Piccola as:

```
listener = Host.class("pi.piccola.bridge.GenericActionListener").new
    val1 = dynamic
    val2 = actionPerformed Event: ...          # the handler service
```

This code creates a Java object implementing the `java.awt.ActionListener` interface and makes it available as a form. The method `actionPerformed` is specified as the handler service from Piccola. This listener is added to a GUI component, e.g., to a button, by calling the appropriate add-method: "`button.addActionListener listener`".

In order to associate a push source with a GUI component, we create the appropriate listener object and add it to the Java GUI component. The required service for the listener is the push service of the multiplexer. For instance, the following Piccola code creates a multiplexer `events` and an action listener and attaches the listener to a given button object:

```
events = newMultiplexer()
''button.addActionListener
    Host.class("pi.piccola.bridge.GenericActionListener").new
        val1 = dynamic
        val2 = actionPerformed = events.push
```

We instantiate a listener for each event type the component understands and use the same push service for all listeners. Thus all events a component generates are pushed downstream starting at the same source.

7.7.3 Deferring the Wiring with Once Functions

It is important that we only add those listeners that are actually needed. Adding listeners to all events a component can raise would make the final application unacceptably slow. Normally only a few events are "interesting" to an application. For instance, every component supports the low-level mouse events but we are often not interested in these and would only like, for instance, a notification when the user clicked on a button.

We defer the adding of the listeners by attaching them lazily. This means that we instantiate and attach a listener only when the service events.bindSink is called, i.e., a sink is attached to the event source.

A once function is a function that is only evaluated the first time it is used and its result is then cached. Eiffel provides a special notion for once-functions [Mey92]. With channels we define a wrapper for services to make them once services.

```
once Service:
    'evaluated = newInitChannel false
    'cache = newReadChannel()
    \X: if (evaluated.receive(), ''evaluated.send true)
        then: cache.read()
        else:
            ''cache.send(Service X)   # set cache with the result of the invocation
            cache.read()              # return the cached value
```

The once function has two local channels. Initially, the channel evaluated is set to false. When the service is invoked for the first time, the wrapped service is invoked and the result cached by sending it along the cache channel. Observe that we send the value true to the channel evaluated thus preventing more evaluations. In case two concurrent invocations of the once functions occur, only one is evaluated, and the other is blocked on the cache channel until a value is available.

Using the once function, we defer instantiation and adding of the listener. The listeners are needed only if a sink is attached to the source associated with the GUI component. Thus all we need to do is to redefine the bindSink service so that it not only binds the stream to the source but also invokes the once function for adding all the listeners.

7.7.4 Specific Filter-GUI Composition

An event-filter is a filter that removes all pushed data-elements unless they are of a given kind. Instead of removing all elements at run-time, the filter can statically detect whether it is going to remove all events to be pushed downstream or not. If the filter will remove all elements, the composition of the source and the filter denotes a null source that never pushes any data.

There are two kinds of event-filters where the composition with GUI sources may yield a null source. A sourceSelector selects only the components that come from a particular GUI element. This allows us to select, for instance, one or several specific buttons inside a composite GUI. An eventTypeSelector selects only those components that are capable

```
# Component is a wrapped Java awt Component, extended with registerAll
asAwtComponent Component:
    'def self =
        Component
        events = newMultiplexer()            # event source
        'registerAllListeners = once         # once service
            \Component.registerAll           # to add all listeners
                do = events.push
                component = Component
        events.bindSink Stream:              # defer adding the listeners
            registerAllListeners()
            events.bindSink Stream
        events._>>_ R:                       # allow static filters to register
            'default: R.prefixSource self.events
            (register = default, R).register (self)
        set P: setProperties(Component, properties = P) # generic set
    self

Action = distributive                        # distributive, event-filter
    apply Sink:                              # apply filter at runtime
        Sink
        push Elem: if (Elem.getID() == ActionEvent.ID)
            then: Sink.push Elem
    register Comp:                           # register at compose time
        if isEmpty (addActionListener = (), Comp).addActionListener
            then: nullPushSource             # return null push source
            else:       # return a new source associated with the component
               'newComponent = asAwtComponent
                   component
                   registerAll X: X.component.addActionListener
                       newActionListener X.do
               newComponent.events
```

Figure 7.9: Wrapping GUI components

of generating the specific event type. Such a filter allows us to receive, for instance, only action events. In addition, an event type selector also knows how to create and add its own listener thus circumventing the invocation of the once service that would register all listeners supported by the GUI component.

Such a specific composition is implemented by a similar extension that we have already encountered in Section 7.5. There, we wanted a specific composition between merged sources and distributive filters. Now we need a specific composition between GUI event sources and static event-filters. For that purpose we overwrite the `>>`-plug of the event source and call `register` on the filter or sink being attached. If this component is not an event-filter, then we invoke the service `prefixSource` as usual for the double dispatch.

As an example, consider the event-filter `Action` shown in Figure 7.9. It provides a `register` service in addition to the `apply` service that all filters must have. The `register` service defines the composition behaviour when composed with event sources and the `apply` service defines the run-time behaviour.

The `register` service works as follows. We return a new AWT component with an event source when the event-filter decides that the GUI component it registers on can generate events of the required type. The 'new' component is simply the wrapped 'old' component with a more specific `registerAll` service. This specific service only adds the action listener interface when needed, i.e., when a sink will be added later on.

The filters for selecting components based on their names are similar.

The service `asAwtComponent` wraps any external GUI component of the AWT framework. It adds bindings for the event source and bindings for setting properties, as discussed in Section 7.6. Note that we assume that the component being wrapped is extended with a binding `registerAll` that instantiates and adds all listeners supported by that particular GUI component.

Finally, it should be noted that we have not implemented event-filter `Action` as presented in Figure 7.9. In fact, we abstract the information how to instantiate, add a specific listener, and detect whether the GUI component can generate such events. We do, however, not show this abstraction here since it does not give more insight into the definition and implementation of our little GUI language.

7.7.5 Summary

We have defined a little language that composes GUI elements and associates a single event source with each GUI. Composing GUI elements yields larger GUI structures and merged event sources. This style allows the programmer to separate the GUI layout from the behaviour. The behaviour does not need to be wired in terms of event listeners, but it can be declaratively specified by push streams.

While seeing a composite GUI as a single source of events is conceptually nice, we cannot directly map this style to the underlying Java framework. For that purpose we extended the push-stream style with distributive event-filters that find their way to the GUI components. Connecting an event-filter to a GUI component is scripted in the style efficiently wired in the underlying framework.

The composition abstractions presented in this and earlier sections are either connectors, wrappers, or coordination abstractions.

- **Connectors** plug components by performing the low-level wiring and constructing composite components. They are implemented using higher-order functions or explicit wires.

- **Generic wrappers** adapt components. In Piccola wrappers are services taking a form and, using form extension, provide default bindings or overwrite bindings. The wrappers we have presented add the connectors to bare components.

- We have seen the **coordination abstraction** once to adapt any service into a once service. The once service uses channels to store the state of the coordinated service, i.e., whether it has already been executed or not.

7.8 Mixins and Inheritance

In this section we demonstrate that inheritance can also be considered as a composition style. We do this by defining a mixin composition style. A *mixin* [Bra92, vLM96] is regarded as an abstract subclass, i.e., a class with an unspecified parent class. Applying a mixin to a class results in a new class which combines the methods of the class and the mixin. *Mixin composition* composes two mixins and denotes a new mixin. We consider a class as a degenerated mixin that does not refer to its parent mixin and use the word *complete* mixin as an acronym for a class.

Let us define a signature with the sort *Mixin* and the operator `*` to compose mixins:

$$Mixin * Mixin \rightarrow Mixin$$

In order to implement mixins we need a factory for mixins. A mixin is created by invoking the global service `Mixin` with the name of the mixin and a service `delta` that defines the methods of the mixin as bindings. The service `delta` is invoked from the `Mixin` abstraction with a parameter `bind` that in turn gives access to `self`, `super`, and the `class`. The class is the complete mixin being instantiated. Here is a complete mixin defining a `Point` class:

```
Point = Mixin
    name = "Point"
    delta P:
        X = newVar()                 # public variables
        Y = newVar()
        myAsString:                  # pretty Print
            'P.bind()
           "x = " + X.get() + ", y = " + Y.get()
        rep:                         # print debug info
            'P.bind()
            println(class.name + ".new(" + self + ")")
        initialize Init:             # initialize
            X.set Init.x
            Y.set Init.y
```

The members of the `Point` mixin are two variables `X` and `Y`, and the methods `myAsString`, `rep` and `initialize`. These members are defined as bindings in the body of the `delta` parameter of the mixin.

Consider the method definition `myAsString`. First of all, observe that the parameter `P` provides a service `bind`. We invoke this service inside any method and add it to the current root context using quotes. The method `myAsString` returns a pretty printed representation of the state of the point object. Note that `X` refers to a binding defined in self. We could also have written `self.X.get()`. The form `self` is returned by `P.bind()` and extends the current context.

The method `rep` prints the class name followed by a formatted representation of self. The identifier `self` refers to self which is a form containing a binding for `myAsString`. In Piccola any form can be converted to a String. If the form has a `myAsString` binding this service is invoked to get a string representation, otherwise a generic representation of all bindings is made.

Finally, the method `initialize` initializes the slots. Observe that we do not extend the current namespace with `self`, thus `X` and `Y` are *statically* bound.

The mixin `ColorMixin` defines the method `myAsString` and defines a variable for storing the `color`.

```
ColorMixin = Mixin
    name = "Colored"
    delta P:
        color = newVar()
        myAsString:         # overwrite
            'P.bind()
            asString(super) + ", color = " + color.get()
        initialize Init:
            'P.bind()
            super.initialize Init
            color.set((color = "Black", Init).color)
```

We use a default color if none is specified in the initialization parameter.

We compose both mixins and instantiate the composite mixin to get a color point object:

```
aColorPoint = (ColorMixin * Point).new
    x = 1
    y = 1
    color = "Yellow"

aColorPoint.rep()           # prints: "ColoredPoint.new(x = 1, y = 1, color = Yellow)
```

The mixin style demonstrates that we can model classical inheritance as a composition. Our model has the usual semantics of a dynamic `self` and `super` that gives access the static super-object. In Figure 7.10 the definition of the `Mixin` service is given.

We explain how `*` composes mixins and `P.bind()` establishes the correct self and super bindings. Let A be the body of the `delta` abstraction of the `Point` mixin:

```
'Point = Mixin
    name = "Point"
    delta P: ...            # A
aPoint = Point.new()
```

When we instantiate the `Point` mixin, A is evaluated as if it was defined in the context given in Figure 7.11. In A, `P.bind()` returns a form containing `self`, `super`, and `class`.

When mixins are composed, we traverse the mixin-expression tree in post-order and extend `self` with each bindings returned by `delta`. Thus, the binding of the most recently

```
def MixinCompose X:
    'def theMixin =                             # construct a composite mixin
        X.mixin
        name = X.mixin.name + X.parent.name
        _*_ R: MixinCompose(parent = R, mixin = theMixin)
        new InitParams:                         # instantiate theMixin
            'def self = theMixin.addDelta       # define object self
                self: self                      # a service to get self
                class = theMixin
                super = initialize = ()         # the dummy super object
            ''self.initialize InitParams
            self                                # return self
        addDelta Self:                          # post-traversal of the composite
            'superObject = X.parent.addDelta Self
            superObject
            X.mixin.addDelta(Self, super = superObject)
    theMixin

Mixin MixinFeatures: MixinCompose
    mixin =
        MixinFeatures
        addDelta Self: MixinFeatures.delta # add methods of this mixin
            bind:
                Self
                Self.self()                     # extend with self
                self = Self.self()              # extend with self binding
    parent =
        addDelta Self: Self.super
        name = ""
```

Figure 7.10: Mixin Abstraction

```
aPoint =
    'def self =
        initialize = ()
        'P = bind:
            class = ...                         # the complete mixin Point
            super = initialize: ()
            self
            self = self
        A                                       # here are calls to P.bind()
    ''self.initialize ()
    self
```

Figure 7.11: Instantiating Mixins

added mixin are first added to the form self that we are building up. Self denotes the fixed-point, whereas super denotes for each addDelta call the form build up so far.

Consider the mixin ColorMixin * Point. The agents *A* and *B* denote the body of the delta services of the Point and ColorMixin, respectively. The form aColorPoint is:

```
aColorPoint =
    'def self =
        'super =
            initialize = ()
            'P = bind:
                class = ColorMixin * Point
                super = initialize: ()
                self
                self = self
            A
        super
        'P = bind:
            class = ColorMixin * Point
            super = super
            self
            self = self
        B
    ''self.initialize (x = 1, y = 1, color = "Yellow")
    self
```

Observe how the result of the delta call of Point becomes the super object inside the delta call for ColorMixin.

There are a number of extensions we can apply to the Mixin abstraction. In his work of object encodings, Schneider presents a meta-class framework that allows to express different inheritance schemas like inner- or outer inheritance [Sch99].

The focus of the example in this section is on the mixin composition style implemented by the * operator and how P.bind() defines self and super objects.

Observe that the generalizing paradigm of forms gives us a very flexible mechanism for generalized inheritance [HK99]. Due to explicit namespaces we can use *static* or *dynamic* inheritance. The method Point.initialize binds X and Y statically, since we do not extend the namespace with P.bind(). Furthermore, observe that we can overwrite not only methods but any binding in the delta parameter. We can, for instance, overwrite the instance variables X and Y with another data-container.

7.9 Aspect Wrappers

In this section we demonstrate how to factor an aspect out of a component and make it a first-class entity. An aspect cross-cuts the functionality of some base components. We cannot implement the functionality of an aspect by adapting the component at a single or small set of locations.

We focus on black-box components. Thus we want to define a wrapper that adds an aspect to an existing component. As an example, assume we have a component for storing points:

```
def newPoint X:
    'x = newVar (x = 0, X).x
    'y = newVar (y = 0, X).y
    setX nx: ''x <- nx                # Change x component
    setY ny: ''y <- ny
    getX = x.get                      # Get x component
    getY = y.get
    _+_ OtherPoint: newPoint          # component-wise add
        x = getX() + OtherPoint.getX()
        y = getY() + OtherPoint.getY()
```

We further assume that we want to use this point component in a context where listeners need to be informed whenever the `X` or `Y` part changes. Thus we adapt such components to make them observable. Listeners can register on the point and get informed, each time either the x- or the y-component is changed.

To make a point observable, consider this ad-hoc wrapper:

```
def asObservablePoint Point:
    Point
    required.changed = newRegistrar()
    setX X:
        Point.setX X
        ''required.changed()
    setY Y:
        Point.setY Y
        ''required.changed()
    _+_ OtherPoint: asObservablePoint(Point + OtherPoint)
```

The following adaption is defined by the wrapper:

- We add a registrar facility `changed` where listeners can register. Refer to Section 7.4 for a description of the registrar facility.
- We wrap the setter services so that an event is forwarded to any registered listener.
- We adapt the +-operator so that its results is also wrapped.

Instead of making an ad-hoc wrapper just for point components we make the wrapper generic. We cannot use a simple nested form, since we have to define a recursive wrapper that also applies on instances returned by the +-service. For that purpose we need to specify which labels bind services that change the state and need to raise the event, and which services act as factory services where we have to adapt the result accordingly. In aspect-oriented terminology, the services to be adapted are the cross-points.

In Piccola, we can use label-sets to specify such wrappers. A label-set is defined by a form where the actual values of the bindings are of no concern. Using inspection we define a subform with respect to the label-set that contains just the labels that are in the set. For instance:

```
ls = labelSet(a = (), b = ())         # a labelset with labels a and b
println ls.subform(b = 7, c = 8)      # prints b = 7
```

```
def weaveObservable X:
    'component = (required = (), X.component)     # ensure required binding
    'component.required.changed = newRegistrar() # create registrar
    component
    map
        form = X.notify.subform X.component       # wrap all notifier services
        each service D:
            service D                             # invoke original service
            ''component.required.changed()        # raise event
    map
        form = X.factories.subform X.component    # wrap factories
        each service D: weaveObservable
            X                                     # with the same parameter
            component = service D                 # but a new component

newObservablePoint X: weaveObservable             # weave observable
    component = newPoint X
    notify = labelSet(setX = (), setY = ())       # these are the notifiers
    factories = labelSet(_+_ = ())                # these are the factories
```

Figure 7.12: A weaver for the observable aspect

The definition of the subform service for a given label-set is straightforward. We iterate over all the labels of the label-set and return only the bindings that are defined in the second form.

We use label-sets to *parameterize* the weaving of the point component with the observable aspect. We say which services are factories that have to be wrapped recursively and which services change the state of the point-component, i.e., which services raise the event. The weaver for the observable aspect and its parameterization are given in Figure 7.12. Observe that we use the label-sets to create subforms of all the setters and factory services of the component.

The global service `map` takes a form and a service `each`. It returns a form that has a binding `l = each l` for each label `l` in the set of labels of the passed form[3]. The service `map` uses inspect to iterate over all labels defined in a form (see Section 3.4.4).

The example presented here is necessarily simple. However, it illustrates the key concept of weaving. Forms allow us to parameterize services in a form and merge them with additional functionality. It should be noted that our weaver cannot use white-box adaption of the component. Thus, if someone later adapts the `newPoint` component, e.g., by adding a `initialize` service that sets both components to 0, this service must also be adapted. In our case self-sends are not captured. A richer style would be obtained by making mixins the base components.

The language AspectJ [KHH+01] allows sets of cross-points to be specified generically. For instance, we could define all setter methods to be of the name `set*` where the `*` acts as a wild-card. Using forms, we have to use explicit labels-sets since the Piccola calculus does not associate any structure with labels.

[3]In fact the service `mapSerivce` adapts only those bindings in the form that are actual services. We assume that the form consists of service bindings only and use the simpler service `map` (see Appendix E).

7.10 Control Abstractions

We have used the combination of higher-order functions and the unifying concept of forms to define compositional abstraction. In this section we demonstrate how the concepts of agents and channels allow the specification of abstractions that cross and encompass the flow of control of services.

The computational power needed to implement such abstractions comes from the underlying semantics that is rooted in the π-calculus. The π-calculus is designed for modeling evolving systems. The important thing to understand is that we can use *channels* to model the continuation of the flow of control. Instead of returning at the end of a service, an agent can send something along a channel and then stop. This mechanism is best illustrated if we implement service bodies as servers that have their own thread of control. When a client calls the server it also sends a reply channel on which the server can return its result. The client does some work in parallel to the request being handled or it immediately waits on the reply channel for the result. Using this schema we can implement a service as:

```
handle X:
    'result = ...                   # calculate the result
    X.replyChannel.send result      # and return it along the replyChannel
invoke Argument:
    'replyChannel = newChannel() # a new reply channel
    ''run(do: handle (X, replyChannel = replyChannel))# invoke handler
    replyChannel.receive()          # wait for the result
```

This is the schema applied when modeling functions and eager evaluation in the π-calculus. Making the reply channel explicit allows us to model various abstractions that change the flow of control. In the rest of this section we present two examples. In Section 7.10.1 we use an explicit continuation channel to model blocks. In Section 7.10.2 we not only have explicit continuation channels but also a context dependent policy. As an example we define an abstraction for exception handling.

7.10.1 Blocks

Consider a service that searches for a specific element in a collection. The collections provide abstractions to iterate over all elements. We can iterate over these elements and analyze each element. If we find what we were looking for, we would like to return this element but we first have to finish the loop over all elements. It would be more economic to jump out of the loop as soon as the desired element is found.

The following code is executed inside a block call:

```
block
    do break:
        collection.forEach
            do Elem: if (hasProperty Elem) (then: break Elem)
```

The global service `block` executes the passed `do break` service. It provides the `do` block with a service to break out of the block. When we call `break` inside the `do` block the main block returns.

```
stop: newChannel().receive()

block doBlock:
    'done = newChannel()
    'break X:                       # define local break abstraction
        done.send X                 # signal termination
        stop()
    run
        do:
            doBlock.do break        # call Block.do with break
            break()                 # if not stopped so far
    done.receive()
```

Figure 7.13: Definition of the block abstraction

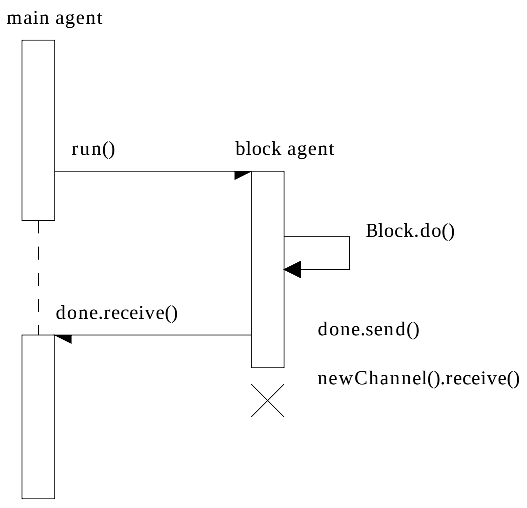

Figure 7.14: Sequence Diagram for a loop

```
OrJoin X:
    'receptor = newChannel()
    ''run (do: receptor.send X.left())
    ''run (do: receptor.send X.right())
    receptor.receive()

try Block:
    'exception = newChannel()
    OrJoin
        left: B.catch exception.receive()
        right:
            'dynamic.raise e:          # define a raise abstraction
                exception.send(e)
                stop()
            Block.do()

raise E: dynamic.raise E
```

Figure 7.15: Defining a try-catch Abstraction

Figure 7.13 shows the definition of the `block` abstraction and Figure 7.14 visualizes which agent is active at which time. Essentially the main agent is immediately blocked after running the `doBlock` agent when it tries to receive a value from the `done` channel. Similar, the block-agent immediately stops after sending a result along the `done` channel. For an external observer, the code cannot be distinguished from a single threaded implementation of the block.

7.10.2 Exception Handling

Another example of an abstraction that crosscuts the normal flow of computation is exception handling as provided in most modern object-oriented languages. The client sets up an exception context and this context is implicitly passed down the invocation stack when calling methods. If one of the methods raises an error the error is propagated upwards the calling stack until an appropriate handler is defined. If such a handler is found, the handler takes over control.

In Piccola, we can implement a try-catch abstraction without access to the invocation stack. The idea is similar to the generic block construct. The only difficulty is to make sure that the correct `raise` abstraction gets passed down the invocation stack and is invoked when the user wants to raise an exception. This is precisely what the `dynamic` namespace is for. The code for defining an exception handler is given in Figure 7.15 and Figure 7.16 visualizes the events when an exception is raised.

The `OrJoin` service takes two services and executes them concurrently. It returns the result of whatever service first terminates.

First, consider the case when no exception is raised during execution of the block. In this case, the second agent, i.e., the exception handler agent immediately blocks while trying to read from the `exception channel`. Since there is no exception raised, `Block.do()` and consequently the `OrJoin` invocation finally return. No other agents may have access to the

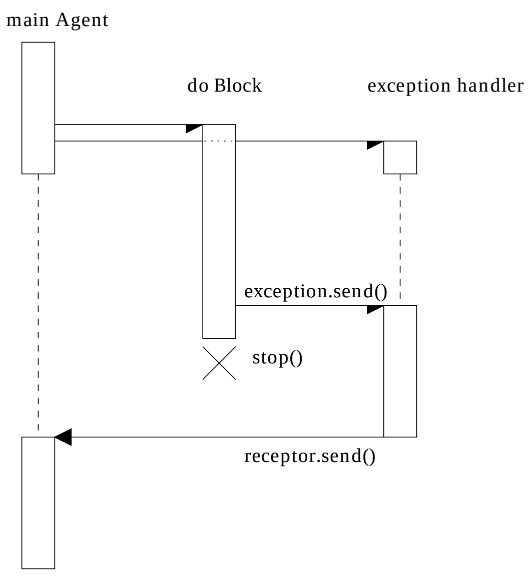

Figure 7.16: Sequence Diagram for Throwing an Exception

`exception` channel, thus the (blocked) exception handler agent will get garbage collected.

If, on the other hand, an exception is raised (see Figure 7.16) then at one point during execution of the `Block.do()` code the *global* `raise` abstraction is called with the dynamic context containing the most recently defined `raise` abstraction. Raise gets forwarded to the dynamic abstraction which first signals the exception and then stops by creating a fresh channel and trying to read from it. The exception handler agent will pick up the value from the `exception` channel, invoke the `catch` service with it and return, consequently terminating the or-join block.

7.11 Discussion

In this chapter we have demonstrated Piccola's expressive power to implement extensible composition abstractions. Figure 7.3 summarizes how the features of Piccola enable our ability to define reusable composition abstractions. The rows consist of the features of Piccola, the columns contain the composition abstraction.

- *Generic Wrappers* are services that adapt forms by adding bindings using form extension. The key enabling feature for wrappers are forms and the fact that they unify arguments and component interfaces. Generic wrappers may iterate over forms relying on the fact that forms are immutable and finitary. Iteration itself is defined using first-class labels and form inspection. Adaption is specified by higher-order services.

  These wrappers may prefix some bindings as *defaults* or *overwrite* bindings. The first approach is better suited for later evolution. The overwriting approach makes the style more closed ensuring certain composition laws. Both mechanism are available for free by form extension.

  The nature of forms as extensible records makes these abstractions *extensible*. We can

	Generic Wrappers	Connectors	Object Model	Coordination Abstractions	Context dependent Policies
Form extension	x		x		x
Immutable forms	x				x
Recursive forms			x		
Finitary form	x				
Form introspection	x				
Channels		x	x	x	
Agents				x	
Higher-order services	x	x	x	x	
Operators		x			
Explicit namespaces			x	x	x
Arguments as Forms	x	x			

Table 7.3: Features for Composition

specialize the behaviour of certain connectors by connecting additional wires when components are plugged. We have demonstrated this by evolving the push-stream style into the merge-stream-style and combining this style with the GUI-style.

- We have demonstrated two approaches to implement first-class *connectors*. We either use higher-order functions or we use explicit wires which itself are based on channels. Wires are generic in their arguments.

  Defining connectors as user-defined *operators* helps to make the architecture explicit and supports a declarative plugging view

- Forms can model classical inheritance in order to define an *object model* on top of Piccola. We have implemented a mixin composition style. Due to the nature of forms the achieved inheritance mechanism is generalized. Not only can we overwrite methods but also instance variables. In contrast, the language Self [US87] was specifically designed to unify the concept of methods and variables into slots.

  Fixed-points allow us to dynamically bind the meta-variable self to the object being defined. Recall from Chapter 5 that fixed-points are represented by channels.

- *Coordination* and *control abstraction* synchronize agents with each other via channels. Piccola uses a call-by-value evaluation strategy. Since agents and channels are explicit entities, we can model different evaluation strategies or richer models for functions like functions that may have an early return. Modeling coordination abstractions relies on the π-calculus foundation.

- The concept of explicit dynamic namespaces allows the programmer to define **context dependent policies**. As an example, we have presented the implementation of a try-catch abstraction.

We capture a set of connectors and components into a composition style. A composition style is a component algebra. It defines a number of sorts and operations on the sorts. The algebra associates each component to a sort; the connectors are the operation symbols of the signature. The signature allows us to express certain compositional rules. If the components are stateless component factories, then composition denotes component factories. If the components are instances then composition yields a composite instance and changes the state of one of the instances from an unconnected to a connected instance.

We have presented a longer example of adapting an object-oriented framework into a composition style. We argue that capturing the framework as a composition style reduces the size of the necessary documentation of the framework for black-box reuse. Contrast the signature of our style with the corresponding parts of the Java GUI framework that is described as an API, i.e., by listing all public methods and describing the classes. We claim that representing a composition style as a little language, with its compact representation of composition, considerably reduces the steep learning curve traditionally associated with object-oriented frameworks, due to the compact representation of composition.

In [AKN00] we have implemented a style for Actor computation [Agh86] and grouped actors. Actors are autonomous entities that provide a `receive` service and require the ability to `send` and `broadcast` messages. In the general case, actors are wired over a bus. A variation of this coordination style is when the bus itself enforces certain rules actors must obey [MU97]. The central operation in these styles is the composition of several actors with a bus.

Scripting an application over low-level wiring improves understandability and maintainability of the resulting application. The composition style allows certain invariants to be enforced at composition time. Most prominently, it forbids illegal composition, like connecting a sink to a filter.

This chapter demonstrated the expressiveness of Piccola. The next chapter presents the other requirement we have for a composition language, namely that it supports reasoning about composition and composition abstractions.

Chapter 8

Reasoning at the Language Level

This chapter demonstrates how we can effectively reason about composition abstractions at the Piccola language level. We do model checking on the states induced by Piccola programs. The example abstractions we will reason about come from the domain of glue code which is the second purpose of the chapter. We demonstrate that Piccola is expressive enough to define reusable glue abstractions.

Glue code is used to overcome compositional mismatches. A compositional mismatch occurs when components need to cooperate but their respective contracts do not match [DW99]. We use the term glue in this narrower sense here: other authors use the word glue simply for "putting things together" [Lum99].

Often, glue code is written for a particular mismatch. Due to restricted expressiveness of the used language we often cannot abstract the glue code from the particular situation and reuse it for similar mismatches. Consider an object for which we have to adapt many method names so that the object implements another interface. We can use delegation for this renaming. Although conceptually simple, writing such glue code is a tedious task, since we always have to reinvent the wheel and reimplement it. Due to the unifying concept of forms, we can abstract glue code in Piccola and define generic and reusable glue code.

There are various techniques to adapt a component. White-box techniques adapt a component by changing or overriding part of its internal specification. Black-box techniques only adapt interfaces, i.e., provided and required services [Bos99]. If the component does not provide additional hooks for specialization, we can only use black-box adaption. A *wrapper* packs the original component into a new one with a suitable interface.

In Piccola, a wrapper is a service that takes the unwrapped form, the *wrappee*, and returns the wrapped form. Form extension allows the wrapper either to overwrite services or to provide default services in the case the wrappee does not contain such bindings, as we have discussed in Section 7.3.

We will use generic glue code wrappers for two purposes.

- First, they are used to adapt components so that the components can cooperate in a context they were not designed for.
- Second, glue code wrappers specify the assumptions a component makes about its environment.

The following observation motivates the second usage of wrappers. If the context can provide the contract specified by the glue wrapper, then wrapping the component does not change the interaction patterns of the wrapped component. Thus we do not have to wrap it, since all its assumptions are already guaranteed.

In their classical paper on architectural mismatch, Garlan *et al.* [GAO95] describe that conflicting assumptions of components can have their root in the nature of components, i.e., the components assume a different control or data model, in the nature of the connectors by assuming a specific interaction pattern, in the global architecture, and in the order in which components are instantiated. The control model defines which component 'has' the main thread of control and it defines the event model. The interaction pattern specifies which part initiates an interaction, e.g., by an asynchronous call or by registering an event listener. In order to detect mismatches, it is crucial to make these assumptions explicit.

The runtime model of Piccola using agents and channels is designed to make such assumptions explicit. Recall the specification of a service call in Section 7.10. We can precisely specify the events that occur when a client invokes a service, i.e., the client makes a call and passes a restricted reply channel. Then he waits on that reply channel for the result. If we want to specify, for instance, an asynchronous call, we can introduce an explicit agent performing the invocation and let the main agent spawn this agent.

This chapter is structured as follows. In the first section we introduce a formalism to reason about the states and transitions a component or a wrapped component can have. As an example, we define a component that acts as non-blocking channel and we study its properties in a concurrent environment. In Section 8.2 we show how to adapt the component to make it thread safe and show how to lift this ad-hoc adaptation into reusable glue. In Section 8.3 we present the specification of a reader-writer policy in Piccola. In Section 8.4 we combine the push-style of the previous chapter with pull-style streams. We present a wrapper that adapts pull-style filters to work in a push-style. We then use our formalism to analyze a wrapped filter and to detect composition mismatches. Based on this knowledge, we then present an improved adapter.

8.1 Transition-based Reasoning

In this section we introduce a formalism to reason about Piccola programs. The formalism is based on the states and transitions a component can have. It uses some conventions to abbreviate the writing of these states.

Consider a component that offers a set of services that do not block, i.e., invoking a service will always eventually return a result without any further interaction with the component. The component assumes that its services are used in a mutually exclusive way. But it does not protect itself from concurrent accesses and contains race conditions.

As an example consider a non-blocking channel. If the channel is empty, it returns the empty form whenever a client tries to receive a value form it. If it is not empty, it returns an arbitrary value that has been sent along it. As long as it is not empty it behaves like an ordinary channel.

To implement a non-blocking channel we need a local counter and a private, blocking channel. Figure 8.1 contains the factory code for non-blocking channels. Sending a data element writes the data to the private channel and increments the counter. The behaviour of receiving a value from the channel depends on the counter. If the counter is zero, i.e., there are currently no values available, we return the empty form. Otherwise receiving decrements the counter and returns an element from the private, blocking channel.

```
newCounter:
    ’c = newInitChannel 0
    inc: ’’c.send c.receive() + 1
    dec: ’’c.send c.receive() - 1
    \:
        ’r = c.receive()
        ’’c.send r
        r
```

```
newNonBlockingChannel:
    ’blocking = newChannel()
    ’count = newCounter()
    send E:
        ’’count.inc()
        ’’blocking.send E
    receive:
        if count() == 0
            then: ()
            else:
                ’’count.dec()
                blocking.receive()
```

Figure 8.1: Specification of a non blocking channel

The non-blocking channel contains a race condition. Assume the channel `blocking` contains one value and the counter is 1 when two concurrent `receive` request are handled. Both requests (or their handling agents) see that the counter is 1, thus they both execute the `else` branch. Both decrement the counter and try to receive from the blocking channel. Obviously only one agent receives a value and the other remains blocked.

In order to formally analyze what happens, recall from Section 3.7 that a *state* in Piccola can be characterized as the set of messages (i.e., which values are written along which channels) and a set of running threads. Each thread is given by a thread context and the action it is going to perform next.

For closed agents the actions are either applications, projections, or receiving a value from a channel. For our purposes, we only want to consider the interesting actions. The interesting actions are (i) receiving a value and (ii) inspecting a form, i.e., applications where the functor is the inspect primitive and the argument is a form with more than two labels. Receiving is a side-effect action that can cause race-conditions.

Thus most applications and all projections are not interesting, since they have an equivalent expression that omits these transition. For instance, if we inspect the empty form $\mathbf{L}()$, we can replace this invocation by its result. This is permitted by the law:

$$\mathbf{L}() \approx \epsilon; \lambda x.(x; \mathit{isEmpty})()$$

In fact, any application can be computed effectively or inlined. For user defined abstractions this is proven in Section 3.8, for projections this is shown in Section 4.6. We assume here that there are no illegal projections.

We are only interested in keeping track of the states where all threads are going to perform an interesting action, i.e., receiving from a channel or inspecting a form. Since the thread context can be viewed as the program counter, we write the program counter as a comment into the Piccola code and use the comments as thread contexts. We also omit the writing of channel scopes and use different channel names.

To illustrate this formalism, let c be the channel `c` of the service `readTwo`:

```
readTwo:
    c.receive()
    # readTwo1
    c.receive()
    c.send()
```

An invocation `readTwo()` in parallel with with three messages cF, cG, and cH can reduce as follows:

$$\begin{aligned} cF \mid cG \mid cH \mid \texttt{readTwo}() &\approx cF \mid cG \mid cH \mid \widehat{\mathcal{E}}[c? \cdot c?] \\ &\rightarrow cG \mid cH \mid \widehat{\mathcal{E}}[F \cdot c?] \\ &\rightarrow cH \mid \widehat{\mathcal{E}}[F \cdot G] \\ &\approx cH \mid c() \mid F \cdot G \end{aligned}$$

The thread context $\widehat{\mathcal{E}}$ models the continuation of the `readTwo` service after the two values are received. In particular, the thread context sends the empty form along c when the term is evaluated. When writing the above reduction, we omit the thread context:

$$\begin{aligned} cF \mid cG \mid cH \mid \texttt{readTwo}() &\rightarrow cG \mid cH \mid \mathit{readTwo1} \\ &\rightarrow cH \mid c() \mid F \cdot G \end{aligned}$$

Sometimes we are only concerned with the possible state transitions and not with the concrete values. In this case we abstract the return values and write:

$$\begin{aligned} cF \mid cG \mid cH \mid \texttt{readTwo} &\rightarrow cG \mid cH \mid \mathit{readTwo1} \\ &\rightarrow cH \mid c() \end{aligned}$$

We only record any internal step before any receive request on a channel. It does not make any difference if we consider a configuration with a parallel agent that is going to send or if we consider the configuration already with the message. This is due to the asynchrony of the Piccola calculus [ACS96].

Analyze the Counter. We now show that the counter specified in Figure 8.1 has exactly n states. For that we consider the possible transitions of the counter when put in parallel with an arbitrary collection of parallel `inc`, `dec`, and λ-requests. We write λ-request for invoking the service of the counter itself.

The initial counter in parallel with an `inc` request evolves as:

$$c0 \mid \texttt{inc}() \rightarrow c1 \mid ()$$

There are no internal steps as the increment service first receives the old value from channel c and then resends the new (incremented) value.

By induction over the number of parallel `inc`, `dec` and λ-requests we show that the state of the counter is always cn. The induction base is established by `c.send 0` in the initialization. Each induction step consists of $cn \rightarrow c(n+1)$ for incrementing requests, of $cn \rightarrow c(n-1)$ for decrement operations, and $cn \rightarrow cn$ for λ-requests.

We summarize such arguments in a statement like: “the invariant is that there is always one number sent along the channel c”.

Modular reasoning. Whenever we use the counter we will employ a higher-level view on the state of component instead of looking at the (private) channels. In fact we can model the counter as a component with the states $\mathit{counter}(n)$ for any number n. The increment and decrement service modify the internal state and return the empty form, the λ-service returns the current value of the counter.

The following can be seen as the specification of the increment and decrement service of our counter:

$$\mathit{counter}(n) \mid \texttt{inc}() \rightarrow \mathit{counter}(n+1) \mid ()$$
$$\mathit{counter}(n) \mid \texttt{dec}() \rightarrow \mathit{counter}(n-1) \mid ()$$

Such a higher-level view of a state abstracts the internal channels. Note that we can only use such a higher level view, when all the channels are private. Otherwise an external agent may disrupt the internal protocol and invalidate the model of the component.

Such a high-level description of the component behaviour looks similar to the CHAM model [BB92] and to the join-calculus [FG96]. Left of an arrow is a soup of particles composed by the parallel composition operator. The particles evolve in one step to the right-hand side of the transition. The general CHAM rules require that the rule might be embedded in a larger soup of particles that does not evolve and that the parallel composition operator is commutative and associative.

In our case, these general CHAM rules are not given. The parallel composition operator is not commutative in Piccola. The main agent, i.e., the topmost right thread, must remain during reduction steps.

Analyze the Non-blocking Channel. We analyze the non-blocking channel. We start with a freshly initialized channel. Its initial state is $\mathit{counter}(0)$. In order to discover the different transitions of the non-blocking counter we add program marks as explained above:

```
receive:
    if counter() == 0
        then: ()
        else:
            # receive1
            ''counter.dec()
            # receive2
            d.receive()
```

We abstract the concrete values stored on the blocking channels. We write $\texttt{blocking}^n$ for n parallel messages $\texttt{blocking}\ F_i$ for $1 \leq i \leq n$. We also abstract the arguments and the return values when interacting with the non-blocking channels. We can infer the following transition for invoking `send` on the non-blocking channel.

$$\mathit{counter}(0) \mid \texttt{send} \rightarrow \mathit{counter}(1) \mid \texttt{blocking} \tag{8.1}$$

This transition says that the empty non-blocking channel becomes the non-blocking channel with one value within a single step

Invoking `receive` reduces as follows:

$$\begin{aligned} \mathit{counter}(1) \mid \texttt{blocking} \mid \texttt{receive} &\rightarrow \mathit{counter}(1) \mid \texttt{blocking} \mid \mathit{receive1} && (8.2)\\ &\rightarrow \mathit{counter}(0) \mid \texttt{blocking} \mid \mathit{receive2} && (8.3)\\ &\rightarrow \mathit{counter}(0) && (8.4)\end{aligned}$$

These transitions say what happens when a `receive` request is handled when the non-blocking channel contains one value. Handling such a request is not atomic. We have two intermediate steps. After the first step $\mathit{counter}(1) \mid \texttt{blocking} \mid \mathit{receive1}$ we have checked the counter value and chosen the `else` branch. We can understand the action *receive1* that that there is a (pending) thread waiting to progress at this point. The second step decrements the counter and the third receives a value from the channel `blocking`. After the third step the `receive` invocation returns.

The sequence of steps can interfere with further `send` or `receive` requests. Here is the reduction that uncovers the race condition informally explained above. Assume, the non-blocking channel holds one element and two `receive` requests are handled concurrently.

$$\begin{aligned}
\mathit{counter}(1) \mid \texttt{blocking} \mid \texttt{receive} \mid \texttt{receive} &\to \mathit{counter}(1) \mid \texttt{blocking} \mid \mathit{receive1} \mid \texttt{receive} \\
&\to \mathit{counter}(1) \mid \texttt{blocking} \mid \mathit{receive1} \mid \mathit{receive1} \\
&\to \mathit{counter}(0) \mid \texttt{blocking} \mid \mathit{receive1} \mid \mathit{receive2} \\
&\to \mathit{counter}(0) \mid \mathit{receive1} \\
&\to \mathit{counter}(-1) \mid \mathit{receive2}
\end{aligned}$$

This reduction leaves the second thread blocked while trying to read from the channel `blocking`. Thus our intended non-blocking channel blocks! The problem is that the both agents decided that the channel was not empty whereas when the second wants to grab the value from `blocking`, the channel was empty.

8.2 Reusable Glue Code

There are several ways to fix the implementation of the non-blocking channel introduced in the previous section. One possibility would be to use a different counter that returns its value together with a decrement operation. However, this solution implies changing the implementation of our non-blocking component.

A less invasive change is to adapt the non-blocking channel. In fact the channel implementation is correct assuming synchronized access. The contract required for any client of the channel is to ensure that there are no interleaving `receive` requests for it.

If we want to use the non-blocking channel in a context where we cannot guarantee synchronized access to `receive` we need to adapt this service by associating a lock channel with it. Each time `receive` is invoked, a value is received from the lock channel and when the `receive` service returns a value is written back to the lock channel. Initially one value is written along the channel. The lock channel acts as a binary semaphore.

Here is a factory for safe non-blocking channels. This factory wraps the previous, non-safe component and using ad-hoc glue code:

```
newSafeNonBlockingChannel:
    'nbChannel = newNonBlockingChannel()     # non-blocking channel
    'lock = newInitChannel()                 # The lock is available
    nbChannel
    receive:                                 # overwrite receive
        ''lock.receive()                     # lock it!
        nbChannel.receive()                  # invoke unsafe receive
        ''lock.send()                        # release lock
```

We instantiate the safe non-blocking channel and invoke `receive` on this instance. This leads to the following reduction. We write b for the blocking channel inside the non-blocking channel and l is the `lock` channel. We write `receive()` for the receive-request on the safe non-blocking channel and `nbChannel.receive()` for the receive-request on the underlying non-blocking channel.

$$\begin{aligned} l() \mid \mathit{counter}(1) \mid b \mid \texttt{receive}() &\rightarrow \mathit{counter}(1) \mid b \mid \texttt{nbChannel.receive}() \\ &\rightarrow \mathit{counter}(1) \mid b \mid \texttt{nbChannel}.\mathit{receive1} \\ &\rightarrow \mathit{counter}(0) \mid b \mid \texttt{nbChannel}.\mathit{receive2} \\ &\rightarrow l() \mid \mathit{counter}(0) \end{aligned}$$

If there is an additional receive request that might infer with the internal steps this request is blocked since it cannot grab the lock l.

We do not need to protect the service `send` of the non-blocking channel, since its state transition is already atomic. Note that the bindings `nbChannel` and `lock` are private. This means that no other agent has access to the service `nbChannel.receive` nor to the channel `lock`. By induction on the number of concurrent requests on the `safeNbChannel` we can derive that it maintains the following invariant:

$$\texttt{safeNbChannel}(n) \mid \texttt{blocking}^n$$

where $n \geq 0$. Since the channel `blocking` is private we omit it in the high-level description of the behaviour:

$$\begin{aligned} \texttt{safeNbChannel}(n) \mid \texttt{send} &\rightarrow \texttt{safeNbChannel}(n+1) \\ \texttt{safeNbChannel}(0) \mid \texttt{receive} &\rightarrow \texttt{safeNbChannel}(0) \\ \texttt{safeNbChannel}(n) \mid \texttt{receive} &\rightarrow \texttt{safeNbChannel}(n-1) \qquad \text{for } n > 0 \end{aligned}$$

There is no need to consider the actual values send and received in order to show that the `safeNbChannel` is in fact a safe non-blocking channel. This simplification reduces the number of states to be analyzed.

Due to the genericity of forms we can abstract the ad-hoc glue code leading to the following wrapper. The wrapper associates a locking channel with the wrapped service:

```
threadSafe Service:
    'lock = newInitChannel()        # The lock is available
    \Argument:
        ''lock.receive()            # lock it!
        Service Argument            # invoke original service
        ''lock.send()               # release lock
```

Now, `threadSafe Service` is a service with a local channel `lock`. While an agent is executing it, no value is written along `lock`. The `lock` is released when the invocation returns. When the service does not return, the adapted service remains blocked.

Instead of using the ad-hoc glue we could equally specify:

```
newSafeNonBlockingChannel:
    'nbChannel = newNonBlockingChannel()
    nbChannel
    receive = threadSafe nbChannel.receive
```

In order to apply the synchronization scheme of full synchronization to a set of services we associate the same `lock` for all services. The following service wraps all the service in a form:

```
synchronized F:
    'lock = newInitChannel()
    map
        form = F
        each Service Arg:
            ''lock.receive(), Service Arg, ''lock.send()
```

The service `threadSafe` is the single service version of synchronized. It holds that:

$$\texttt{threadSafe}\,\texttt{S} \approx \texttt{(synchronized(x = S).x)} \tag{8.5}$$

for any service `S`

The `synchronized` wrapper is idempotent, i.e.:

$$\texttt{synchronized}(\texttt{synchronized}\,\texttt{F}) \approx \texttt{synchronized}\,\texttt{F} \tag{8.6}$$

for any form `F`.

In the remainder of this section we prove law 8.6. Proving this law serves two purposes. Obviously it gives us confidence in the `synchronized` wrapper. Second and more important, it serves as an example for reasoning about unknown services. Reasoning about open system requires that we can derive properties of abstractions even though we have unbound variables, i.e., unknown services.

In the previous section it is sufficient to consider only actions in threads that receive from a channel and form inspection applications. Now, we cannot inline applications anymore in order to discover their internal structure and transitions because the behaviour of some services is unknown. Instead we model the invocations of any service s to which F gives access. For the following we write s for a service that is bound in F with a label `s`.

Since we know nothing about the behaviour of s we assume one of:

$$A \mid sG \rightarrow A' \mid H \tag{8.7}$$
$$A \mid sG \rightarrow A'' \mid \mathbf{0} \tag{8.8}$$

where A is the global state when the service is called and G is the argument to s. Equation 8.7 assumes that the invocation sG returns and we receive a value H and the system stays in state A'. If sG does not return (equation 8.8), either because it is blocked or loops, we have a new state A'' and a blocking result modeled by the null-agent.

Now, let $A_1 = \texttt{synchronized}(\texttt{synchronized}\ F)$ and $A_2 = \texttt{synchronized}\ F$. The locking channels are written a, b and c and they are different due to restriction from any other channel in the system. We show that $(A_1.s)G$ and $(A_2.s)G$ are bisimulations for all s in F and all arguments G.

We first analyze the transitions of A_1 and we assume that sG returns. Thus we have:

$$\begin{aligned} A \mid a() \mid b() \mid (A_1.s)G &\rightarrow A \mid b() \mid \widehat{\mathcal{E}}[b?] \\ &\rightarrow A \mid \widehat{\mathcal{E}}'[F.sG] \\ &\rightarrow A' \mid a() \mid b() \mid H \end{aligned}$$

where $\widehat{\mathcal{E}}$ is going to receive from the locking channel b , then invoking sG, and after reception of the result $\widehat{\mathcal{E}}$ is going to emit the messages $a()$ and $b()$. The thread context $\widehat{\mathcal{E}}'$ is the continuation of $\widehat{\mathcal{E}}$ after the receiving a value from channel b.

When we assume that the invocation sG does not return, the last transition of above is not possible and we have:

$$\begin{aligned} A \mid \widehat{\mathcal{E}}'[F.sG] &\rightarrow A'' \mid \widehat{\mathcal{E}}'[\mathbf{0}] \\ &\approx A'' \mid \mathbf{0} \end{aligned}$$

Assuming SG returns, we consider the transitions in agent A_2:

$$\begin{aligned} A \mid c() \mid (A_2.s)G &\rightarrow A \mid \widehat{\mathcal{E}}[F.sG] \\ &\rightarrow A' \mid c() \mid H \end{aligned}$$

and similarly when sG does not return. Both systems evolve bisimilar and are independent of A. Thus the reductions are independent of any context and law 8.6 holds.

Like in the previous section, we prefer to write such reductions without considering the global state and the thread context. We only keep track of where the thread continues when the external invocation returns.

For instance, invoking the double synchronized service of above:

```
service F:
    ''a.receive()
    # sync1
    ''b.receive()
    s F     # sync2
    ''b.send()
    ''a.send()
```

reduces by the following atomic transitions:

$$\begin{aligned} a() \mid \texttt{service}() &\rightarrow \mathit{sync1} \\ b() \mid \mathit{sync1} &\rightarrow \mathit{sync2}[sF] \\ \mathit{sync2}[G] &\rightarrow a() \mid b() \mid G \end{aligned}$$

We write $\mathit{sync2}[sF]$ for an invocation of an external service. Notice that such an invocation is on the right-hand side of a reduction arrow. When the service returns, it triggers $\mathit{send}[F]$. Note that this convention is possible since there is only a single (static) invocation of the service s. When we have multiple invocations of the service s we need to write the thread contexts to differentiate them.

The fact that we need two actions to model an external service invocation can also be explained by modeling the external service by an autonomous agent. In order to invoke the service, we send a message, and we then wait on the reply channel for the result. Once the result is available we consume the result-message and continue. This is described by the transitions that consumes $\mathit{sync2}[G]$.

```
newRWPolicy:
    'writers = newInitChannel()
    'readers = newInitReadChannel 0
    preReader:
        'r = readers.receive()
        if r == 0 (then: writers.receive())
        readers.send r + 1
    postReader:
        'r = readers.receive()
        if r == 1 (then: writers.send())
        readers.send r - 1
    preWriter = writers.receive
    postWriter = writers.send
```

Figure 8.2: A reader-writer policy

8.3 Reader-Writer Policy

In the previous section we have presented a wrapper to make components thread safe. In this section we demonstrate how to define more complex coordination abstractions. The reader-writer policy is a family of concurrency control designs that apply when any number of readers can be executing simultaneously as long as there are no writers, but writers require exclusive access.

While it is relatively straightforward to implement a reader-writer policy for a given set of reader and writer methods it is more problematic to implement it generically. For example, Lea [Lea99] gives a generic implementation of a reader-writer policy for Java using an abstract class. The implementation defines pre- and post-methods for both reader and writer methods. Using a template method [GHJV95] `doRead()`, the method `read()` executes `beforeRead()`, possibly blocking until the reader is allowed to enter its critical region. The critical region is implemented in `doRead()` and must be provided by a subclass. Then, a method `afterRead()` does some cleanup. By subclassing, a programmer provides its own implementation of `doRead()`. Unfortunately, it is not possible to subclass the read method so that it can take arguments or to have several different reader methods. If this is necessary, the programmer can add arbitrary new methods taking arguments, but then it is the programmer's responsibility to correctly call `beforeRead()` and `afterRead()` for each reading method and analogously for writer methods.

In contrast to Java which uses outer-inheritance, we may use BETA's inner inheritance [MMPN93] to implement the reader-writer policy. In BETA, a superclass implements the policy and reader-writer components are subclasses and specialize reader- and writer-methods accordingly. Inner inheritance overcomes the reuse problem with the arguments, but it still is an inheritance-based approach. It relies on correct subclassing, i.e., it is white-box reuse. The approach we present here, is completely wrapper based. As such our solution is completely black-box.

Figure 8.2 shows the definition for a reader-writer policy in Piccola. The reader-writer

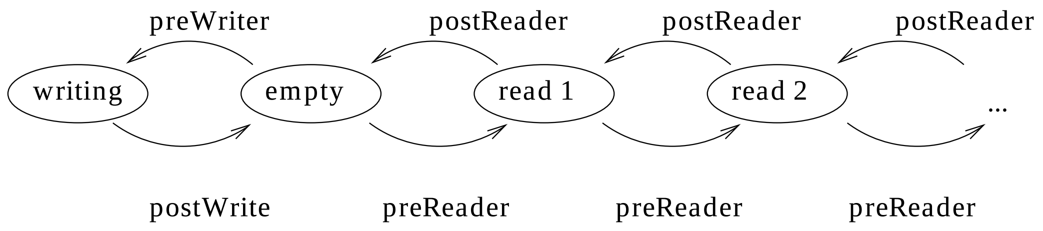

Figure 8.3: State transitions of the Reader-Writer policy

policy has at least the following states which are derived from Figure 8.3.

$$\begin{aligned} \mathit{writing} &= \mathit{readers}(0) \\ \mathit{empty} &= \mathit{writers}() \mid \mathit{readers}(0) \\ \mathit{reading}\ n &= \mathit{readers}(n) && \text{for } n > 0. \end{aligned}$$

Additional states may not be reached if the reader-writer policy is correctly used. Correctly used means that all `postWriter` and `postReader` have had their corresponding pre-service called before. If this condition holds, there cannot be a writer and a reader or several writers be active at the same time.

An example of incorrect usage would be the following sequence of calls:

```
policy = newRWPolicy()
policy.postWriter()
```

leading to a state $\mathit{writers}() \mid \mathit{writers}() \mid \mathit{readers}()$ which would allow two writers to enter the critical section.

In order to ensure correct usage of the policy we define a generic reader-writer wrapper that adds the corresponding pre- and post services around the reader and writer services of a component. As such, a client does not have access to the policy component and cannot disturb the protocol of it. The code is given in Figure 8.4.

Observe that this glue wrapper works for arbitrary provided services that are split into reader and writer services. Since arguments are packed as forms, the wrapper can specify them generically, i.e., does not put any constraint on them. Furthermore, we can apply the reader-writer adapter to arbitrary component.

More complex synchronization schemas like those based on the *service-object synchronization paradigm* (SOS) [McH94] may be implemented in a similar way. The SOS paradigm specifies the events arrival, start, and termination in the lifetime of a service invocation. Starting a service can be delayed based on the notion of guards. User defined actions are associated with the three event types. For generic synchronization policies (GSP), the programmer defines a synchronization pattern and applies it to a set of methods.

8.4 Adapting Filters

In this section we will give more complex examples of glue code for adapting filters. We reuse the push-style filter we have used in the previous chapter. While the focus in Section

```
readerWriter X:
    'policy = newRWPolicy()          # local policy object
    map                              # adapt all reader services
        form = X.reader
        each Service Arg:
            ''policy.preReader()
            Service Arg
            ''policy.postReader()
    map                              # adapt all writer services
        form = X.writer
        each Service Arg:
            ''policy.preWriter()
            Service Arg
            ''policy.postWriter()
```

Figure 8.4: Wrapping with a Reader-Writer Policy

7.2 is on wrapping the functionality of a framework as a style, the focus here is on the interaction semantics of the components. The semantics is important when combining components with the same functionality but a different control model.

The behaviour of a common class of filters is described using functions. This type of filter maps each incoming data-element to an outgoing element. We call this a transformer filter. The following service constructs a push-style filter:

```
newPushTransformer Service: asFilter          # support composition style
    apply Sink:
        Sink
        push Elem: Sink.push(Service Elem)
```

Notice that we use the wrapper `asFilter` to convert the bare component into the push-style defined in Section 7.2.

8.4.1 Integrate Host Components

We demonstrate how a host component is adapted transparently and efficiently inside Piccola. The section also demonstrates the usefulness of the partial evaluation algorithm we have defined in Chapter 6.

Assume we have a host component that offers the functionality of a sink. In Java, this is an instance of the class `java.io.BufferedWriter`. The API specifies that such instances provide the methods:

```
public void write(String str) throws IOException;
public void close();
```

Obviously, a `BufferedWriter` object can play the role of a sink in a push-style. In order to use the external object within our style, there are two mismatches that need to be solved: (i) the method name needs to be adapted and (ii) the argument passed needs to be converted into an (external) string object. As an example, the following code instantiates a new push-style sink from a given (host) file object. Refer to Section 5.8 for the technical details of instantiating the writer stream object in JPiccola.

```
newFileSink FileObject: asSink            # support composition style
    'filewriter = Host.class("java.io.FileWriter").new FileObject
    'writer = Host.class("java.io.BufferedWriter").new filewriter
    writer
    push Obj: writer.write(asString Obj)
```

The global service `asString` converts an arbitrary form into a string. If `Obj` is already a string then `asString` returns this string.

This type of glue code fixes mismatches at the level of the exchanged data. Such glue code is removed completely by the partial evaluation algorithm presented in Chapter 6. As an example, the program:

```
'output = newFileSink (Host.class("java.io.File").new "out")
''output.push 7
''output.close()
```

is specialized to:

```
'y1 = Host.class "java.io.File"
'y2 = y1.new "out"
'y3 = Host.class "java.io.FileWriter"
'y4 = y3.new y2
'y5 = Host.class "java.io.BufferedWriter"
'output = y5.new y4
'y6 = output.write(7.toString())          # push 7
'y7 = output.close()
```

The literal form `7` denotes an external number object. In JPiccola, this form has the service `toString` since it denotes a instance of the Java class `java.lang.Integer`.

8.4.2 Pull Filter

A pull-stream style is similar in functionality to the push-stream style but different in its instantiation and thread control model. In the push-style, elements are actively pushed downstream. In the pull-style, elements are actively pulled from upstream components.

The wiring-interface of a pull-stream consists of two services: `next` and `close`. We implement a pull filter by a service that maps a pull source to a filtered pull source. As in Chapter 7 we use functions to perform the wiring and double dispatch to compose pull-style filters.

As we will see, a subtle but important difference of the push-stream to the pull-stream style is that the `next` service returns a specific *air bubble* value whenever the stream is empty. In our example, the air bubble token is the empty form. In the push-stream style we are not concerned about such tokens since we assume that only valid data is pushed downstream.

The code that wraps any service into a pull-style filter is given below.

```
newPullTransformer Service: asPullFilter  # Define algebraic style
    apply PullSource:
        PullSource
        next:                             # Overwrite pulling of next element
            'e = PullSource.next()
            if (isEmpty e)                # Don't apply Service on air-bubbles
                then: e
                else: Service e
```

```
newPullDefragmenter assemble: asPullFilter
    apply PullSource:
        PullSource
        next:
            'e1 = PullSource.next()
            if (isEmpty e1)
                then: e1                        # forward air bubble token
                else:
                    'e2 = PullSource.next()
                    if (isEmpty e2)
                        then: e1                # forward e1 unprocessed
                        else: assemble e1 e2
```

Figure 8.5: Pull style defragmenter

Whereas the functionality of a transformer is equally easy written in both the push- and the pull-style, the functionality of certain other filter types is more naturally specified in a specific style.

Consider the case of a defragmenter filter that combines two or more consecutive data elements of the stream by a given `assemble` service. Using the pull style, the functionality of a binary defragmenter is given in Figure 8.5. For each pull request, we fetch two data elements, assemble them into a new element, and return it. The only inconvenience is caused by the proper treatment of air bubble tokens. We only assemble *real* data elements. If the upstream filter returns an empty form on pulling we forward it unchanged.

Using the push style, it is more clumsy to express the same functionality (see Figure 8.6). We have to introduce a `storedValue` channel to store the pushed values and a `saved` channel to decide whether there is a stored value or not. After every second `push` call we invoke the service to combine the two data-elements.

In addition to storing every other pushed element, the push defragmenter is responsible for the closing of streams. The `close` operation must be extended so that it flushes out any pending stored value and forwards them downstream.

Finally, observe that the channel `saved` also acts as a lock for the `push` and `close` operation. While a `push` or a `close` request is being handled, the channel does not contain any values. The push defragmenter maintains the following invariant: If it contains the message `saved true` then it also contains a message `storedValue F` for some form. If the message is `saved false` then the channel `storedValue` is empty.

The push-style defragmenter is more restricted than the corresponding pull-style defragmenter. For instance the pull-style defragmenter does not specify how several `next` calls interleave, whereas the push-style filter ensures full synchronization between `push` and `close` services. The pull style filter *assumes* synchronous calls — the push style filter *enforces* them. Another difference is that the pull-style filter removes air bubble tokens, whereas the push-style filter assumes that no such tokens are pushed altogether. In the following we will encounter these differences again when we formally reason about the composition of pull- and push-style filters. In fact, the reasoning will help us to detect these differences.

It should be clear that the pull style is no more convenient to use. One might ask why we do not specify everything in terms of pull styles. The situation would be equally involved

```
newPushDefragmenter Service: asFilter
    apply Sink:
        'storedValue = newChannel()
        'saved = newInitChannel false
        Sink
        push Elem:
            if saved.receive()
                then:
                    Sink.push(Service storedValue.receive() Elem)
                    saved.send false
                else:
                    storedValue.send Elem
                    saved.send true
        close:
            if saved.receive()
                then: Sink.push storedValue.receive()  # flush
            Sink.close()
            saved.send false
```

Figure 8.6: Push style defragmenter

when we consider the functionality of a fragmenter, i.e., a filter that reads data and produces two or more new data elements out of it. This filter is easily given as a push style component, and it needs to buffer additional data when used in a pull context [KBH+01].

8.4.3 Adapting Pull Filter

Assume we have a black-box pull-style filter and we want to use it in a push-style context. We now present a glue code adapter that wraps arbitrary pull-style filters to a push-style. We adapt any pull-filter as a push-filter with a coordinator agent that does the following (see Figure 8.7):

- The upstream filter pushes elements into a one-slot buffer `slot`. See Appendix E for the definition of such a buffer. A full buffer blocks when `put` is called, an empty buffer blocks if `get` is called. The slot is the input port of the pull filter.

- An autonomous coordinator agent *pumps* elements from the pull filter and pushes them upwards.

- The coordinator stops pulling elements whenever the stream is closed.

- Whenever close is called we write an air bubble token into the slot to unblock the coordinator. When the pull filter needs further data afterwards, it receives the empty forms.

Adapting a pull-filter to a push-style is possible in all languages that have some way of expressing concurrency. In the following, we analyze our adapter using the semantic model of Piccola. In fact, the adapter in Figure 8.7 contains a race-condition as we will see. We will

```
wrapPullFilterAsPush PullFilter: asFilter
    apply Sink:
        Sink
        'slot = newSlot()
        'running = newVar true
        'slot.get: if (*running)
            then: slot.get()
            else: ()                   # return () when closed
        'pullSource = PullFilter.apply(next = slot.get, close: ())
                                       # input for the pull-stream is the slot
        ''run                          # start coordinator
            do:
                loop
                    while: *running
                    do: Sink.push(pullSource.next())
                Sink.close()
        push = slot.put                # store elements in the 1-slot buffer
        close:
            ''running <- false         # signal coordinator to stop
            ''run (do: slot.put())     # unblock slot
```

Figure 8.7: Adapting Pull filter to a push style

also detect the under-specification we already mentioned: the treatment of air bubble tokens and the synchrony of push calls. The other difference is that the adapted filter has built-in concurrency and allows more parallelism.

How should we formally reason about the adapter? An obvious requirement is that it should wrap the functionality of a pull-filter to a push-filter as if we had hand-coded the push-filter. We expect that a service wrapped as a push-style transformer denotes a component with the same behaviour as the service wrapped as pull-style transformer and wrapped as a push-style filter. Formally we study the relation:

$$\texttt{newPushTransformer S} \overset{?}{\approx} \texttt{wrapPullFilterAsPush}(\texttt{newPullTransformer S}) \tag{8.9}$$

In order to compare these two expressions, we evaluate them as far as possible. When we apply the first filter to a sink we get the following specification:

```
A =                         # newPushTransformer(S).apply Sink
    Sink
    push Elem:
        'x = S Elem     # a1
        Sink.push x     # a2
```

The push stream A inherits `close` from `Sink` and provides its own `push` service definition. Calling `A.push` leads to a call of the service S followed by a call of the `Sink.push`

```
B =                  # wrapPullFilterAsPush(newPullTransformer S).apply Sink
    Sink
    'slot = newSlot()
    'running = newVar true
    run                                          # The coordinator agent:
        do:
            loop                                 # co
                while: *running
                do:                              # co1
                    'x = slot.get()
                    if (isEmpty x)
                        then: Sink.push x
                        else:
                            'x = S x             # co2
                            Sink.push x
            Sink.close()                         # co3
    push = slot.put
    close:
        ''running <- false
        # close1
        run (do: ''slot.put())
```

Figure 8.8: Comparing adapted service filters

service. When this service returns the main `push` service returns:

$$\begin{aligned}\texttt{push F} &\to a1[SF]\\ a1[F] &\to a2[\texttt{Sink.push } F]\\ a2[F] &\to F\end{aligned}$$

Note that push calls and close requests are not interleaved. It is not specified what happens when another request for `push` occurs while another `push` request is being handled.

For the adapted sink, the situation is more complex. Figure 8.8 shows the adapted filter B applied to a sink.

The variable `running` in B is either true or false. We write `running` when it is true and $\neg$`running` otherwise. We write `slot` for the empty slot, and `slot F` when it contains a non-empty form, and `slot()` when the slot contains the empty form.

The coordinator agent `co` triggers the following transitions:

$$\texttt{running} \mid co[F] \to \texttt{running} \mid co1 \tag{8.10}$$

$$\neg\texttt{running} \mid co[F] \to \neg\texttt{running} \mid co3[\texttt{Sink.close}()] \tag{8.11}$$

$$\texttt{slot}\ () \mid co1 \to \texttt{slot} \mid co[\texttt{Sink.push}()] \tag{8.12}$$

$$\texttt{slot F} \mid co1 \to \texttt{slot} \mid co2[SF] \tag{8.13}$$

$$co2[F] \to co[\texttt{Sink.push F}] \tag{8.14}$$

The first two transitions are derived from the `while` loop. We write $co[F]$ since the coordinator loop is also triggered when the external function `Sink.push` returns. If `running` is true the then-block is executed; otherwise the while block terminates and the coordinator closes the sink (Transition 8.11). The remaining transitions represent the while block. Observe that we have different transitions depending whether the value in the `slot` is the empty form or not. If the slot contains the empty form, i.e., an air-bubble, we invoke `Sink.push` (Transition 8.6). Otherwise we invoke the external service S with the value in the slot and then invoke the `Sink.push` with the result of the previous invocation.

This sequence of execution is interleaved with external requests to `push` and `close` as follows:

$$\texttt{slot} \mid \texttt{push}\ F \rightarrow \texttt{slot F} \mid () \tag{8.15}$$

$$\neg\texttt{running} \mid \texttt{close} \rightarrow \mathit{close1} \mid \neg\texttt{running} \mid () \tag{8.16}$$

$$\texttt{running} \mid \texttt{close} \rightarrow \mathit{close1} \mid \neg\texttt{running} \mid () \tag{8.17}$$

$$\texttt{slot} \mid \mathit{close1} \rightarrow \texttt{slot}() \tag{8.18}$$

A call to `push` immediately returns when the slot is empty, otherwise it is blocked (Transition 8.15). A call to `close` sets the `running` variable to false, no matter what its current value is. Invoking `close` immediately returns, but in addition spawns an asynchronous agent *close1*. This agent is specified by Transition 8.18. It stores the empty form into the slot in order to unblock the coordinator.

These are the observable differences between the behaviour of stream A and B.

- The push-filter A returns when the external service S and the downstream push operation have returned. Contrary, the `push` service of filter B immediately returns whenever the internal `slot` is empty. We can observe this difference by taking for S a service that blocks on an external `tick` call. Consider:

  ```
  channel = newChannel()
  myService X:
      ''channel.receive()     # block on the channel
      X
  tick = channel.send         # Used to proceed with myService

  B = wrapPullFilterAsPush(newPullTransformer myService) >> Stdout
  B.push "Hello"              # push returns, coordinator blocked
  tick()                      # unblock coordinator, prints output
  ```

  Executing this code finally prints out "`Hello`". The reason is that the coordinator and not the main agent is blocked while executing the service `myService`.

  In contrast, the same script blocks, when we use the single-threaded push transformer:

  ```
  A = (newPushTransformer myService) >> Stdout
  A.push "Hello"              # blocks, since no tick is available
  ```

  This scenario uncovers the fact that the adapted pull-filter introduces additional concurrency.

- The wrapped pull filter B contains a race condition. Assume that the slot is not empty, i.e., it contains an unprocessed data-element when the stream gets closed, i.e., the variable `running` is set to false. Depending on the internal state, the coordinator may decide not to proceed anymore since `running` is false letting the final element unprocessed in the slot.

The following trace uncovers the race condition:

$$\begin{array}{ll} & \texttt{running} \mid \texttt{slot} \mid co2[SF_1] \mid \texttt{push}\ F_2 \mid \texttt{close} \\ \xrightarrow{\texttt{push}} & \texttt{running} \mid \texttt{slot}\ F_2 \mid co2[SF_1] \mid \texttt{close} \\ \xrightarrow{\texttt{close}} & \neg\texttt{running} \mid close1 \mid \texttt{slot}\ F_2 \mid co2[SF_1] \\ \xrightarrow{SF_2} & \neg\texttt{running} \mid close1 \mid \texttt{slot}\ F_2 \mid co[\texttt{Sink.push}\ F_3] \\ \xrightarrow{co} & \neg\texttt{running} \mid close1 \mid \texttt{slot}\ F_2 \mid co3[\texttt{Sink.close}()] \end{array}$$

We have written the action that reduces over the arrow. Observe that the `slot` still contains the value F_2 that was the most recently pushed.

- Additionally, air bubble tokens are handled differently: Filter B does not apply the service S on empty forms that are pushed. In contrast, A applies S to all pushed forms.

The under-specification of the dynamics of our stream leads to different behaviour A race-condition is caused by the wrong assumption that if `push` returns, we can call `close` safely. However, with the introduced concurrency and the added buffer, this assumption does no longer hold. An other problem is the (so far undocumented) assumption that no air-bubbles are pushed downstream a push stream.

We now proceed as follows: We formally specify the assumptions of a push-style filter and then present a correct version of the adapter. The main difficulty for the adapter is to inform the coordinator when the stream gets closed allowing the coordinator to remove any pending data-elements from the buffer. For that purpose we define a closeable slot.

8.4.4 Enforce Contextual Dependencies

In order to more rigorously specify the behaviour of a push-style we encode the assumptions by a glue wrapper. The assumptions are:

- `Push` and `close` calls are mutually exclusive. There is only one `close` call at end. After the `close` call, no more `push` calls are attempted.
- No empty forms (as air-bubbles) are pushed.

We specify a push-style filter that synchronizes access to the downstream `push` and `close` operations and maintains a lock that it removes when the stream is closed. This prevents duplicated calls to `close` and calls to `push` after the stream has been closed. The specification of the filter is given in Figure 8.9. While the stream is not closed, the empty form is written along the channel `open`. When the stream is closed this lock is removed. Further calls to `close` and `push` block in the service `open.read`.

The other assumption that no empty forms are pushed to a push style filter is enforced by the `noAirBubble` filter on Figure 8.10.

We codify the assumption of a push-style filter as:

```
assume Filter: safe >> noAirBubble >> Filter >> safe >> noAirBubble
```

```
safe: asFilter
    apply Sink:
        'open = newInitReadChannel()
        Sink
        synchronized
            push E:
                ''open.read()           # check lock
                Sink.push E
            close:
                ''open.receive()        # remove lock
                Sink.close()
```

Figure 8.9: Enforce Synchronized Access to Filters

```
noAirBubble: asFilter
    apply Sink:
        Sink
        push E: if (isEmpty E)
            else: Sink.push E
```

Figure 8.10: Remove Air Bubbles

Observe that we also apply the `safe` filter downstream. This prevents the wrapped filter from pushing data-elements concurrently.

The function `assume` wraps any push-style filter such that the assumptions are guaranteed: calls to `push` and `close` are synchronized and no air bubble tokens can be pushed in nor will not be pushed out of the stream.

8.4.5 A Closeable Slot

The main difficulty in the adaption of the pull filter to a push filter is that we must interrupt the coordinator when it is blocked on the empty slot. In general, we cannot stop an agent that is blocked while receiving from a channel. We have to know if there is a blocked agent on the slot.

Instead of informing the coordinator directly, we use a closeable slot. This one-slot buffer has a service `close`. If it is closed and we invoke `get`, it always returns the same dummy element. As we will show, the the adapter guarantees that the slot can be closed only if it is empty.

A closeable slot has three states: *full X*, *empty* and *closed*. When the closeable slot is closed, further calls to `close` and `put` will not be possible. The high-level transitions of the slot must be:

$$\begin{array}{ll} \textit{empty} \mid \texttt{put X} \rightarrow \textit{full } X \mid () & \textit{full } X \mid \texttt{get}() \rightarrow \textit{empty} \mid (\texttt{true}, \texttt{value} = X) \\ \textit{empty} \mid \texttt{close}() \rightarrow \textit{closed} \mid () & \textit{closed} \mid \texttt{get}() \rightarrow \textit{closed} \mid (\texttt{false}, \texttt{value} = ()) \end{array}$$

The service `get` returns values as follows. If the slot is not closed, `get` returns the form

```
newCloseableSlot:
    'value = newChannel()
    'isEmpty = newInitChannel()             # initially empty
    get:
        'open = val.receive()
        ''if open
            then: isEmpty.send()            # enable close and put
            else: val.send open             # slot is closed
        open
    put X:
        ''isEmpty.receive()
        ''val.send(true, value = X)
    close:
        ''isEmpty.receive()
        ''val.send(false, X = ())
```

Figure 8.11: Closeable Slot

`true` extended with a binding `value = X` where `X` is the form previously put into the slot. If the slot is closed, we return `false` extended with `value = ()`. Since booleans are encoded as forms, the expression `true, value = X` is considered a boolean form with a value.

The implementation of the `closeable` slot is given in Figure 8.11. It ensures the following invariant: When the slot is full, we have the message `value(true, value = X)` where `X` is the value in the slot. If the slot is closed, we have the message `value(false, value = ())`. Finally, when the slot is empty, we have the message `isEmpty()`.

8.4.6 A Synchronized Adapter for Pull Filters

We use the closeable slot as a synchronizer for an improved version of the adapter. We call this a synchronized adapter as it maintains the synchrony of push calls with interleaving pull-filter processing. A push call does not return unless the downstream filter has returned or the adapted pull filter decides that it needs more element. The synchronized adapter is given in Figure 8.12.

The important synchronization channel is `pushLock`. This channel blocks the client of a `push` call from returning until the coordinator is waiting for the next data-element or the coordinator loop has terminated. Observe that we overwrite the `slot.get` service to signal on the `pushLock` channel.

Termination of the stream is handled as follows: Whenever `close` is called, the `running` flag is set to false and the slot it closed. This means that the coordinator can now grab values from the slot and receives the empty form. At one point the pull source will return an element. If this element is not an empty form, i.e., an air bubble, it is pushed downstream. Now the coordinator agent sees that the `running` flag is set to false. It exits the while loop and closes the push stream. Blocked `push` clients are released by signaling on the channel `pushLock` and the `done` channel signals termination. The `close` requests returns when the coordinator has finished its work.

```
wrapPullFilterAsSyncPush PullFilter: asFilter
    apply Sink:
        Sink
        'slot = newCloseableSlot()
        'pushLock = newReadChannel()
        'done =  newChannel()
        'running = newVar true
        'slot.get:
            ''pushLock.send()     # coordinator expects a value
            slot.get().value
        'pullSource = PullFilter.apply(next = slot.get, close: ())
        run
            do:
                loop
                    while: *running
                    do:
                        'e = pullSource.next()
                        if (isEmpty e)
                            else: Sink.push e
                Sink.close()
                done.send()
        push E:
            pushLock.receive()
            slot.put E
            pushLock.read()       # synchronize with coordinator
        close:
            ''if (*running)
                then:
                    ''running <- false
                    slot.close()
                    done.receive()
```

Figure 8.12: Synchronized pull stream adapter

```
Sink =          # assume(newPushDefragmenter assemble).apply(Sink)
    open = newInitReadChannel()
    'val = newChannel()
    'saved = newChannel()
    ''saved.send false
    Sink
    push Elem:
        ''open.read()
        if (isEmpty Elem)
            else:
                if (saved.receive())
                    then:
                        'result = assemble val.receive() Elem
                        if (isEmpty result)
                            else: Sink.push result
                        saved.send false
                    else:
                        val.send Elem
                        saved.send true
    close:
        open.receive()
        if (saved.receive())
            then: Sink.push(val.receive())
        Sink.close()
        saved.send false
```

Figure 8.13: Applying the hand-coded Push Defragmenter

Using the wrapper `assume` to enforce correct usage of the filter, for any service `S` it holds and for any de-fragmentation service `assemble`:

$$\texttt{assume(newPushTransformer S)} \approx$$
$$\texttt{assume(wrapPullFilterAsSyncPush(newPullDefragmenter S))}$$
$$\texttt{assume(newPushDefragmenter assemble)} \approx$$
$$\texttt{assume(wrapPullFilterAsSyncPush(newPullDefragementer assemble))}$$

In order to convince ourselves that these equations are valid we can, as before, apply both filters to a sink and simplify the resulting expressions by applying the beta-reduction. The corresponding simplified expression for the handwritten defragmenter is in Figure 8.13, the adapted pull filter in Figure 8.14.

When a `push` request arrives on an initialized adapted filter the following sequence of transitions occurs. To simplify reading, we omit the state of the `running` variable which is always true.

```
Sink = # assume(wrapPullFilterAsSyncPush(newPullDefragmenter assemble)).apply(Sink)
    'lock = newInitChannel()
    'slot = newCloseableSlot()
    'pushLock = newReadChannel()
    'done =  newChannel()
    'running = newVar true
    'slot.get:
        ''pushLock.send()
        slot.get().value
    run
        do:
            loop                                # co
                while: *running
                do:
                    'e1 = slot.get()            # co1
                    if (isEmpty e1)
                        else:
                            'e2 = slot.get()    # co2
                            if (isEmpty e2)
                                then: Sink.push e1
                                else:
                                    'r = assemble e1 e2
                                    if (isEmpty r) (else: Sink.push r)
            Sink.close()
            done.send()
    Sink
    push E:
        lock.receive()                          # push1
        pushLock.receive()                      # initialize
        if (isEmpty E) (else: slot.put E)       # push2
        pushLock.read()
        lock.send()
    close:
        lock.receive()
        ''if *running
            then:
                ''running <- false              # close1
                slot.close()                    # close2
                done.receive()
```

Figure 8.14: Applying the adapted Pull Defragmenter

$$
\begin{aligned}
&\texttt{push } X \mid \texttt{lock}() \mid \mathit{Co} \mid \texttt{slot.empty} \\
&\xrightarrow{\texttt{push}} \quad \texttt{push1 } X \mid \mathit{Co} \mid \texttt{slot.empty} && (8.19)\\
&\xrightarrow{\mathit{Co}} \quad \texttt{push1 } X \mid \mathit{Co1} \mid \texttt{pushLock}() \mid \texttt{slot.empty} && (8.20)\\
&\xrightarrow{\mathit{push1}} \quad \texttt{push2} \mid \mathit{Co1} \mid \texttt{slot.full } X \\
&\xrightarrow{\mathit{Co1}} \quad \texttt{push2} \mid \mathit{Co2}(X) \mid \texttt{slot.empty} \\
&\xrightarrow{\mathit{push2}} \quad \texttt{lock}() \mid \mathit{Co2}(X) \mid \texttt{slot.empty}
\end{aligned}
$$

At this point the invocation push returns. Notice that the coordinator agent is in state *Co2* and it has read the pushed form X. We assume that X is not the empty form. The only nondeterminism that can occur here is that the first reductions 8.19 and 8.20 may occur in different order. However, this would not change the overall behaviour. While serving this push request, additional push and close requests are blocked since the lock is not free.

We now consider what happens if at this point a close request occurs. We expect that the form X is flushed and forwarded to the downstream sink and that close only returns after the downstream sink returned.

$$
\begin{aligned}
&\texttt{close} \mid \texttt{lock}() \mid \mathit{Co2}(X) \mid \texttt{running} \mid \texttt{slot.empty} \\
&\xrightarrow{\texttt{close}} \quad \mathit{close1} \mid \mathit{Co2}(X) \mid \neg\texttt{running} \mid \texttt{slot.empty} \\
&\xrightarrow{\mathit{close1}} \quad \mathit{close2} \mid \mathit{Co2}(X) \mid \neg\texttt{running} \mid \texttt{slot.closed} \\
&\xrightarrow{\mathit{Co2}} \quad \mathit{close2} \mid \mathit{Co}[\texttt{Sink.push } X] \mid \neg\texttt{running} \mid \texttt{slot.closed} \\
&\xrightarrow{\mathit{Sink.push}} \quad \mathit{close2} \mid \texttt{Sink.close} \mid \neg\texttt{running} \mid \texttt{slot.closed} \\
&\xrightarrow{\mathit{Sink.close}} \quad \mathit{close2} \mid \texttt{done}() \mid \neg\texttt{running} \mid \texttt{slot.closed} \\
&\xrightarrow{\texttt{close2}} \quad \neg\texttt{running} \mid \texttt{slot.closed}
\end{aligned}
$$

As expected, the coordinator invokes Sink.push and then Sink.close. When close finally returns, the coordinator loop has finished. In addition, no more push and close requests are handled since the lock is not free anymore.

The remaining transitions are similar. For instance, when the coordinator is in state $\mathit{Co2}(X)$ and an additional push request is handled, the coordinator will invoke assemble and then forward the result to the downstream sink. This however, is only the case, if the pushed values are not air-bubbles, i.e., different from the empty form. Thus, under the assumptions of correct usage, the wrapped pull filter behaves like the hand written push filter.

8.5 Discussion

We have presented how to reason about Piccola programs at the language level. As non-trivial examples we demonstrated how generic glue code wrappers overcome compositional mismatches and how these wrappers formalize the contract between a component and its environment.

We have studied two extended examples. First, we considered synchronization wrappers that express the synchronization constraints assumed by a component. We use the synchronization wrappers to make components safe in a multi-threaded environment. The wrappers separate the functionality of the component from their synchronization aspects. If the constraints assumed by the component hold in a particular composition the wrapper is not needed. In particular the wrapper is not necessary when the component is already wrapped. This property is formally expressed by imposing that the wrappers are idempotent.

The second study compares push- and pull-flow filters. We demonstrate how to adapt pull-filters so that they work in a push-style. We have constructed a generic adapter for this task in two iterations. The first version contains a race-condition that may lead to data being lost. The formal model of Piccola is used to analyze the traces of an adapted filter and to detect this bug. To fix the problem, we specify the dynamics of a push-style filter, namely that push and close calls are mutually exclusive, that no further push calls are attempted after close, and that no air-bubble elements are pushed downstream. Having clarified the interaction protocol in a wrapper, we present an improved version of the generic adapter. We show that the adapter ensures these invariants.

We propose the use of wrappers to formalize and codify the contracts between components and the style. Such contracts supplement a composition style. This allows us to develop the style independently of the components for it. The composition abstractions ensure that the properties of the style hold invariantly and the components establish these invariants. However, more work is needed to fertilize this idea. For instance, we would like to have assistant tools for the actual model checking. A possibility is to automatically extract the state transitions of a Piccola program and use model checking tools such as the labeled transition system analyzer of Magee and Kramer [MK99] for this task.

Chapter 9

Conclusion and Future Work

This chapter sums up the contributions made in the thesis, points to future work and provides some final conclusions.

9.1 Validation

In this thesis we claim that

> *Extensible composition abstractions can be defined and implemented on a foundation of* forms, agents *and* channels. *A set of plug-compatible components is captured by a* composition style*.*

We have defined the properties of a form and introduced a calculus with explicit namespaces that builds on forms, agents and channels. Agent-expressions and form-expressions are unified in the sense that an agent evolves to a form. A form is a fully reduced agent. While evolving, an agent may communicate with other agents by exchanging forms along channels. We use the term *form* for the unification of services, extensible records and environments. In classical record calculi, a distinction is made between record labels and variables of the calculus. This distinction is not present in Piccola. In Chapter 5 we present Piccola as a composition language on top of the calculus. While the calculus gives us expressiveness and a sound semantics, the Piccola language eases the process of programming by adding user-defined operators, explicit dynamic namespaces and recursive forms.

Piccola is a pure composition language because all computation eventually happens in external components and composition only is performed in Piccola. For that purpose, it is important that external components can be integrated seamlessly. We use reflection on the host language to access any host component as a form from Piccola. We then adapt the interface of a host component inside Piccola. Due to the extensibility of forms, this adaption is transparent to services defined in Piccola and external services. The adapted interface maintains the original identity of the component. When we invoke a host service with the adapted component as an argument, the host service will receive the original component.

We implement a number of composition abstractions as first-class citizen of Piccola. We show that the unifying concept of forms together with the runtime model of agents and channels is expressive enough to encode generic wrappers, connectors, control and coordination abstractions, context dependent policies, and richer object-models that support inheritance as first-class citizens of Piccola. Implementing such abstractions as first-class citizens is a necessary feature for a composition language to make the architecture explicit in the code.

We use the simplicity of the Piccola formalism at several places. We show that the calculus can safely be embedded into the more general framework given by the π-calculus. The exercise of embedding Piccola into the π-calculus is useful for two purposes. First, it allows us to use reasoning techniques from the π-calculus. Second, working on the encoding helped us to get a better understanding of what the minimal kernel for a composition calculus should be. In fact, the second reason turned out to be more important than the first. We have discovered important simplifications on the Piccola calculus like the fact that assignment can be expressed inside Piccola and does not need to be treated as primitive.

Another usage of the simplicity of the Piccola calculus is given in the partial evaluation algorithm in Chapter 6. The algorithm separates any Piccola expression into a referentially transparent and a side-effectful part. The fact that forms are immutable values is the key aspect that enables specialization. We can inline projections at specialization time. The algorithm is used to speed up a Piccola interpreter and it helps a composition environment to provide information about the value of identifiers at composition time.

Finally, we present how Piccola can be used to reason about composition abstractions. We infer the states and transitions of a program from the code and the semantics of agents and asynchronous channels. As an example, we show how this formalism helps us to detect composition mismatches and how we can use wrappers to formalize the contracts between components and composition styles.

In summary, we have given evidence that Piccola meets our requirements for a composition language of Chapter 2. It is expressive enough to implement composition abstractions as first-class entities, it supports reasoning due to the high-level semantics of forms, agents and channels, and it supports accessing external components.

9.2 Future Work

We kept the syntax of the Piccola language minimal. As a result, Piccola code sometimes has more the flavour of a calculus than a high-level programming language. Reading Piccola code can be hard since the code often is very dense. This is partially due to the minimal syntax and partially due to the lack of a proper composition environment. In the following we discuss some enhancements to alleviate the situation.

The Piccola system is still command line based. Debugging is tedious, although the Squeak version [SA01] of Piccola has some nice debugging features to step through code, to halt groups of agents, and to explore the history of forms, i.e., when the form was accessed and how it was built. We would like to extend this system and make it platform independent by implementing a composition environment inside Piccola.

Programming Piccola, we start with form expressions that consist of only a few bindings. As we add more bindings, at some point we need a fixed point since the dependencies between the services become too complex to handle. This problem could be alleviated if the form we are currently defining was available as an explicit form much like the current root context. Forms would have a notion of self leading towards an object-based language. However, to add such a notion, it is necessary to give the semantics of fixed points directly on the calculus, instead of defining a translation.

Further work is needed to give a fully abstract encoding of Piccola into $L\pi$. The foundations of the π-calculus we choose seems appropriate for most usages. However, there is a lot of research going on in studying calculi for secure and distributed system (e.g., [CG98, VC99]) that we have not considered in the current work. Such calculi allow the user to reason about important properties like safety for open and distributed systems.

We have not considered types in our initial design for pragmatic reasons. We did not want to be constrained by types. However, the author sometimes wished a type system would have detected type errors earlier. Due to the explicit representation of namespaces we can type open expressions in Piccola. In a component environment where new components are added at runtime, we have to give up the idea of a closed system that can be type-checked at once. We are working on a notion of types that separates provided and required types. Both these types can be inferred from a script. A conformance relation has to assert that a provided type conforms to a required type of another component.

The Piccola system we have implemented is a prototype. We are now at a stage where we can start to work on an efficient implementation. The partial evaluation algorithm is a first step in that direction. In a project that is going on we try to generate byte-code for the Java and Squeak virtual machines directly from form expressions. It is not obvious at this point which services we can map to classes and which services to methods. For instance, the service `newVar` that creates a new Piccola variable could directly be mapped to a constructor for a single slot class. Other services like `println` naturally map to a method of an appropriate object. It is not clear where the border lies.

We have presented a language that allows the programmer to implement many design patterns and aspects as first-class citizen. More work is needed to give methodological help how to design a composition styles, i.e., how to find their core abstractions. The examples we have demonstrated are obviously tiny problems and come from well understood domains. It is challenging to apply these ideas to bigger systems.

It is also instructive to use Piccola for giving semantics to the different extensions for object-oriented languages we mentioned in Chapter 2. This would not only improve our understanding of the differences between these approaches, but it would also enable the composition of components of the different paradigms, e.g., combine aspects with composition filters and context relations within a well defined semantical framework.

It would also be interesting to attach tools for the π-calculus and model checking systems to Piccola. Garlan *et al.* [GMW00] introduce ACME as a generic interchange format for architectural description languages. It should be possible to map styles to this language and use other formalisms to infer properties of the resulting system.

9.3 Concluding Remarks

We have presented Piccola, a composition language based on the unifying principle of forms, agents and channels. The language supports the specification and implementation of composition abstractions as first-class values. Using the language we propose to make frameworks available as domain-specific languages. These little languages are packed as a composition styles in terms of components, connectors and connecting rules.

Ever since software engineers have built systems, they have tried to make them more robust and flexible. The basic principles to achieve these goals are separation of concern, encapsulation and decomposition. The object-oriented approach, now widely applied and accepted in industry with good tool and methodological support, is the most recent incarnation of these principles. Still, the approach has not achieved the goal of making objects as composable as we would like [Ude94]. While it is probable that Piccola will not be the final answer either, we are convinced that the notion of forms, agents and channels shows us the direction to make systems more robust and flexible by defining high-level, declarative and extensible composition abstractions.

Appendix A

The Piccola calculus

This appendix summarizes syntax and semantics of the Piccola calculus of Chapter 3. Agents A, B and forms F, G, H are defines as follows:

$$
\begin{array}{rclll|lll}
A, B, C & ::= & \epsilon & \textit{empty form} & & \mathbf{R} & \textit{current root} \\
& | & A; B & \textit{sandbox} & & x & \textit{variable} \\
& | & x{\mapsto} & \textit{bind} & & \mathit{hide}_x & \textit{hide} \\
& | & \mathbf{L} & \textit{inspect} & & A \cdot B & \textit{extension} \\
& | & \lambda x.A & \textit{abstraction} & & A B & \textit{application} \\
& | & \nu c.A & \textit{restriction} & & A \mid B & \textit{parallel} \\
& | & c? & \textit{input} & & c & \textit{output}
\end{array}
$$

$$
\begin{array}{rclll|lll}
F, G, H & ::= & \epsilon & \textit{empty form} & & S & \textit{service} \\
& | & x{\mapsto}F & \textit{binding} & & F \cdot G & \textit{extension}
\end{array}
$$

$$
\begin{array}{rclll|lll}
S & ::= & F; \lambda x.A & \textit{closure} & & \mathbf{L} & \textit{inspect} \\
& | & x{\mapsto} & \textit{bind} & & \mathit{hide}_x & \textit{hide} \\
& | & c & \textit{output} & & &
\end{array}
$$

The following rules infer that agent A reduces to B:

$$(F; \lambda x.A)\ G \rightarrow F \cdot x{\mapsto}G; A \qquad (\textit{reduce beta}) \qquad\qquad cF \mid c? \rightarrow F \qquad (\textit{reduce comm})$$

$$F \cdot x{\mapsto}G; x \rightarrow G \qquad (\textit{reduce project})$$

$$\mathbf{L}\epsilon \rightarrow \epsilon; \lambda x.(x; \mathit{isEmpty})\epsilon \qquad (\textit{reduce inspect empty})$$

$$\mathbf{L}S \rightarrow \epsilon; \lambda x.(x; \mathit{isService})\epsilon \qquad (\textit{reduce inspect service})$$

$$\mathbf{L}(F \cdot x{\mapsto}G) \rightarrow \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_x \qquad (\textit{reduce inspect label})$$

$$\frac{A \equiv A' \quad A' \rightarrow B' \quad B' \equiv B}{A \rightarrow B} \quad (\textit{reduce struct}) \qquad\qquad \frac{A \rightarrow B}{\mathcal{E}[A] \rightarrow \mathcal{E}[B]} \quad (\textit{reduce propagate})$$

where label_x is the form $\mathit{project}{\mapsto}(\epsilon; \lambda x.(x; x)) \cdot \mathit{hide}{\mapsto}\mathit{hide}_x \cdot \mathit{bind}{\mapsto}(x{\mapsto})$ and $\mathcal{E}$ is an evaluation defined by the grammar:

$$\mathcal{E} ::= [] \mid \mathcal{E} \cdot A \mid F \cdot \mathcal{E} \mid \mathcal{E}; A \mid F; \mathcal{E} \mid \mathcal{E}A \mid F\mathcal{E} \mid A|\mathcal{E} \mid \mathcal{E}|A \mid \nu c.\mathcal{E}$$

The operators have the following precedence:

$$\textit{application} > \textit{extension} > \textit{restriction}, \textit{abstraction} > \textit{sandbox} > \textit{parallel}$$

The congruence $\equiv$ is the smallest congruence satisfying the following axioms:

$$\begin{array}{rlrclr}
 & & F;A\cdot B & \equiv & (F;A)\cdot(F;B) & \text{(sandbox ext)}\\
 & & F;AB & \equiv & (F;A)(F;B) & \text{(sandbox app)}\\
 & & A;(B;C) & \equiv & (A;B);C & \text{(sandbox assoc)}\\
 & & F;G & \equiv & G & \text{(sandbox value)}\\
 & & F;\mathbf{R} & \equiv & F & \text{(sandbox root)}\\
 & & F\cdot\epsilon & \equiv & F & \text{(ext empty right)}\\
 & & \epsilon\cdot F & \equiv & F & \text{(ext empty left)}\\
 & & (F\cdot G)\cdot H & \equiv & F\cdot(G\cdot H) & \text{(ext assoc)}\\
 & & S\cdot(x\mapsto F) & \equiv & (x\mapsto F)\cdot S & \text{(ext service commute)}\\
 & & x\mapsto F\cdot x\mapsto G & \equiv & x\mapsto G & \text{(single binding)}\\
x\neq y & \text{implies} & x\mapsto F\cdot y\mapsto G & \equiv & y\mapsto G\cdot x\mapsto F & \text{(ext bind commute)}\\
 & & (F\cdot S)G & \equiv & SG & \text{(use service)}\\
 & & S\cdot S' & \equiv & S' & \text{(single service)}\\
 & & hide_x(F\cdot x\mapsto G) & \equiv & hide_x F & \text{(hide select)}\\
x\neq y & \text{implies} & hide_y(F\cdot x\mapsto G) & \equiv & hide_y F\cdot x\mapsto G & \text{(hide over)}\\
 & & hide_x\epsilon & \equiv & \epsilon & \text{(hide empty)}\\
 & & hide_x S & \equiv & S & \text{(hide service)}\\
 & & (A\mid B)\mid C & \equiv & A\mid(B\mid C) & \text{(par assoc)}\\
 & & (A\mid B)\mid C & \equiv & (B\mid A)\mid C & \text{(par left commute)}\\
 & & (A\mid B)\cdot C & \equiv & A\mid B\cdot C & \text{(par ext left)}\\
 & & F\cdot(A\mid B) & \equiv & A\mid F\cdot B & \text{(par ext right)}\\
 & & (A\mid B)C & \equiv & A\mid BC & \text{(par app left)}\\
 & & F(A\mid B) & \equiv & A\mid FB & \text{(par app right)}\\
 & & (A\mid B);C & \equiv & A\mid B;C & \text{(par sandbox left)}\\
 & & F;(A\mid B) & \equiv & F;A\mid F;B & \text{(par sandbox right)}\\
 & & F\mid A & \equiv & A & \text{(discard zombie)}\\
 & & \nu cd.A & \equiv & \nu dc.A & \text{(commute channels)}\\
c\notin fc(A) & \text{implies} & A\mid\nu c.B & \equiv & \nu c.(A\mid B) & \text{(scope par left)}\\
c\notin fc(A) & \text{implies} & (\nu c.B)\mid A & \equiv & \nu c.(B\mid A) & \text{(scope par right)}\\
c\notin fc(A) & \text{implies} & (\nu c.B)\cdot A & \equiv & \nu c.(B\cdot A) & \text{(scope ext left)}\\
c\notin fc(A) & \text{implies} & A\cdot\nu c.B & \equiv & \nu c.(A\cdot B) & \text{(scope ext right)}\\
c\notin fc(A) & \text{implies} & A;\nu c.B & \equiv & \nu c.(A;B) & \text{(scope sandbox left)}\\
c\notin fc(A) & \text{implies} & (\nu c.B);A & \equiv & \nu c.(B;A) & \text{(scope sandbox right)}\\
c\notin fc(A) & \text{implies} & (\nu c.B)A & \equiv & \nu c.BA & \text{(scope app left)}\\
c\notin fc(A) & \text{implies} & A(\nu c.B) & \equiv & \nu c.AB & \text{(scope app right)}\\
 & & cF & \equiv & cF\mid\epsilon & \text{(emit)}
\end{array}$$

Appendix B

Proofs for Chapter 3

B.1 Proof of Proposition 3.13

In order to prove Proposition 3.13 we first show the following Lemma. Lemma 3.14 states the same as the proposition but only for closed agents, i.e. for agents of the form $F; A$. Note that all closed agents A are equivalent to $\epsilon; A$ which is proved by induction over closed agents A.

Lemma B.1 *Each agent $F; A$ is congruent to a canonical agent:*

$$F; A \equiv \nu c_1...c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k)$$

for messages M_i and threads A_j for $0 \leq i \leq m, 0 \leq j < k$ and A_k is a thread or a form for $k \geq 1$.

Proof. We prove this by induction over A.

- $A = G$ for a form G. The main agent is $F; G \equiv G$ which is a form.
- $A = \mathbf{R}$. The main agent is $F; \mathbf{R} \equiv F$ which is a form.
- $A = x$. The main agent is $F; x$ which is case 3 of Definition 3.11.
- $A = B; C$. We have $F; (B; C) \equiv (F; B); C$. Using the induction hypothesis we have:

$$\begin{aligned} F; B &\equiv \nu c_1...c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k) \\ (F; B); C &\equiv \nu c_1...c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k); C \\ &\equiv \nu c_1...c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_{k-1} \mid A_k; C) \end{aligned} \quad \text{(B.1)}$$

  Now A_k is either a form or a thread.

  - If A_k is a thread $\widehat{\mathcal{E}}[A']$ for some $\widehat{\mathcal{E}}$ and A', then $\widehat{\mathcal{E}}; C$ is the required thread context.
  - Otherwise A_k is a form G. By the induction hypothesis — note that the induction is over $A \equiv B; C$ — we have:

$$G; C \equiv \nu d_1...d_{n'}.(M'_1 \mid ... \mid M'_{m'} \mid A'_1 \mid ... \mid A'_{k'}) \quad \text{(B.2)}$$

Combining (B.1) and (B.2) we have:

$$\begin{aligned} F;B;C \equiv \nu c_1 \ldots c_n d_1 \ldots d_{n'}.(&M_1 \mid \ldots \mid M_m \mid M'_1 \mid \ldots \mid M'_{m'} \\ &\mid A_1 \mid \ldots \mid A_{k-1} \mid A'_1 \mid \ldots \mid A'_{k'}) \end{aligned}$$

which is of the required form.

- $A = B \cdot C$. By rule *sandbox ext* we have $F;(B \cdot C) \equiv (F;B) \cdot (F;C)$. We apply the induction hypothesis on both parts:

  $$\begin{aligned} F;B &\equiv \nu c_1 \ldots c_n.(M_1 \mid \ldots \mid M_m \mid A_1 \mid \ldots \mid A_k) \\ F;C &\equiv \nu d_1 \ldots d_{n'}.(M'_1 \mid \ldots \mid M'_{m'} \mid A'_1 \mid \ldots \mid A'_{k'}) \end{aligned}$$

  As in the previous case, A_k is either a form or can be written as a thread context. Both cases can be calculated similar to the previous case and by using the fact that $F \cdot G$ is congruent to a form.

- $A = \lambda x.B$. The main agent is $F;\lambda x.B$ which is a closure.

- $A = BC$. We have $F;(BC) \equiv (F;B)(F;C)$ using rule *sandbox app* and the induction hypothesis:

  $$\begin{aligned} F;B &\equiv \nu c_1 \ldots c_n.(M_1 \mid \ldots \mid M_m \mid A_1 \mid \ldots \mid A_k) \\ F;C &\equiv \nu d_1 \ldots d_{n'}.(M'_1 \mid \ldots \mid M'_{m'} \mid A'_1 \mid \ldots \mid A'_{k'}) \end{aligned}$$

  Now, A_k is either a form or a can be written as a thread context:

  - Either A_k is an agent $\widehat{\mathcal{E}}[A']$ for some $\widehat{\mathcal{E}}$ and A' of the required form. With $\widehat{\mathcal{E}}' = \widehat{\mathcal{E}}(M'_1 \mid \ldots \mid M'_{m'} \mid A'_1 \mid \ldots \mid A'_{k'})$

    $$\nu c_1 \ldots c_n d_1 \ldots d_{n'}.(M_1 \mid \ldots \mid M_m \mid \widehat{\mathcal{E}}'[A'])$$

    is the required canonical agent.
  - Otherwise A_k is a form G. Combining the two hypothesis equations yields:

    $$\begin{aligned} F;BC \equiv\ & \nu c_1 \ldots c_n d_1 \ldots d_{n'}.(_1 \mid \ldots \mid (_M m \mid A_1 \mid \ldots \mid A_{k-1} \\ &\quad \mid G(M'_1 \mid \ldots \mid M'_{m'} \mid A'_1 \mid \ldots \mid A'_{k'})) \\ \equiv\ & \nu c_1 \ldots d_{n'}.(M_1 \mid \ldots \mid M_m \mid M'_1 \mid \ldots \mid M'_{m'} && \text{(by rule } \textit{par app right}\text{)} \\ &\quad \mid A_1 \mid \ldots \mid A_{k-1} \mid A'_1 \mid \ldots \mid A'_{k'-1} \mid GA'_{k'}) \end{aligned}$$

    If $A'_{k'}$ is an agent $\widehat{\mathcal{E}}[A']$ for some $\widehat{\mathcal{E}}$ and A' in the required format, then $F\widehat{\mathcal{E}}$ is the required thread context.
    If, on the other hand, $A'_{k'}$ is a form there are several possibilities depending on G.
    - If $G \equiv G' \cdot c$ for some channel c then we can apply rule *emit* and rule *use service* to conclude $GA'_{k'} \equiv cA'_{k'} \mid \epsilon$. The required canonical term is:

      $$\begin{aligned} F;BC \equiv \nu c_1 \ldots d_{n'}.(&M_1 \mid \ldots \mid M_m \mid M'_1 \mid \ldots \mid M'_{m'} \mid cA'_{k'} \\ &\mid A_1 \mid \ldots \mid A_{k-1} \mid A'_1 \mid \ldots \mid A'_{k'-1} \mid \epsilon) \end{aligned}$$

      This case adds the new message $cA'_{k'}$ to the final result.

- If $G \equiv G' \cdot hide_x$ then $GA'_{k'}$ is a preform and thus congruent to a form G''. The required canonical term is:

$$\begin{aligned} F; BC \equiv \nu c_1 ... d_{n'}.(M_1 \mid ... \mid M_m \mid M'_1 \mid ... \mid M'_{m'} \\ \mid A_1 \mid ... \mid A_{k-1} \mid A'_1 \mid ... \mid A'_{k'-1} \mid G'') \end{aligned}$$

This subcase shows that the agent $F; BC$ is a barb.

- If $G \equiv G' \cdot (x \mapsto)$ is similar to the previous case. $GA'_{k'} \equiv x \mapsto A'_{k'}$ is the required binding.
- Otherwise, G contains a closure, an inspection or a projection service, or is a form without a service. In that case, $GA'_{k'}$ is the required main agent and we have an application, case 2.

- $A = \nu c.B$. The main agent is $F; \nu c.B \equiv \nu c.F; B$ and the conclusion holds trivially by applying the induction hypothesis on $F; B$.

- $A = B \mid C$. We have $F; (B \mid C) \equiv (F; B) \mid (F; C)$ and the conclusion holds by the induction hypothesis and the scope extrusion rules.

- $A = c?$. The required thread context is $[]$ and the agent is $F; c?$, case 4. □

With this lemma, we can now prove Proposition 3.13. The lemma is used for the induction step with a sandbox expression.

Proof. Proposition 3.13 is proved by induction over A.

- $A = F$. For any form F holds by definition.

- $A = \mathbf{R}$. Trivial, case 1.

- $A = x$. Trivial, case 1.

- $A = A; B$. By the induction hypothesis we have:

$$A \equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k)$$

Now A_k is either a thread or a form. If it is a thread $\widehat{\mathcal{E}}[A']$ for some $\widehat{\mathcal{E}}$ then $\widehat{\mathcal{E}}; B$ is the required thread context. If, on the other hand, A_k is a form F, we have:

$$\begin{aligned} A; B &\equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_{k-1} \mid F); B \\ &\equiv \nu c_1 ... c_n d_1 ... d_{n'}.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_{k-1} \mid F; B) \quad \text{by rule } \textit{par sandbox left} \end{aligned}$$

By Lemma B.1, $(F; B)$ is congruent to a canonical agent.

- $A = B \cdot C$. We apply the induction hypothesis on both parts:

$$\begin{aligned} B \equiv \nu c_1 ... c_n.(M_1 \mid ... \mid M_m \mid A_1 \mid ... \mid A_k) \\ C \equiv \nu d_1 ... d_{n'}.(M'_1 \mid ... \mid M'_{m'} \mid A'_1 \mid ... \mid A'_{k'}) \end{aligned}$$

$B \cdot C$ is combined by applying scope extrusion and rule *par ext left*. A_k is either a form or a thread. Both cases can be calculated similar to the case $A \equiv B; C$ and by using the fact that $F \cdot G$ is congruent to a form.

- $A = \lambda x.B$. This is case 1 of Definition 3.11.
- $A = BC$. This case is similar to the case BC in the proof of Lemma B.1.
- $A \equiv \nu c.B$. The conclusion holds trivially by applying the induction hypothesis.
- $A \equiv B \mid C$. The conclusion holds trivially by applying the induction hypothesis and scope extrusion.
- $A \equiv c?$. The required thread context is $[]$ and we have case 4. $\square$

Appendix C

Proofs for Chapter 4

C.1 Proof of Lemma 4.12

Proof. The lemma is proved by structural induction on form processes. The base cases are easy. We only show projection for $P = \mathit{fun}\langle \tilde{f}, s\rangle$. By using a replication theorem we have that $P' = \overline{q} \mid P$ which is case (1b).

The induction steps require slightly more work. We show two cases:

- Assume $P = \mathit{bind}\langle \tilde{f}, x, \tilde{g}\rangle \mid Q \in \mathcal{F_P}$ and $y \neq x$. Then $P \xrightarrow{f_h[y,p]} P \mid \overline{p}\langle \tilde{f}\rangle$ and we have case (3) with $Q = \mathbf{0}$. On the other hand we have $P \xrightarrow{\tilde{h}[x,p]} P \mid (\nu \tilde{h})\overline{p}\langle \tilde{h}\rangle \mid \mathit{empty}\langle \tilde{h}\rangle$ which also fulfills case (3) with $Q = \mathit{empty}\langle \tilde{h}\rangle \in \mathcal{F_P}$.
- Assume $P = \mathit{ext}\langle \tilde{f}, \tilde{g}, \tilde{h}\rangle \mid Q \in \mathcal{F_P}$. We have

$$P \xrightarrow{f_s[p,q]} P \mid (\nu r_1, r_2)\overline{g_s}\langle r_1, r_2\rangle \mid \overline{h_s}\langle r_1, r_2\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q} \tag{C.1}$$

  By the induction hypothesis the process $\overline{g_s}\langle r_1, r_2\rangle$ interacts with P in either of the following ways:

$$P \mid \overline{g_s}\langle r_1, r_2\rangle \Longrightarrow P \mid \overline{r_1}\langle x\rangle \quad \text{for some } x \tag{C.2}$$

$$P \mid \overline{g_s}\langle r_1, r_2\rangle \gtrsim P \mid \overline{r_2} \tag{C.3}$$

  and the same applies for $\overline{h_s}\langle r_1, r_2\rangle$. If both interactions are given by (C.2) then we have:

$$\begin{aligned}
P \mid (\nu r_1, r_2)\overline{g_s}\langle r_1, r_2\rangle \mid \overline{h_s}\langle r_1, r_2\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q} &\gtrsim \quad \text{by (C.2)}\\
P \mid (\nu r_1, r_2)\overline{r_2} \mid \overline{r_2}\langle\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q}\langle\rangle &\gtrsim \quad \text{along } r_2\\
P \mid (\nu r_1, r_2)(r_1(x).\overline{p}x) \mid \overline{q}\langle\rangle \equiv P \mid \overline{q} &
\end{aligned}$$

  which shows case (4b). On the other hand we assume for one interaction (C.2) applies. We have:

$$\begin{aligned}
P \mid (\nu r_1, r_2)\overline{g_s}\langle r_1, r_2\rangle \mid \overline{h_s}\langle r_1, r_2\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q} &\Longrightarrow\\
P \mid (\nu r_1, r_2)\overline{r_1}\langle l\rangle \mid \overline{h_s}\langle r_1, r_2\rangle \mid (r_1(x).\overline{p}\langle x\rangle) \mid r_2.r_2.\overline{q} &\xrightarrow{\tau}\\
P \mid (\nu r_1, r_2) \mid \overline{h_s}\langle r_1, r_2\rangle \mid \overline{p}\langle x\rangle \mid r_2.r_2.\overline{q} \approx P \mid \overline{p}\langle x\rangle &
\end{aligned}$$

The ground congruence is given by the fact, that the remaining processes send at most one element along r_2. Therefor the process $r_2.\overline{q}$ can be garbage collected and case (4a) holds.

The other cases are similar and simpler. In fact, label selection for an extended form is the most interesting case, since it nondeterministically chooses a label. We have to convince ourselves that either a label is returned or the absence of any binding is signaled. □

C.2 Proof of Lemma 4.16

Proof. We show the individual cases.

(1) We show that $\mathcal{S}$ defined as $(\nu\tilde{g},\tilde{h})ext\langle\tilde{f},\tilde{h},\tilde{g}\rangle \mid empty\langle\tilde{g}\rangle \mid P_F\langle\tilde{h}\rangle \;\mathcal{S}\; P_F\langle\tilde{f}\rangle$ is a bisimulation up to expansion and context. The only interesting case is an input transition with subject in $\tilde{f}$. The other cases are simulated trivially.

- Assume $P_F\langle\tilde{f}\rangle \xrightarrow{f_p[x,p,q]} P''$. By Lemma 4.12, one of the following holds:

$$P'' \gtrsim \overline{p}\langle\tilde{g}'\rangle \mid P_F\langle\tilde{f}\rangle \qquad (*)$$
$$P'' \gtrsim \overline{q} \mid P_F\langle\tilde{f}\rangle$$

  In any case there is a static context $\mathcal{C}$ with $P'' \gtrsim \mathcal{C}[P_F\langle\tilde{f}\rangle]$. For (*) it is $\mathcal{C} = \overline{p}\langle\tilde{g}'\rangle \mid []$ On the other hand we have the following. Let $E = ext\langle\tilde{f},\tilde{h},\tilde{g}\rangle \mid empty\langle\tilde{g}\rangle$.

$$\begin{aligned} (\nu\tilde{g},\tilde{h})E \mid P_F\langle\tilde{h}\rangle &\xrightarrow{f_p[x,p,q]}\gtrsim \qquad \text{(expand } ext_p\text{)} \\ (\nu\tilde{g},\tilde{h})E \mid P_F\langle\tilde{h}\rangle \mid (\nu r_1)\overline{g_p}\langle x,p,r_1\rangle \mid r_1.\overline{h_p}\langle x,p,q\rangle &\gtrsim \\ (\nu\tilde{g},\tilde{h})E \mid P_F\langle\tilde{h}\rangle \mid \overline{h_p}\langle x,p,q\rangle &\gtrsim \\ \mathcal{C}[(\nu\tilde{g},\tilde{h})E \mid P_F\langle\tilde{h}\rangle] & \qquad \text{assuming } (*) \end{aligned}$$

  Observe that the universal quantification is handled by the static context definition according to Lemma 4.14.

- The other cases are similar.

(2) Use the same relation $\mathcal{S}$ as in case (1) but change the extension clause to $ext\langle\tilde{f},\tilde{g},\tilde{h}\rangle$.

(3) We set $P = (\nu\tilde{f},\tilde{f}',\tilde{g},\tilde{h})R \mid ext\langle\tilde{f}',\tilde{f},\tilde{g}\rangle \mid ext\langle\tilde{e},\tilde{f}',\tilde{h}\rangle$ and the right-hand side $Q = (\nu\tilde{f},\tilde{g},\tilde{g}',\tilde{h})R \mid ext\langle\tilde{g}',\tilde{g},\tilde{h}\rangle \mid ext\langle\tilde{e},\tilde{f},\tilde{g}'\rangle$. We show that $\mathcal{S} = (P,Q)$ is a bisimulation up-to expansion and context. Again, only the input transitions on $\tilde{e}$ are interesting. We show the case $\mu = \tilde{e}_h[x,p]$. The case with $\tilde{e}_s[x,p,q]$ is similar. The other two cases are simpler.

Assume $P \xrightarrow{\mu} P''$. We have to show that there is a static context $\mathcal{C}$ and P', Q' such that $P'' \gtrsim \mathcal{C}[P']$, $Q \overset{\hat{\mu}}{\Longrightarrow}\gtrsim \mathcal{C}[Q']$, and $P' \;\mathcal{S}\; Q'$.

We have:

$$\begin{aligned} P'' &\gtrsim (\nu \tilde{f}, \tilde{f}', \tilde{g}, \tilde{h}, \tilde{r})R \mid ext\langle \tilde{f}', \tilde{f}, \tilde{g}\rangle \mid ext\langle \tilde{e}, \tilde{f}', \tilde{h}\rangle \\ &\quad \mid r_1(\tilde{f}'').r_2(\tilde{h}'').(\nu \tilde{e}')\overline{p}\langle \tilde{e}'\rangle \mid ext\langle \tilde{e}', \tilde{f}'', \tilde{h}''\rangle \\ &\gtrsim (\nu \tilde{f}, \tilde{f}', \tilde{g}, \tilde{h})R \mid ext\langle \tilde{f}', \tilde{f}, \tilde{g}\rangle \mid ext\langle \tilde{e}, \tilde{f}', \tilde{h}\rangle \\ &\quad \mid (\nu \tilde{e}', \tilde{f}'', \tilde{h}'')P^* \mid ext\langle \tilde{e}', \tilde{f}'', \tilde{h}''\rangle \end{aligned}$$

P^* is constructed using Lemma 4.12(3) and the fact that $P_F\langle \tilde{f}\rangle \mid P_G\langle \tilde{g}\rangle \mid ext\langle \tilde{f}', \tilde{f}, \tilde{g}\rangle \in \mathcal{F}_\mathcal{P}$. According to Lemma 4.12 there are now two cases: If $P^* = \mathbf{0}$ the required context is:

$$\mathcal{C} = (\nu \tilde{f}, \tilde{f}', \tilde{g}, \tilde{h})R \mid []$$

On the other hand if $P^* \in \mathcal{F}_\mathcal{P}$ the context becomes:

$$\mathcal{C} = (\nu \tilde{f}, \tilde{f}', \tilde{g}, \tilde{h})R \mid ext\langle \tilde{f}', \tilde{f}, \tilde{g}\rangle \mid ext\langle \tilde{e}, \tilde{f}', \tilde{h}\rangle \mid []$$

Both cases can be simulated by a similar expansion for Q.

(4) The following calculation shows the necessary result:

$$\begin{array}{rll} [\![x{\mapsto}F]\!]_a & \approx & \text{by 4.13} \\ (\nu \tilde{r})[\![F]\!]_{r_1} \mid r_1(\tilde{f}).(\nu \tilde{g}, \tilde{f}')[\![()]\!]_{r_2} \mid bind\langle \tilde{g}, x, \tilde{f}\rangle \mid ext\langle \tilde{f}', \tilde{g}, \tilde{h}\rangle \mid r_2(\tilde{h}).\overline{a}\langle \tilde{f}'\rangle & \approx & \text{by 4.14} \\ (\nu \tilde{f}, \tilde{g}, \tilde{h}, \tilde{f}')P_F\langle \tilde{f}\rangle \mid empty\langle \tilde{h}\rangle \mid bind\langle \tilde{g}, x, \tilde{f}\rangle \mid \overline{a}\langle \tilde{f}'\rangle \mid ext\langle \tilde{f}', \tilde{g}, \tilde{h}\rangle & \approx & \text{using (1)} \\ (\nu \tilde{f}, \tilde{g})F_P\langle \tilde{f}\rangle \mid bind\langle \tilde{g}, x, \tilde{f}\rangle \mid \overline{a}\langle \tilde{g}\rangle & & \end{array}$$

(5) Apply the definition of $[\![A; B]\!]_a$.

(6) The process P that models the form F is a parallel composition of replicated input processes on private names. Thus we cannot interact with P. Note that the F does not contain any Piccola channel. □

C.3 Proof of Lemma 4.17

Proof. Assume $A \equiv B$. We prove $[\![A]\!]_a \approx [\![B]\!]_a$ by induction on the derivation of $A \equiv B$. We have to consider all congruence rules:

(reflexive) $[\![A]\!]_a \approx [\![A]\!]_a$ holds vacuously.

(symmetric) $\approx$ is symmetric by definition.

(transitive) $\approx$ is transitive since $\dot{\approx}$ is the largest relation that includes bisimulation.

(congruence) $\approx$ is a congruence by definition.

(par assoc) We have $[\![(A \mid B) \mid C]\!]_a = (\nu \tilde{r})[\![A]\!]_{r_1} \mid [\![B]\!]_{r_2} \mid [\![C]\!]_a \equiv [\![A \mid (B \mid C)]\!]_a$ by structural congruence on π-processes.

(par left commute) We have $[\![(A \mid B) \mid C]\!]_a = (\nu \tilde{r})[\![A]\!]_{r_1} \mid [\![B]\!]_{r_2} \mid [\![C]\!]_a \equiv [\![(B \mid A) \mid C]\!]_a$ by structural congruence on π-processes.

(par ext left) The process $[\![(A \mid B) \cdot C]\!]_a$ has the form $(\nu\tilde{r})[\![A \mid B]\!]_{r_1} \mid r_1(\tilde{f}).P$ and by Lemma 4.15(3) this is congruent to $[\![A \mid B \cdot C]\!]_a$.

(par ext right) We have $[\![F \cdot (A \mid B)]\!]_a = (\nu\tilde{r})[\![F]\!]_{r_1} \mid r_1(\tilde{f}).([\![A]\!]_{r_3} \mid P)$ and $\tilde{f} \notin fv([\![A]\!]_{r_3})$. Thus by Lemma 4.15(1) this is congruent to $(\nu\tilde{r})[\![A]\!]_{r_3} \mid [\![F]\!]_{r_1} \mid r_1(\tilde{f}).P$ and this is the process $[\![A \mid (F \cdot B)]\!]_a$.

(par app left) Using Lemma 4.15(3), we have $[\![(A \mid B)C]\!]_a = (\nu\tilde{r})[\![A \mid B]\!]_{r_1} \mid r_1(\tilde{f}).P$ which is congruent to $[\![A \mid BC]\!]_a$.

(par app right) We have $[\![F(A \mid B)]\!]_a = (\nu\tilde{r})[\![F]\!]_{r_1} \mid r_1(\tilde{f}).([\![A]\!]_{r_3} \mid [\![B]\!]_{r_2}) \mid r_2(\tilde{g}).P$ and again by Lemma 4.15(1) this is $(\nu\tilde{r})[\![A]\!]_{r_3} \mid [\![F]\!]_{r_1} \mid r_1(\tilde{f}).[\![B]\!]_{r_2} \mid r_2(\tilde{g}).P = [\![A \mid FB]\!]_a$.

(par sandbox left) We have $[\![(A \mid B); C]\!]_a = (\nu\tilde{r})[\![A]\!]_{r_2} \mid [\![B]\!]_{r_1} \mid r_1(\tilde{f}).[\![C]\!]_a^{\tilde{f}}$. Using Lemma 4.15(3) this is congruent to $[\![A \mid B; C]\!]_a$.

(par sandbox right) $[\![F; (A \mid B)]\!]_a = (\nu\tilde{r})[\![F]\!]_{r_1} \mid r_1(\tilde{f}).[\![A]\!]_{r_2}^{\tilde{f}} \mid [\![B]\!]_a^{\tilde{f}}$. By Lemma 4.15(2) we can duplicate the form F and conclude

$$(\nu\tilde{r})[\![F]\!]_{r_1} \mid r_1(\tilde{f}).[\![A]\!]_{r_2}^{\tilde{f}} \mid [\![F]\!]_{r_3} \mid r_3(\tilde{f}).[\![B]\!]_a^{\tilde{f}} = [\![F; A \mid F; B]\!]_a$$

(discard zombie) We have $[\![F \mid A]\!]_a = (\nu\tilde{r})[\![F]\!]_{r_1} \mid [\![A]\!]_a$. By Lemma 4.16(6) we conclude $(\nu\tilde{r})[\![F]\!]_{r_1} \mid [\![A]\!]_a \approx \mathbf{0} \mid [\![A]\!]_a \equiv [\![A]\!]_a$.

(commute channels) The property $[\![\nu c, d.A]\!]_a \equiv [\![\nu d, c.A]\!]_a$ follows directly from the translation of scopes.

(scope par left) Follows directly from the scoping rules π. Assume $c \notin fc(A)$. We have $[\![A \mid \nu c.B]\!]_a = (\nu r_1, c)[\![A]\!]_{r_1} \mid [\![B]\!]_a = [\![\nu c.A \mid B]\!]_a$.

The other scope extrusion rules scope par right, scope ext left, scope ext right, scope sandbox left, scope sandbox right, scope app right, and scope app left are similar to this case.

(emit) A calculation yields:

$$\begin{aligned}
[\![cF]\!]_a &= (\nu\tilde{r})[\![c]\!]_{r_1} \mid r_1(\tilde{f}).[\![F]\!]_{r_2} \mid r_2(\tilde{g}).\overline{f_i}\langle r_3, r_4\rangle \mid r_3(s).\bar{s}\langle \tilde{g}, a\rangle \\
&\approx (\nu\tilde{r}, \tilde{f}, s)fun\langle \tilde{f}, s\rangle \mid (!s(\tilde{g}, r).(\nu\tilde{h})\bar{c}\langle \tilde{g}\rangle \mid \bar{p}\langle \tilde{h}\rangle \mid empty\langle \tilde{h}\rangle) \mid [\![F]\!]_{r_2} \mid r_2(\tilde{g}).P \\
&\approx (\nu\tilde{r}, \tilde{f}, s)fun\langle \tilde{f}, s\rangle \mid (!s(\tilde{g}, r).\bar{c}\langle \tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid [\![F]\!]_{r_2} \mid r_2(\tilde{g}).P
\end{aligned}$$

by Lemma 4.14 there exists a Q such that:

$$\begin{aligned}
&\approx (\nu\tilde{r}, \tilde{f}, s)fun\langle \tilde{f}, s\rangle \mid (!s(\tilde{g}, r).\bar{c}\langle \tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})\overline{r_2}\langle \tilde{g}\rangle \mid Q \mid r_2(\tilde{g}).P \\
&\approx (\nu\tilde{r}, \tilde{f}, s)fun\langle \tilde{f}, s\rangle \mid (!s(\tilde{g}, r).\bar{c}\langle \tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \overline{f_i}\langle r_3, r_4\rangle \mid r_3(s).\bar{s}\langle \tilde{g}, a\rangle \\
&\approx (\nu\tilde{r}, \tilde{f}, s)fun\langle \tilde{f}, s\rangle \mid (!s(\tilde{g}, r).\bar{c}\langle \tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \overline{r_3}\langle s\rangle \mid r_3(s).\bar{s}\langle \tilde{g}, a\rangle
\end{aligned}$$

The process $fun\langle\tilde{f}, s\rangle$ is not used anymore:

$$\begin{aligned}
&\approx (\nu\tilde{r}, s)(!s(\tilde{g}, r).\bar{c}\langle\tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \overline{r_3}\langle s\rangle \mid r_3(s).\bar{s}\langle\tilde{g}, a\rangle \\
&\approx (\nu\tilde{r}, s)(!s(\tilde{g}, r).\bar{c}\langle\tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \overline{r_3}\langle s\rangle \mid r_3(s).\bar{s}\langle\tilde{g}, a\rangle \\
&\approx (\nu\tilde{r}, s)(!s(\tilde{g}, r).\bar{c}\langle\tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \bar{s}\langle\tilde{g}, a\rangle \\
&\approx (\nu\tilde{r}, s)(!s(\tilde{g}, r).\bar{c}\langle\tilde{g}\rangle \mid [\![\epsilon]\!]_r) \mid (\nu\tilde{g})Q \mid \bar{c}\langle\tilde{g}\rangle \mid [\![\epsilon]\!]_a
\end{aligned}$$

Thus we have $[\![cF \mid \epsilon]\!]_a = (\nu r)[\![cF]\!]_r \mid [\![\epsilon]\!]_a = [\![cF]\!]_a$. Observe that the calculation uses the fact that the sent expression is available as a form F. If it was an arbitrary agent, we could not use Lemma 4.14, thus $cA \not\approx cA \mid \epsilon$.

(sandbox ext) Using Lemma 4.16(5) and the replication theorems we have $[\![F; A \cdot B]\!]_a \approx [\![(F; A) \cdot (F; B)]\!]_a$.

(sandbox app) Similar to the previous case. Observe that both definition of $[\![A \cdot B]\!]_a$ and $[\![AB]\!]_a$ first make A and B available and differ only in the process that receives the value of B.

(sandbox assoc) $[\![(A; B); C]\!]_a \approx (\nu\tilde{r}[\![A]\!]_{r_1} \mid (r_1(\tilde{f}).[\![B]\!]^{\tilde{f}}_{r_2}) \mid (r_2(\tilde{g}).[\![C]\!]^{\tilde{g}}_a) \approx [\![A; (B; C)]\!]_a$ due to the associativity of the parallel operator.

(sandbox value) $[\![F; G]\!]_a = (\nu r)[\![F]\!]_r \mid r(\tilde{f}).[\![G]\!]^{\tilde{f}}_a$. We use Lemma 4.14 for $\tilde{f} \notin fv([\![G]\!]^{\tilde{f}}_a)$ and that the emission of F can be rewritten as: $(\nu\tilde{f})\bar{r}\langle\tilde{f}\rangle \mid P_F$. Thus the communication along r can be optimized away: $[\![F; G]\!] \approx (\nu\tilde{f})P_F \mid [\![G]\!]_a \equiv [\![G]\!]_a$.

(sandbox root) $[\![F; \mathbf{R}]\!]_a = (\nu r)[\![F]\!]_r \mid r(\tilde{f}).\bar{a}\langle\tilde{f}\rangle$. By Lemma 4.14, we have $[\![F]\!]_r \approx (\nu\tilde{f})P_F \mid \bar{r}\langle\tilde{f}\rangle$. Thus we have $[\![F; \mathbf{R}]\!]_a \approx (\nu\tilde{f}, r)P_F \mid \bar{r}\langle\tilde{f}\rangle \mid r(\tilde{f}).\bar{a}\langle\tilde{f}\rangle \approx (\nu\tilde{f})P_F \mid \bar{a}\langle\tilde{f}\rangle = [\![F]\!]_a$.

(ext empty right) We use Lemma 4.16(1) to infer

$$\begin{aligned}
[\![F \cdot \epsilon]\!]_a &\approx (\nu\tilde{f}, \tilde{g}, \tilde{h})P_F\langle\tilde{f}\rangle \mid empty\langle\tilde{g}\rangle \mid ext\langle\tilde{h}, \tilde{f}, \tilde{g}\rangle \mid \bar{a}\langle\tilde{h}\rangle \\
&\approx (\nu\tilde{f})P_F\langle\tilde{f}\rangle \mid \bar{a}\langle\tilde{f}\rangle \\
&\approx [\![F]\!]_a
\end{aligned}$$

as desired.

(ext empty left) Similar to the previous case but use Lemma 4.16(2).

(ext assoc) Let $R = P_F\langle\tilde{f}\rangle \mid P_G\langle\tilde{g}\rangle \mid P_H\langle\tilde{h}\rangle$. Now using Lemma 4.16(3) we have:

$$\begin{aligned}
[\![(F \cdot G) \cdot H]\!]_a &\approx (\nu\tilde{f}, \tilde{f}', \tilde{g}, \tilde{h}, \tilde{e}) \mid R \mid ext\langle\tilde{f}', \tilde{f}, \tilde{g}\rangle \mid ext\langle\tilde{e}, \tilde{f}', \tilde{h}\rangle \mid \bar{a}\langle\tilde{e}\rangle \\
&\approx (\nu\tilde{f}, \tilde{f}', \tilde{g}, \tilde{h}, \tilde{e}) \mid R \mid ext\langle\tilde{f}', \tilde{g}, \tilde{h}\rangle \mid ext\langle\tilde{e}, \tilde{f}, \tilde{f}'\rangle \mid \bar{a}\langle\tilde{e}\rangle \approx [\![F \cdot (G \cdot H)]\!]_a
\end{aligned}$$

(ext service commute) To see that $[\![S \cdot (x \mapsto F)]\!]_a \approx [\![(x \mapsto F) \cdot S]\!]_a$ we have to convince ourselves that

$$\begin{aligned}
P &= fun\langle\tilde{f}, s\rangle \mid bind\langle\tilde{g}, x, \tilde{h}\rangle \mid ext\langle\tilde{e}, \tilde{f}, \tilde{g}\rangle, \text{ and} \\
Q &= fun\langle\tilde{f}, s\rangle \mid bind\langle\tilde{g}, x, \tilde{h}\rangle \mid ext\langle\tilde{e}, \tilde{g}, \tilde{f}\rangle
\end{aligned}$$

are ground bisimilar. The only interesting cases are input on a channel of $\tilde{e}$. The bisimulation can be computed: Projection yields in both cases either the empty form or the tuple $\tilde{h}$ if projection is done on label x. The service channel returned by requests on $\tilde{e}_i$ is s in both cases. Hiding requests for a label different than x yield congruent processes again. Hiding requests on the label x either return forms $\epsilon \cdot G$ or $G \cdot \epsilon$ which are congruent by Lemma 4.16. Label selection returns the label x in both cases.

(ext bind commute) Similar to the previous case. The congruent processes P and Q are:

$$P = bind\langle \tilde{f}, y, \tilde{h}' \rangle \mid bind\langle \tilde{g}, x, \tilde{h} \rangle \mid ext\langle \tilde{e}, \tilde{f}, \tilde{g} \rangle$$
$$Q = bind\langle \tilde{f}, y, \tilde{h}' \rangle \mid bind\langle \tilde{g}, x, \tilde{h} \rangle \mid ext\langle \tilde{e}, \tilde{g}, \tilde{f} \rangle$$

As in the previous case, projection, service selection, label hiding and inspection behave the same.

(use service) The process $P = [\![B]\!]^{\tilde{e}}_{r_2} \mid r_2(\tilde{g}).\overline{f_i}\langle r_3, r_4 \rangle \mid r_3(s).\overline{s}\langle \tilde{g}, a \rangle$ that is part of $[\![AB]\!]_a$ only has f_i as a free name of $\tilde{f}$. Thus to see that $[\![(F \cdot S)A]\!]_a \approx [\![SA]\!]_a$ we have to show that $\mathcal{S} = \{(P, Q)\}$ with

$$P = (\nu f_p, f_h, f_s) fun\langle \tilde{f}, s \rangle \mid P_G\langle \tilde{g} \rangle$$
$$Q = (\nu f_p, f_h, f_s, \tilde{h}) ext\langle \tilde{f}, \tilde{g}, \tilde{h} \rangle \mid fun\langle \tilde{h}, s \rangle \mid P_G\langle \tilde{g} \rangle$$

is a ground bisimulation. The interesting transitions are input on f_i. It is readily checked that $\mathcal{S}$ is a ground bisimulation.

(single service) To see that $[\![S \cdot S']\!]_a \approx [\![S']\!]_a$ use the bisimulation $\mathcal{S}$:

$$\mathcal{S} \overset{\text{def}}{=} \{(fun\langle \tilde{f}, s \rangle, P) \text{ with } P = (\nu \tilde{g}, \tilde{h}) ext\langle \tilde{f}, \tilde{g}, \tilde{h} \rangle \mid fun\langle \tilde{g}, s \rangle \mid fun\langle \tilde{h}, s' \rangle\}$$

It is readily verified that $\mathcal{S}$ is a ground bisimulation.

(single binding) Similar to the previous case.

(hide select) The terms $[\![hide_x(F \cdot x{\mapsto}G)]\!]_a$ and $[\![hide_x F]\!]_a$ are ground bisimilar. To see this we first use expansion to reduce the invocation of the hide feature on the encodings of $F \cdot x{\mapsto}G$ and F. By Lemma 4.12 and Lemma 4.16(2) $ext\langle \tilde{h}, \tilde{f}, \tilde{g}' \rangle \mid bind\langle \tilde{g}', x\tilde{g} \rangle \mid \overline{h_h}\langle x, r \rangle$ expands to a process $\overline{f_h}\langle x, r \rangle$ which shows the desired result.

(hide over) A similar argument like in the previous case.

(hide empty) Follows from the definition of $empty\langle \tilde{f} \rangle$.

(hide service) Follows from the definition of $fun\langle \tilde{f}, s \rangle$. □

Appendix D

Proofs for Chapter 6

In this appendix we formally prove correctness of the partial evaluation algorithm. We first show a few helper lemmas that use the notion of referential transparency. For short, with referential transparent terms we can use distribution and substitution as derived from a calculus with the Church-Rosser property and without explicit namespaces.

The main part of the correctness proof is then given by Lemma D.12 which is an induction of functional agents showing correctness for the function *split*.

In order to simplify reading of the embedding of lazy forms R and the combining of side-effect terms P we use the following notation:

$$embed(R) = \lfloor R \rfloor \qquad\qquad combine'(P) = \lceil P \rceil$$

The first lemma proves correctness for the *labels* predicate.

Lemma D.1 *For $F \vdash \lfloor R \rfloor$ with $G \approx F; \lfloor R \rfloor$ it holds: $labels(R) \subseteq labels(G)$.*

Proof. By induction on R. □

By this above lemma, the set of labels cannot shrink when we reduce a lazy form (or — more precisely — its embedding) to a form. For an example where the set grows consider $R = (x{\mapsto}y \cdot z)$ and $F = (y{\mapsto}\epsilon \cdot z{\mapsto}(a{\mapsto}\epsilon))$. It holds that:

$$\begin{aligned} F; \lfloor R \rfloor = G &\approx (x{\mapsto}\epsilon \cdot a{\mapsto}\epsilon) \\ labels(R) &= \{x\} \\ labels(G) &= \{x, a\} \end{aligned}$$

With lazy forms, the sets of labels can grow when we evaluate the lazy form to an ordinary form. In contrast, the sets of labels in a side-effect form remains the same. This is due to the fact that all side-effects are bound by nested forms.

Lemma D.2 *For any side-effect form P that reduces to a form F as follows:*

$$A \mid \lceil P \rceil \Rightarrow A' \mid F$$

it holds: $labels(F) = labels(P)$.

Proof. By induction on the top-level reductions of $\lceil P \rceil$. □

Disjoint labels of side effect terms and referentially transparency give rise to some powerful algebraic laws:

Lemma D.3 *For any $A \vdash B$ it holds: $A; B; F \approx A; F$*

Proof. Obvious since $G; F \approx F$. $\square$

Lemma D.4 *For $\lceil P_1 \rceil \vdash \lfloor R_1 \rfloor$ and $\lceil P_2 \rceil \vdash \lfloor R_2 \rfloor$ with labels$(P_1) \cap$ labels$(P_2) = \emptyset$ the following holds:*

$$(\lceil P_1 \rceil; \lfloor R_1 \rfloor) \cdot (\lceil P_2 \rceil; \lfloor R_2 \rfloor) \approx \lceil P_1 \rceil; \lceil P_2 \rceil; \lfloor R_1 \rfloor \cdot \lfloor R_2 \rfloor \tag{D.1}$$
$$(\lceil P_1 \rceil; \lfloor R_1 \rfloor)(\lceil P_2 \rceil; \lfloor R_2 \rfloor) \approx \lceil P_1 \rceil; \lceil P_2 \rceil; \lfloor R_1 \rfloor \cdot \lfloor R_2 \rfloor \tag{D.2}$$

Proof. Diagram chasing shows that the equations are bisimulations. We only show the first congruence.

Assume agent $(\lceil P_1 \rceil; \lfloor R_1 \rfloor) \cdot (\lceil P_2 \rceil; \lfloor R_2 \rfloor)$ reduces to $F_1; \lfloor R_1 \rfloor) \cdot (\lceil P_2 \rceil; \lfloor R_2 \rfloor)$. There must be a G_1 such that $F_1; \lfloor R_1 \rfloor \approx G_1$. Then $G_1 \cdot (\lceil P_2 \rceil; \lfloor R_2 \rfloor)$ reduces for the same reason to $G_1 \cdot G_2$. On the opposite: $\lceil P_1 \rceil; \lceil P_2 \rceil; \lfloor R_1 \rfloor \cdot \lfloor R_2 \rfloor$ reduces to

$$F_1; \mathbf{R} \cdot F_2; \lfloor R_1 \rfloor \lfloor R_2 \rfloor \approx (F_1; \mathbf{R} \cdot F_2; \lfloor R_1 \rfloor) \cdot (F_1; \mathbf{R} \cdot F_2; \lfloor R_2 \rfloor)$$

Since the labels of F_1 and of F_2 are disjoint this is equivalent to $(F_1; \lfloor R_1 \rfloor) \cdot (F_2; \lfloor R_2 \rfloor) \approx G_1 \cdot G_2$. $\square$

An important property of lazy forms is that we can distribute them over subterms. Observe that the congruence rules of the Piccola calculus distribute only forms, the following lemma allows us to distribute lazy forms.

Lemma D.5 *For any agent C with $C \vdash \lfloor R \rfloor$ it holds:*

$$C; \lfloor R \rfloor; A \cdot B \approx C; (\lfloor R \rfloor; A) \cdot (\lfloor R \rfloor; B) \tag{D.3}$$
$$C; \lfloor R \rfloor; AB \approx C; (\lfloor R \rfloor; A)(\lfloor R \rfloor; B) \tag{D.4}$$

Proof. Similar to the proof of Lemma D.4 $\square$

The following lemma shows that we can move the context R into an abstraction. We have to take care that no name capture occurs. Notice that the congruence is only valid for lazy contexts which do not contain the keyword $\mathbf{R}$ anymore.

Lemma D.6 *For $F \vdash \lfloor R \rfloor$ and $x \notin fv(R)$ it holds:*

$$F; \lambda x.(\lfloor R \rfloor \cdot x \mapsto x; B) \approx F; \lfloor R \rfloor; \lambda x.B$$

Proof. Since $F \vdash \lfloor R \rfloor$ there is a form F' with $F; \lfloor R \rfloor \approx F'$. Now we have:

$$\begin{aligned} F; \lfloor R \rfloor; \lambda x.B &\approx F'; \lambda x.B \\ &\approx \epsilon; \lambda x.(F' \cdot x \mapsto x; B) \\ &\approx \epsilon; \lambda x.(F; \lfloor R \rfloor \cdot x \mapsto x; B) \end{aligned}$$

since R does not contain x free

$$\begin{aligned} &\approx \epsilon; \lambda x.(F \cdot x \mapsto x; \lfloor R \rfloor \cdot x \mapsto x; B) \\ &\approx F; \lambda x.(\lfloor R \rfloor \cdot x \mapsto x; B) \end{aligned}$$

as expected. $\square$

The next lemma allows us to use some form of alpha conversion for lazy forms. Since agents of the form $\lceil P \rceil; \lfloor R \rfloor$ do not refer to **R** in $\lfloor R \rfloor$ we can rename bound names.

Lemma D.7 *For all labels x, y, lazy forms R and side effect terms P with $y \notin fv(R \cdot P)$ it holds:*

$$\lambda x.(\lceil P \rceil; \lfloor R \rfloor) \approx \lambda y.(\mathbf{R} \cdot x{\mapsto}y; \lceil P \rceil; \lfloor R \rfloor)$$

Proof. We show that the relation defined by putting both sides of the equation into any context is a bisimulation. The only interesting context F is when it is put into a sandbox and invoked with an argument G. Let $A = \lceil P \rceil; \lfloor R \rfloor$. On one side we have:

$$\begin{aligned}(F; \lambda y.(\mathbf{R} \cdot x{\mapsto}y; A))G &\rightarrow F \cdot y{\mapsto}G; \mathbf{R} \cdot x{\mapsto}y; A \\ &\approx F \cdot y{\mapsto}G \cdot x{\mapsto}G; A\end{aligned}$$

On the other side we also have:

$$(F; \lambda x.A)G \rightarrow F \cdot x{\mapsto}G; A$$

And the two expressions on the right-hand side are bisimular, since A does not refer to y. The important property is that lazy forms do not refer to **R** thus the label y can be ignored. □

The following lemma relates lazy forms and substitution.

Lemma D.8 *For any form F and G and any lazy form R it holds:*

$$F \cdot x{\mapsto}G; \lfloor R \rfloor \approx F; \lfloor R[x/G] \rfloor$$

Proof. By induction over R. □

The following lemma strengthens the beta equivalence law of the Piccola calculus. We can substitute a lazy form provided it is embedded in the right context. Observe that we extend F with $x{\mapsto}\epsilon$. We do this because the empty form is the minimal assumption we can make for the argument x.

Lemma D.9 *For any P, R, P_1, R_1 and x with $P \cdot x{\mapsto}\epsilon; \lceil P_1 \rceil \vdash \lfloor R_1 \rfloor$ and $P \vdash R$ we have:*

$$P; (\lambda x.(\lceil P_1 \rceil; \lfloor R_1 \rfloor))R \approx P; \lceil P_1[x/R] \rceil; \lfloor R_1[x/R] \rfloor$$

Proof. By induction over P_1 and R_1. We only show a few cases.

- Case $P_1 = \epsilon, R_1 = \epsilon$. We have to show that

  $$P; (\lambda x.(\mathbf{R}; \epsilon))R \approx P; \mathbf{R}; \epsilon$$

  Both terms can only reduce when P has reduced to a form F. Since $P \vdash R$ there is a G with $G \approx F; \lfloor R \rfloor$. Thus we have:

  $$\begin{aligned}F; \lambda x.(\mathbf{R}; \epsilon)G &\approx F; x{\mapsto}G; \epsilon \\ F; \mathbf{R}; \epsilon &\approx \epsilon\end{aligned}$$

- Case $P_1 = \epsilon, R_1 = \lambda y.P'; R'$. We have to show that

  $$P; \lambda x.(\mathbf{R}; \lambda y.\lceil P' \rceil; \lfloor R' \rfloor)R \approx P; \mathbf{R}; (\lambda y.(P'; R'))[x/R]$$

  Like above, we assume that P has reduced to a form F and consequently R became G. Thus the left-hand side of the above congruence becomes:

  $$F; (\lambda x.(\mathbf{R}; \lambda y.\lceil P' \rceil; \lfloor R' \rfloor))F' \approx F \cdot x{\mapsto}F'; \lambda y.(\lceil P' \rceil; \lfloor R' \rfloor)$$

  and the conclusion follows by Lemma D.8.

- Case $P_1 = y \mapsto R_2 R_3, R_1 = y$. This is the interesting case since we look up the side-effect y in the lazy form. We have to show

  $$P; (\lambda x.(\mathbf{R} \cdot y \mapsto R_2 R_3; y))R \approx P; \mathbf{R} \cdot y \mapsto R_2 R_3[x/R]; y$$

  Like before the above expression can only reduce after P has reduced to a F and consequently R became G. Now the left-hand side becomes

  $$F; x \mapsto G \cdot y \mapsto R_3 R_4; y$$

  and $R_3 R_4$ will have a free x. By Lemma D.8 the conclusion holds. □

Finally, the following lemma relates nesting of side effects:

Lemma D.10 *For any side effects terms P and lazy forms R with y fresh it holds:*

$$\lceil P \rceil; \lfloor R \rfloor \approx y \mapsto \lceil P \rceil; \lfloor nest(R, y, P) \rfloor$$

Proof. By induction on the length of P. □

Proposition D.11 (Correctness) *For all functional agents A and forms F it holds:*

$$partial(A, F) \approx F; A$$

In order to show Proposition D.11 we show a stronger result from which the proposition is a special case since forms are subsumed by lazy forms.

Lemma D.12 *For every functional agent A, and every lazy form R' there exists a lazy form R and a side effect term P such that it holds:*

1. *$partial(A, R') \approx \lfloor R' \rfloor; A$*
2. *Let $(P, R) = split(A, R')$. For every form F with $F \vdash \lfloor R' \rfloor$ it holds:*

$$F; \lceil P \rceil \vdash \lfloor R \rfloor$$

Proof. By parallel structural induction over A. Both properties are needed in order to show the induction steps.

- Case ϵ: It holds $partial(\epsilon, R') = \mathbf{R}; \epsilon$ and conclusion 1 holds by Lemma D.3. Conclusion 2 holds trivially.
- Case $\mathbf{R}$: We have $partial(\mathbf{R}, R') = \mathbf{R}; \lfloor R' \rfloor \approx \lfloor R' \rfloor \approx \lfloor R' \rfloor; \mathbf{R}$ which is conclusion 1. Conclusion 2 follows by Lemma D.1 and the fact that $F; \mathbf{R} \equiv F$.
- Case $\mathbf{L}$: We have $partial(\mathbf{L}, R) = \mathbf{R}; \mathbf{L} \equiv \mathbf{L}$ which shows 1. Conclusion 2 holds trivially as $\mathbf{L}$ is a service.

- **run** and **new** are similar.
- Case x: We distinguish if R' contains the label x:
  - $x \in \mathit{labels}(R')$. We have $\mathit{split}(x, R') = (\epsilon, \mathit{project}(R', x))$. Conclusion 1 can be readily verified by the definition of *project*. For 2 we have to show
  $$F; \mathbf{R} \vdash \mathit{project}(R', x)$$
  for any $F \vdash \lfloor R' \rfloor$. Consider the following: $F \vdash \lfloor R' \rfloor$ implies $F; \mathbf{R} \vdash \lfloor R' \rfloor$. If $\mathit{project}(R', x)$ is a proper subterm of R' the conclusion holds. Otherwise it has the form $R''.x$ with R'' a proper subterm of R' and $x \in \mathit{labels}(R')$. The conclusion follows from Lemma D.1 since the set of labels cannot shrink.
  - $x \notin \mathit{labels}(R')$. We have $\mathit{split}(x, R') = (y{\mapsto}(R'; x), y)$ with y a fresh label. Conclusion 1: $\mathbf{R} \cdot y{\mapsto}(\lfloor R' \rfloor; x); y \approx R'; x$ is a trivial indirection along y. For the other part we have to show that
  $$F; \mathbf{R} \cdot y{\mapsto}(\lfloor R' \rfloor; x) \vdash y$$
  which is trivial since the left-hand side will always contain the unique label y.
- Case $x{\mapsto}A$: Let $\mathit{split}(A, R') = (P, R)$. By the induction hypothesis we assume:
$$\lceil P \rceil; \lfloor R \rfloor \approx \lfloor R' \rfloor; A \tag{D.5}$$
$$F; \lceil P \rceil \vdash \lfloor R \rfloor \qquad \text{for } F \vdash \lfloor R' \rfloor \tag{D.6}$$
Now we have
$$\begin{aligned} \lfloor R' \rfloor; x{\mapsto}A &\approx x{\mapsto}(\lfloor R' \rfloor; A) \\ &\approx x{\mapsto}(\lceil P \rceil; \lfloor R \rfloor) \\ &\approx \lceil P \rceil; x{\mapsto}\lfloor R \rfloor \end{aligned}$$
which shows the first claim. The second part
$$F; \lceil P \rceil \vdash x{\mapsto}\lfloor R \rfloor$$
follows directly form its induction hypothesis.
- Case $A; B$. Let $\mathit{split}(A, R') = (P_1, R_1)$ and $\mathit{split}(B, R_1) = (P_2, R_2)$. By the induction hypothesis we assume:
$$\begin{aligned} \lceil P_1 \rceil; \lfloor R_1 \rfloor &\approx \lfloor R' \rfloor; A \\ \lceil P_2 \rceil; \lfloor R_2 \rfloor &\approx \lfloor R_1 \rfloor; B \\ F_1; \lceil P_1 \rceil &\vdash \lfloor R_1 \rfloor & \text{for } F_1 \vdash \lfloor R' \rfloor \\ F_2; \lceil P_2 \rceil &\vdash \lfloor R_2 \rfloor & \text{for } F_2 \vdash \lfloor R_1 \rfloor \end{aligned}$$
Part 2: $F; \lceil P_1 \rceil; \lceil P_2 \rceil \vdash \lfloor R_2 \rfloor$ follows directly from the hypothesis. The first part follows readily:
$$\begin{aligned} \mathit{partial}(A; B, R') &= \lceil P_1 \rceil; \lceil P_2 \rceil; \lfloor R_2 \rfloor \\ &= \lceil P_1 \rceil; \mathit{partial}(B; R_1) \\ &\approx \lceil P_1 \rceil; \lfloor R_1 \rfloor; B \\ &= \mathit{partial}(A, R'); B \\ &\approx \lfloor R' \rfloor; A; B \end{aligned}$$

- Case $A \cdot B$. Let $split(A, R') = (P_1, R_1)$ and $split(B, R') = (P_2, R_2)$. By the induction hypothesis we can now assume:

$$\begin{aligned} \lceil P_1 \rceil; \lfloor R_1 \rfloor &\approx \lfloor R' \rfloor; A \\ \lceil P_2 \rceil; \lfloor R_2 \rfloor &\approx \lfloor R' \rfloor; B \\ F; \lceil P_1 \rceil &\vdash \lfloor R_1 \rfloor \qquad\qquad \text{for } F \vdash \lfloor R' \rfloor \\ F; \lceil P_2 \rceil &\vdash \lfloor R_2 \rfloor \end{aligned}$$

  Part 2: $F; \lceil P_1 \rceil; \lceil P_2 \rceil \vdash \lfloor R_1 \cdot R_2 \rfloor$ follows from the fact that $labels(P_1) \cap labels(P_2) = \emptyset$. This is guaranteed by choosing unique labels for binding the side effects. Part 1 uses Lemma D.4 and Lemma D.5:

$$\begin{aligned} partial(A \cdot B; R') &= \lceil P_1 \rceil; \lceil P_2 \rceil; \lfloor R_1 \rfloor \cdot \lfloor R_2 \rfloor \\ &\approx (\lceil P_1 \rceil; \lfloor R_1 \rfloor) \cdot (\lceil P_2 \rceil; \lfloor R_2 \rfloor) \\ &= (\lfloor R' \rfloor; A) \cdot (\lfloor R' \rfloor; B) \\ &\approx \lfloor R' \rfloor; A \cdot B \end{aligned}$$

  Note that the side-conditions for these lemmas are given by the induction hypothesis part 2.

- Case $\lambda x.A$. Let $split(A, R' \cdot x{\mapsto}x) = (P, R)$. By the hypothesis it holds:

$$\begin{aligned} \lceil P \rceil; \lfloor R \rfloor &\approx \lfloor R' \rfloor \cdot x{\mapsto}x; A \\ F; \lceil P \rceil &\vdash \lfloor R \rfloor \qquad\qquad \text{for } F \cdot x{\mapsto}\epsilon \vdash \lfloor R' \rfloor \end{aligned}$$

  For part 2

$$F; \mathbf{R} \vdash \lambda x.(\lceil P \rceil; \lfloor R \rfloor)$$

  holds trivially since $F; \lambda x.A$ is a service.

  For part 1 we assume that x does not occur free in R'. Otherwise we apply Lemma D.7. We can then use Lemma D.6 to move $\lfloor R' \rfloor$ out of the abstraction:

$$\begin{aligned} partial(\lambda x.A, R') &= \mathbf{R}; \lambda x.(\lceil P \rceil; \lfloor R \rfloor) \\ &\approx \lambda x.(\lfloor R' \rfloor \cdot x{\mapsto}x; A) \\ &\approx \lfloor R' \rfloor; \lambda x.A \end{aligned}$$

- Case: AB. This case is the most interesting case. The induction hypothesis and variables are the same as for $A \cdot B$:

$$\begin{aligned} split(A, R') &= (P_1, R_1) \\ split(B, R') &= (P_2, R_2) \\ \lceil P_1 \rceil; \lfloor R_1 \rfloor &\approx \lfloor R' \rfloor; A \\ \lceil P_2 \rceil; \lfloor R_2 \rfloor &\approx \lfloor R' \rfloor; B \\ F; \lceil P_1 \rceil &\vdash \lfloor R_1 \rfloor \qquad\qquad \text{for } F \vdash \lfloor R' \rfloor \\ F; \lceil P_2 \rceil &\vdash \lfloor R_2 \rfloor \end{aligned}$$

  The labels in P_1 and P_2 are disjoint. Now there are two main cases depending on $service(R_1)$: For easy of reference we omit $service$.

– Case $R_1 = \lambda x.(\lceil P_3 \rceil; \lfloor R_3 \rfloor)$. We have to show

$$\begin{aligned} & \lfloor R' \rfloor; AB \approx \mathit{partial}(AB, R') \\ & \approx \lceil P_1 \rceil; \lceil P_2 \rceil; y \mapsto \lceil P_3[x/R_2] \rceil; \lfloor \mathit{nest}(R_3, y, P_3)[x/R_2] \rfloor && \text{(D.7)} \\ & F; \lceil P_1 \rceil; \lceil P_2 \rceil; y \mapsto \lceil P_3[x/R_2] \rceil \vdash \lfloor \mathit{nest}(R_3, y, P_3)[x/R_2] \rfloor && \text{(D.8)} \end{aligned}$$

Calculate:

$$\begin{aligned} & \lceil P_1 \rceil; \lceil P_2 \rceil; y \mapsto \lceil P_3[x/R_2] \rceil; \lfloor \mathit{nest}(R_3, y, P_3)[x/R_2] \rfloor \\ & \approx \lceil P_1 \rceil; \lceil P_2 \rceil; \lceil P_3[x/R_2] \rceil; \lfloor R_3[x/R_2] \rfloor && \text{by Lemma D.10} \\ & \approx \lceil P_1 \rceil; \lceil P_2 \rceil; (\lambda x. \lceil P_3 \rceil; \lfloor R_3 \rfloor) \lfloor R_2 \rfloor && \text{by Lemma D.9} \\ & \approx (\lceil P_1 \rceil; (\lambda x. \lceil P_3 \rceil; \lfloor R_3 \rfloor))(\lceil P_2 \rceil; \lfloor R_2 \rfloor) && \text{by Lemma D.4} \\ & \approx (R'; A)(R'; B) \\ & \approx (R'; AB) \end{aligned}$$

A special case is when $P_3 = \epsilon$. In that case $\epsilon[x/R_2] = \epsilon$ and thus the application of Lemma D.10 can be avoided. This shows claim 1 for the first two cases of the definition of *split* in Table 6.5. Part 2, i.e., (D.8) is verified by induction over the length of P_3.

– If the service of R_1 is not known. We have

$$\mathit{partial}(AB, R') \ = \ \lceil P_1 \rceil; \lceil P_2 \rceil; y \mapsto (R_1 R_2); y$$

from which conclusion 1 and 2 follow immediately. These are all cases and concludes our proof. □

An easy corollary of the above is that for all closed agents A it holds: $\mathit{partial}(A) \approx A$. This is due to the fact that for closed A we have $A \approx \epsilon; A$.

Appendix E

Core library abstractions

Channels. The following are channels initialized with a value or with an additional, non-destructive read service.

```
newReadChannel:
    'ch = newChannel()
    ch
    read:                                        # non-destructive read
        'r = ch.receive()
        ''ch.send r
        r

'initialized Factory X:                          # NB. curried
    'component = Factory()
    ''component.send X
    component

newInitChannel = initialized newChannel          # initialize channel with X
newInitReadChannel = initialized newReadChannel # initialize and read
```

Containers. The following service `newSlot` specifies a one-slot buffer. If the slot is empty calling `get` on it blocks, if the slot is full `put` blocks. The invariant is that exactly one value is written on either channel:

```
newSlot:
    'val = newChannel()
    'isEmpty = newInitChannel()
    get:
        val.receive()                            # block until non-empty
        ''isEmpty.send()
    put X:
        ''isEmpty.receive()                      # block until empty
        ''val.send X
```

Variables are channels that always contain one value. Observe that variables support operators to set and get the value.

```
newVar X:
    'var = newInitReadChannel X
    set X:
        ''var.receive()
        ''var.send X
        X
    get = var.read
    *_ = get
    _<-_ = set
```

Mapping. The following service maps every binding in a given form with a function.

```
buildValue:
    'slot = newInitReadChannel()
    slot.read                               # get current value
    extend V: slot.send(slot.receive(), V)  # extend current value

# requires a service X.f and a form X.form
map X:
    'value = buildValue()
    ''forEachLabel
        form = X.form
        do L: value.extend
            L.bind X.f(L.project form)
    value()
```

Bibliography

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings SIGSOFT 93, ACM Software Engineering Notes*, volume 18, pages 9–20, December 1993.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AC97] E. Agerbo and A. Cornils. Implementing GoF design patterns in BETA. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357, pages 92–95. Springer-Verlag, 1997.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[ACS96] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 147–162. Springer-Verlag, 1996.

[Ado90] Adobe Systems Incorporated. *PostScript Language Reference Manual*, 1990.

[AG94] R. Allen and D. Garlan. Formal connectors. CMU-CS-94-115, Carnegie Mellon University, March 1994.

[Agh86] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[AKH92] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29(8):737–760, December 1992.

[AKN00] F. Achermann, S. Kneubuehl, and O. Nierstrasz. Scripting coordination styles. In A. Porto and G.-C. Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[All97] R. J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[ALSN01] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola – a small composition language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[AMY97] K. Asai, H. Masuhara, and A. Yonezawa. Partial evaluation of call-by-value λ-calculus with side-effects. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–21, Amsterdam, the Netherlands, June 1997.

[AN00] F. Achermann and O. Nierstrasz. Explicit Namespaces. In J. Gutknecht and W. Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[AN01] F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[ASS91] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT electrical engineering and computer science series. McGraw-Hill, 1991.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.

[AWB$^+$94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1994.

[Bar84] H. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[BAW93] L. Bergmans, M. Aksit, and K. Wakita. An object-oriented model for extensible concurrent systems: The composition-filters approach. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BCTW96] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. Wise. A framework for event-based software integration. *IEEE Transactions on Software Engineering*, 5(4):378–421, October 1996.

[Bec97] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[Ben86] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[BFVY96] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.

[BG97] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, February 1997.

[BGW93] D. Bobrow, R. Gabriel, and J. White. Clos in context – the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. *Information and Computation*, 76(2/3), 1984. Also appeared in Proceedings of the International Symposium on Semantics of Data Types, Springer, LNCS (1984), and as SRC Research Report 1.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bos97] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, November 1997.

[Bos99] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, March 1999.

[Bou92] G. Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, 1992.

[Bou97] G. Boudol. The pi-calculus in direct style. In *Conference Record of POPL '97*, pages 228–241, 1997.

[BPS99] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 43–66, Lisbon, Portugal, June 1999. Springer-Verlag.

[Bra92] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[BW00] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 201–225. Springer-Verlag, 2000.

[Car93] L. Cardelli. Extensible records in a pure calculus of subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*, pages 373–425. MIT Press, 1993.

[CCM] Corba Components Package, Corba Components and Scripting. http://www.omg.org/technology/corba/corba3releaseinfo.htm.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of POPL '93*, pages 493–501. ACM, January 1993.

[CD99] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, 1999.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[CGKF01] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. In *Proceedings of Monterey Workshop*, 2001.

[CH98] M. Carlsson and T. Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1998.

[CIW99] D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *ACM Transactions on Software Engineering and Methodology*, 33(2):101–31, February 1999.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In V. Gruhn, editor, *ESEC'01*. ACM Press, 2001.

[Cle95] P. C. Clements. From subroutines to subsystems: Component-based software development. *American Programmer*, 8(11), 1995.

[CM93] L. Cardelli and J. C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*, pages 295–350. MIT Press, 1993.

[Cop99] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison Wesley, Reading, Mass., 1999.

[Dam94] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. Ph.D. thesis, University of Geneva, 1994.

[DF98] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. World Scientific, Singapore, 1998.

[DM98] W. De Meuter. Agora: The story of the simplest MOP in the world — or — the scheme of object–orientation. In J. Noble, I. Moore, and A. Taivalsaari, editors, *Prototype-based Programming*. Springer-Verlag, 1998.

[DMN70] O.-J. Dahl, B. Myrhaug, and K. Nygaard. (Simula67) Common Base Language. Technical Report N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, N, October 1970.

[Duc97] S. Ducasse. Message passing abstractions as elementary bricks for design pattern implementation. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology (ECOOP'97 Workshop Reader)*, volume 1357 of *LNCS*, pages 96–99. Springer-Verlag, June 1997.

[DW99] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1999.

[DZ99] S. Dal-Zilio. *Le calcul bleu: types et objects*. Ph.D. thesis, Université de Nice - Sophia Antipolis, July 1999. In french.

[EP93] S. Eisenbach and R. Paterson. Pi-calculus semantics of the concurrent configuration language darwin. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[Ern99] E. Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.

[Fai87] J. Fairbairn. Making form follow function: An exercise in functional programming style. *Software - Practice and Experience*, 17(6):379–386, 1987.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[FG98] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of ICALP '98*, pages 844–855, 1998.

[FGL$^+$96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, August 1996.

[FHJ96] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, Philadelphia, Pennsylvania, May 1996.

[FMQ96] G.-L. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, November 1996.

[Fou98] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. Ph.D. thesis, Ecole Polytechnique, 1998.

[FS97] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, October 1997.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GH98] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*. Elsevier ENTCS, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[GMW00] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, 2000.

[HC01] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering*. Addison Wesley, 2001.

[HHK95] M. Hansen, H. Hüttel, and J. Kleist. Bisimulations for asynchronous mobile processes. In I. Lee and S. A. Smolka, editors, *Proceedings of 6th International Conference on Concurrency Theory (CONCUR '95, Philadelphia)*, volume 962 of *LNCS*. Springer-Verlag, 1995.

[HK99] G. Hedin and J. L. Knudsen. Language support for application framework design. In M. E. Fayad, D.C.Schmidt, and R. Johnson, editors, *Implmenting Application Frameworks: Object-Oriented Frameworks at Work*. Wiley, 1999.

[HO93] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Höl93] U. Hölzle. Integrating independently-developed components in object-oriented languages. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, volume 707 of *LNCS*, pages 36–56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings ECOOP'91*, volume 512 of *LNCS*, pages 133–147, Geneva, Switzerland, July 15–19 1991. Springer-Verlag.

[HT92] K. Honda and M. Tokoro. On asynchronous communication semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 21–51. Springer-Verlag, 1992.

[Hud96] P. Hudak. Building domain specic embedded languages. *ACM Computing Surveys*, 28(4es), December 1996.

[Hud98] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[IKM$^+$97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, volume 21, November 1997.

[Ing86] D. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 347–349, November 1986.

[IW95] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JGS93] N. J. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JML98] S. P. Jones, E. Meijer, and D. Leijen. Scripting COM components in haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.

[Joh00] J. Johnson. *GUI Bloopers*. Morgan Kaufmann, 2000.

[KBH$^+$01] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Thread transparency in information flow middleware. Technical Report CSE-01-004, OGI, School of Science and Engineering, Oregon, 2001.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[KH97] S. N. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In *Proceedings of the Conference on Domain-Specific Languages*, pages 297–310, Berkeley, CA, USA, October 1997. USENIX.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceeding ECOOP'01*, 2001.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[KMF01] E. Kıcıman, L. Melloul, and A. Fox. Towards zero-code composition. Submitted to Hot Topics in Operating Systems (HotOS VIII)., 2001.

[KP84] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[KPT96] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Conference Record of POPL '96*, pages 358–371. ACM Press, January 1996.

[Kru97] D. J. Kruglinski. *Inside Visual C++*. Microsoft Press, 1997.

[LAN00] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G. Leavens and M. Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[Lau94] C. Lau. *Object-Oriented Programming Using SOM and DSOM*. Van Nostrand Reinhold, March 1994.

[Lea99] D. Lea. *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns*. The Java Series. Addison Wesley, 2nd edition, 1999.

[LHB01] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In J. Bosch, editor, *Proceedings GCSE'2001*, volume 2186 of *LNCS*. Springer-Verlag, 2001.

[LHJ95] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95*, pages 333–343, San Francisco, California, 1995.

[LKA$^+$95] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[Lon01] A. Longshaw. Choosing between COM+, EJB, and CCM. In *Component-Based Software Engeneering*, pages 621–640. Addison Wesley, 2001.

[LSLX94] K. J. Lieberherr, I. Silva-Lepe, and C. Xaio. Adaptive object-oriented programming using graph-based customizations. *Communications of the ACM*, 37(5):94–101, May 1994.

[Lum99] M. Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[Lut96] M. Lutz. *Programming Python*. O'Reilly & Associates, Inc., 1996.

[MB97] M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of TOOLS USA '97*, July 1997.

[McH94] C. McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[McI69] M. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–150. NATO Science Committee, January 1969.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.

[Mer00] M. Merro. *Locality in the π-calculus and applications to distributed object*. PhD thesis, Ecole de Mines de Paris, October 2000.

[Mét96] D. L. Métayer. Software architecture styles as graph grammars. In D. Garlan, editor, *SIGSOFT'96: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23. ACM Press, 1996.

[Mét98] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, July 1998.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MH00] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 2nd edition, 2000.

[Mil75] R. Milner. Processes, a mathematical model of computing agents. In *Logic Colloquium, Bristol 1973*, pages 157–174. North Holland, Amsterdam, 1975.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil91] R. Milner. The polyadic π calculus: a tutorial. ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil99] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[MK99] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[MKN00] M. Merro, J. Kleist, and U. Nestmann. Local π-calculus at work: Mobile objects as mobile processes. In *Proceedings of TCS 2000*, LNCS. Springer-Verlag, August 2000.

[MMPN93] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Reading, Mass., 1993.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89*, pages 14–23. IEEE Computer Society Press, Washington, DC, June 1989.

[Mor97] M. Morrison. *Presenting Java Beans*. Sams net, 1997.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and -86, Computer Science Dept., University of Edinburgh, March 1989.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings ICALP '92*, volume 623 of *LNCS*, pages 685–695, Vienna, July 1992. Springer-Verlag.

[MS98] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, July 1998.

[MSC99] A. K. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *LNCS*. Springer-Verlag, April 1999.

[MT97] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of ESEC/FSE'97*, pages 60–76, Zürich, Switzerland, September 1997.

[MU97] N. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In D. Garlan and D. L. Mètayer, editors, *Proceedings COORDINATION'97*, volume 1282 of *LNCS*, pages 81–97, Berlin, Germany, September 1997. Springer-Verlag.

[ND95] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

[Nis00] S. Nishizaki. Programmable enviroment calculus as theroy of dynamic software evolution. In *Proceedings ISPSE 2000*. IEEE Computer Society Press, 2000.

[NM95] O. Nierstrasz and T. D. Meijler. Requirements for a composition language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.

[NP96] U. Nestmann and B. C. Pierce. Decoding choice encodings. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *LNCS*, pages 179–194, Pisa, Italy, August 1996. Springer-Verlag.

[Ode95] M. Odersky. Applying π: Towards a basis for concurrent imperative programming. In *Proc. 2nd ACM SIGPLAN Workshop on State in Programming Languages*, January 1995.

[Ode00] M. Odersky. Functional nets. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 1–25. Springer-Verlag, March 2000.

[OKH$^+$95] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, pages 235–250, 1995.

[Ous98] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[Pal97] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Conference Record of POPL '97*, pages 256–265, Paris, France, January 1997.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[Pit80] K. Pitman. Special forms in lisp. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 179–197, August 1980.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

[Pre95] W. Pree. Framework development and reuse support. In M. M. Burnett, A. Goldberg, and T. G. Lewis, editors, *Visual Object-Oriented Programming*, pages 253–268. Manning Publishing Co., 1995.

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, October 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.

[PT00] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, May 2000.

[PV97] J. Parrow and B. Victor. The update calculus. In M. Johnson, editor, *Algebraic Methodology and Software Technology (Proceedings of AMAST '97)*, volume 1349 of *LNCS*, pages 409–423, Sydney, Australia, December 1997. Springer-Verlag.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RE94] M. Radestock and S. Eisenbach. What do you get from a pi-calculus semantics? In *Proceedings of Parallel Architectures and Languages Europe (PARLE '94)*, volume 817 of *LNCS*, pages 635–647. Springer-Verlag, 1994.

[Rém94] D. Rémy. *Typing Record Concatenation for Free*, chapter 10, pages 351–372. MIT Press, April 1994.

[Rep91] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, volume 26, pages 293–305, Toronto, June 1991.

[RJ97] D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

[Rog97] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[SA01] N. Schärli and F. Achermann. Partial evaluation of inter-language wrappers. In *Workshop on Composition Languages, WCL'01*, September 2001.

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[San93] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. Ph.D. thesis, Computer Science Dept., University of Edinburgh, May 1993.

[San00] D. Sangiorgi. Lazy functions and mobile processes. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, May 2000.

[San01] D. Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253, 2001.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered design with mixin layers. In E. Jul, editor, *Proceedings ECOOP'98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[SBMW99] N. Sample, D. Beringer, L. Melloul, and G. Wiederhold. CLAM: Composition language for autonomous megamodules. In P. Ciancarini and A. L. Wolf, editors, *Proceedings of Coordination'99*, volume 1594 of *LNCS*, pages 291–306, 1999.

[SC96] M. Shaw and P. Clements. Toward boxology: Preliminary classification of architectural styles. In *Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development*, pages 50–54, 1996.

[Sch99] J.-G. Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[Sch01] N. Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[SDK$^+$95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software and architecture and tools to support them. *IEEE Transactions on Software Engineering*, April 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Shi96] O. Shivers. A universal scripting framework or, Lambda: the ultimate little language. In J. Jaffer and R. Yap, editors, *Concurrency and Parallelism: Programming, Networking and Security*, pages 254–265. Springer-Verlag, 1996.

[SM92] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In W. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *LNCS*, pages 32–46. Springer-Verlag, 1992.

[SML99] L. Seiter, M. Mezini, and K. Lieberherr. Dynamic component gluing. In *Proc. First International Symposium on Generative and Component-Based Software Engineering, GCSE'99*, LNCS, 1999.

[Smo94] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proceedings of Constraints in Computational Logics*, volume 845 of *LNCS*, pages 50–72. Springer-Verlag, 1994. Available as Research Report RR-94-16 from DFKI Kaiserslautern.

[Smo95] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, Berlin, 1995.

[SN99] J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Sou95] J. Soukop. Implementing patterns. In J. Coplien and D.Schmidt, editors, *Pattern Languages of Program Design*, pages 395–412. Addison Wesley, 1995.

[Spi89] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[SPL98] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[SSB99] M. Sato, T. Sakurai, and R. M. Burstall. Explicit environments. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *LNCS*, pages 340–354, L'Aquila, Italy, April 1999. Springer-Verlag.

[SW01] D. Sangiorgi and D. Walker. *The Pi-Calculus - A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Szy98] C. A. Szyperski. *Component Software*. Addison Wesley, 1998.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119, Los Angeles CA, USA, 1999.

[Tra93] W. Tracz. Parameterized programming in LILEANNA. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pages 77–86, Indianapolis, IN, February 1993. ACM Press.

[Ude94] J. Udell. Componentware. *Byte*, 19(5):46–56, May 1994.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

[VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[Vli96] J. Vlissides. The hollywood principle. *C++ Report*, 8, February 1996.

[vLM96] M. van Limberghen and T. Mens. Encapsultation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS. Springer-Verlag, 1995.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, February 1995.

[WCO00] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 3rd edition, 2000.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, NY, 1990.

[Woj00] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge, March 2000.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, September 1999. Proceedings of ICFP'99.

Curriculum Vitae

Name	Franz Achermann
Birth Date	May 20, 1969
Nationality	Swiss

Education

1995	Diploma Computer Science, University of Berne
1989	Matura Typus B, Kollegium St. Fidelis, Stans

Work Experience

1997-2001	PhD. student and assistant, Institut für Informatik und angewandte Mathematik, University of Berne
1995-1996	Application Developer, CSC Ploenzke (Schweiz) AG
1989-1995	Student of Computer Science, University of Berne