A π-Calculus Based Approach for Software Composition

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Lumpe

von Deutschland

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

A π-Calculus Based Approach for Software Composition

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Lumpe

von Deutschland

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

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, den 21. Januar 1999

Der Dekan:
Prof. Dr. A. Pfiffner

Abstract

Present-day applications are increasingly required to be flexible, or "open" in a variety of ways. By flexibility we mean that these applications have to be *portable* (to different hardware and software platforms), *interoperable* (with other applications), *extendible* (to new functionality), *configurable* (to individual users' or clients' needs), and *maintainable*. These kinds of flexibility are currently best supported by component-oriented software technology: components, by means of abstraction, support portability, interoperability, and maintainability. Extendibility and configurability are supported by different forms of binding technology, or "glue": application parts, or even whole applications can be created by composing software components; applications stay flexible by allowing components to be replaced or reconfigured, possibly at runtime.

This thesis develops a formal language for software composition that is based on the π-calculus. More precisely, we present the $\pi\mathcal{L}$-calculus, a variant of the π-calculus in which agents communicate by passing extensible, labeled records, or so-called "forms", rather than tuples. This approach makes it much easier to model compositional abstractions than it is possible in the plain π-calculus, since the contents of communication are now independent of position, agents are more naturally polymorphic since communication forms can be easily extended, and environmental arguments can be passed implicitly.

The $\pi\mathcal{L}$-calculus is developed in three stages: (i) we analyse whether the π-calculus is suitable to model composition abstractions, (ii) driven by the insights we got using the π-calculus, we define a new calculus that has better support for software composition (e.g., provides support for inherently extensible software construction), and (iii), we define a first-order type system with subtype polymorphism and sound record concatenation that allows us to check statically an agent system in order to prevent the occurrences of run-time errors.

We conclude with defining a first Java-based composition system and Piccola, a prototype composition language based on the $\pi\mathcal{L}$-calculus. The composition system provides support for integrating arbitrary compositional abstractions using both Piccola and standard bridging technologies like RMI and CORBA. Furthermore, the composition systems maintains a *composition library* that provides components in a uniform way.

Contents

Chapter 1

Introduction

This thesis develops a formal language for software composition that is based on the π-calculus [65]. More precisely, we present the $\pi\mathcal{L}$-calculus which is an extension of the π-calculus, where we replace tuple communication by communication of so-called forms, a record-like data structure. The $\pi\mathcal{L}$-calculus is developed in three stages: (i) we analyse whether the π-calculus is suitable to model composition abstractions, (ii) driven by the insights we got using the π-calculus, we define a new calculus that has better support for software composition (e.g., provides support for inherently extensible software construction), and (iii), we define a first-order type system with subtype polymorphism that allows us to check statically an agent system in order to prevent the occurrences of runtime errors.

1.1 Background

One of the key challenges for programming language designers today is to provide the tools that will allow software engineers to develop robust, flexible, distributed applications from plug-compatible software components [76]. Current object-oriented programming languages typically provide an ad hoc collection of mechanisms for constructing and composing objects, and they are based on ad hoc semantic foundations (if any at all) [73]. A language for composing open systems, however, should be based on a rigorous semantic foundation in which concurrency, communication, abstraction, and composition are primitives.

The ad hoc nature of object-oriented languages can be manifested in three ways:

1. The granularity and nature of software abstractions may be restricted: the designer of a software component may be forced (unnaturally) to define it as an object. Useful abstractions may be finer (e.g., mixins) or coarser (e.g., modules) or even higher-order (e.g., a synchronization policy).

2. The abstraction mechanisms themselves may be ad hoc and inflexible: programmers typically have only limited facilities for defining which features are visible to which clients, how binding of features (static or dynamic) should be resolved, or what kinds of entities may be composed.

3. Language features are informally specified or even implementation dependent. Combinations of features may exhibit unpredictable behaviour in different implementations.

Given the ad hoc way in which software composition is supported in existing languages, we identify the need for a rigorous semantic foundation for modelling the composition of concurrent object systems from software components. Moreover, if we can understand all aspects of software components and their composition in terms of a small set of primitives, then we have a better hope of being able to cleanly integrate all required features in one unifying concept.

There are several plausible candidates as computational models for objects or compositional abstractions. The λ-calculus has the advantage of having a well-developed theoretical foundation and being well-suited for modelling encapsulation, composition and type issues [24], but has the disadvantage of saying nothing about concurrency or communication. Process calculi such as CCS [62] have been developed to address just these shortcomings. Early work in modelling concurrent objects [83, 84] has proven CCS to be an expressive modelling tool, except that dynamic creation and communication of new communication channels cannot be directly expressed and that abstractions over the process space cannot be expressed within CCS itself, but only at a higher level.

The π-calculus [65] addresses these shortcomings by allowing new names to be introduced and communicated much in the same way that the λ-calculus introduces new bound names. This is needed for modelling creation of new objects with their own unique object identifiers. The basic (monadic) calculus allows only communication of channel names. The polyadic π-calculus [64] supports communication of tuples, needed to model passing of complex messages. The higher-order π-calculus [100] supports the communication of process abstractions, which is needed for modelling software composition within the calculus itself. Interestingly, the polyadic and higher-order variants of the π-calculus can be faithfully translated (or "compiled") down to the basic calculus, so one may confidently use the features of richer variants of the calculus knowing that their meaning can always be understood in terms of the core calculus. The π-calculus has previously been used by Walker [120], Jones [47] and Barrio [9] to model various aspects of object-oriented languages. Moreover, Sangiorgi [103] presented an interpretation of Abadi and Cardelli's first-order functional *Object Calculus* [1] into a typed π-calculus with variant types.

A further simplification has been studied by Honda [43], who proposed that asynchronous communication provides a better foundation for distributed systems, without any loss of expressive power. Sangiorgi [101] worked also in this area and proposed

a so-called *polyadic mini π-calculus* that essentially forms the core language for PICT [89, 92, 112] that we have used as platform to model object-oriented and component-oriented abstractions.

In this thesis we show that the π-calculus can be used to model composition mechanisms. However, it is inconvenient for modeling general composition abstractions due to the dependence on positional parameters in communications. In the context of the λ-calculus Dami [28, 30] identified a similar problem. He proposed λN, a calculus in which parameters are identified by names rather than positions. Like Dami, we shall introduce an explicit naming scheme to address parameters by names resulting in the $\pi\mathcal{L}$-calculus, an offspring of the asynchronous π-calculus.

1.2 Contribution of the thesis

The contributions of the thesis can be summarized as follows:

- We show that common object-oriented programming abstractions such as dynamic binding, inheritance, genericity, and class variables are most easily modelled when metaobjects are explicitly reified as first class entities (i.e., processes). Furthermore, we show that various roles which are typically merged (or confused) in object-oriented languages such as classes, implementations, and metaobjects, each show up as strongly-typed, first class processes.

- Based on the idea of Dami [28], we define the polymorphic $\pi\mathcal{L}$-calculus, where the communication of tuples is replaced with the communication of labeled parameters. In fact, in the $\pi\mathcal{L}$-calculus parameters are identified by names rather than positions which provides a higher degree of flexibility and extensibility for software composition. We give a basic theory for the $\pi\mathcal{L}$-calculus and define a asynchronous bisimulation relation on $\pi\mathcal{L}$-agents.

- We introduce so-called *polymorphic form extension* (or polymorphic record concatenation) which is a essential feature for modelling higher-level compositional abstractions like classes and class inheritance. In fact, with polymorphic form extension we get a powerful mechanism for software composition, since it allows us to compose arbitrary services in order to achieve the required behaviour.

- We present a sound first-order type system for the $\pi\mathcal{L}$-calculus that incorporates asymmetric record concatenation. Moreover, we present a type inference algorithm that does automatic type reconstruction starting form a totally untyped program. To our knowledge, it is the first time that asymmetric record concatenation has been fully incorporated in both the type system and the type inference algorithm for it.

- We show that, unlike in Turner's approach [112], type unification in the presence of polymorphic form extension requires to consider the complete agent which, however, allow us to reuse Wand's type inference algorithm [121] to reconstruct the type annotations for $\pi\mathcal{L}$-agents.

- We present a Java-based composition system and PICCOLA, a prototype composition language. The kernel of the composition system is an abstract $\pi\mathcal{L}$-machine that is implemented in Java. In fact, the abstract $\pi\mathcal{L}$-machine implements exactly the $\pi\mathcal{L}$-calculus as defined in Chapter 4. Moreover, the composition system provides support for integrating arbitrary compositional abstractions using standard bridging technologies like RMI and CORBA and maintains a *composition library* that provides components in a uniform way.

1.3 Road map

The rest of the thesis is organized as follows. Chapter 2 gives a survey of component-oriented concepts. In Chapter 3, we show that the asynchronous polyadic π-calculus can be used to model compositional abstractions. In Chapter 4, we defined the $\pi\mathcal{L}$-calculus an extension of the π-calculus, where communication of tuples is replaced be communication of so-called forms. In Chapter 5, we introduce a typing scheme for the $\pi\mathcal{L}$-calculus. Furthermore, we present a type inference algorithm for the typed $\pi\mathcal{L}$-calculus. In Chapter 6, we present a composition system and the composition language PICCOLA. Finally, Chapter 7 presents conclusions and future directions of the development of the $\pi\mathcal{L}$-calculus and the composition language PICCOLA.

Chapter 2

Survey of Component-Oriented Concepts

Software component technology is still new and there is an ongoing discussion what component orientation is all about. The first question to be answered is – why do we need component software? Nierstrasz and Dami [74] argue that the composition of reusable and configurable software components will allow us to cope with evolving requirements by unplugging and reconfiguring only the affected parts. Following Szyperski [110], components are the way software technology has to go because all other engineering disciplines have introduced components as they became mature. Furthermore, software components and appropriate composition mechanisms provide the means for systematic software reuse [99, 106].

In order to describe "component orientation" we use the scheme defined by Nierstrasz and Dami [74], who distinguish methodological and technical aspects. At the methodological level, a component is a component because is has been *designed* to be composed with other software components. A component is normally not designed in isolation, but as an element of a *component framework* that provides (i) a library of black-box components, (ii) a reusable *software architecture* in which the components fit in, and (iii) some form of *glue* (connectors) that allows us to compose components.

At the technical level, the vision of component-oriented software development is indeed very old. Already in 1969 McIlroy [56] has proposed basic principles of component-oriented software construction. The basic idea of his proposal was that we should not think any more about which mechanism should we use but what mechanism should we build. He viewed components as families of routines which are constructed based on so-called rational principles so that these families fit together as building blocks. McIlroy stated that these families constitute components which are black-box entities.

Unfortunately, his vision could not be established at this time. There are several reasons for that like the idea that components should be build system independently or that a component catalogue must be available in order to allow application pro-

grammers to choose the right component for a specific problem. Interestingly, the way McIlroy defined how the concrete instances of components can be created is very similar to the recently proposed *open implementation approach* of Kiczales, et al. [49].

Composition enables prefabricated "things" to be reused by rearranging them in new composites [110]. But all the concepts in resuability are useless if there is no discipline in programming [94]. Systematic reuse requires a foundation of high-quality components with proper documentation [99]. Software is often not initially designed for being reused. Effective reuse depends not only on finding and reusing components, but also on the ways components are combined. Even components with appropriate functionality may fail to work together if composition breaks assumptions imposed by implicit styles (architectures) and packaging distinctions [106]. Developing reusable components also means that these components will usually deployed in other contexts than initially developed for. Therefore, it is important to find appropriate development mechanisms that allow one to evaluate, adapt, and integrate exiting or newly created components in new contexts. With "component orientation" software engineering moves to "component engineering".

Roughly spoken, components are for composition. Therefore, it is also important to distinguish *computational* and *compositional* aspects of software development. An application can be viewed as computational entity that produces some results, and as a composition of software artefacts (components) that fit together to achieve the required behaviour of the application.

In the composition process components are not only distinguished objects. Moreover, we have software abstractions like mixins, functions, interfaces, agents, protocols, procedures and modules as valid candidates for components, but only a few of them are instantiated directly as objects. We can assign each software abstraction the attribute compositional or computational. Compositional abstractions are mixins, interfaces and protocols. These abstractions are more or less organizational patterns that help to structure an application more conveniently. On the other side, functions, procedures and agents are computational abstractions having always a run-time representation. We use *classes* and *inheritance* to define such abstractions. Therefore, we can view "component orientation" as a higher level form of "object orientation".

It has to be noted that component approach needs a *critical mass* to take off [110]. This means, before we can efficiently use the component approach, we need a rich set of software artefacts (component libraries). Once such component libraries are available, we can apply reusability-metrics [94] to find the appropriate components. The development of new software is then driven by what is available to be reused. The component approach replaces revolution by evolution. But all this requires that software developer respect a discipline in programming – a component-oriented programming discipline.

2.1 What is a component?

A software component is an element of a component framework.

Although this definition appears to be circular, it captures an essential characteristic of components. Components of a stereo system, or of a modular furniture system are only components because they have been *designed* to be combined and composed with other components. A single component that does neither belong to a component system nor is composable in any way is a contradiction in terms. Furthermore, a component cannot function outside a defined framework.

A software component is a "static abstraction with plugs" [74].

A component itself is a software abstraction, i.e., it is (to a greater or lesser degree) a "black-box" that hides implementation details. Furthermore, a component must be instantiated to be used (this is imposed by the charaterization as "static abstraction"). In class based object-oriented programming languages we distinguish between classes and objects being instances of a specific class. Unfortunately, we do not have such a convenient way to distinguish between components and their instances. This is a frequent source of confusion. Talking about components we often mean their instances. The different roles of components and their instances are hidden by the way we develop component-oriented software. Composition is usally done within a composition environment. Inside such an environment the difference between components and their instances vanishes. The application programmer, even at design time, always works with instances.

A component has *plugs*, which are not only used to provide services, but also to require them. A true "black-box" component advertices all of its features and dependencies by means of public plugs. There are no hidden dependencies.

Plugs are the main prerequisite for composition. A plug is a *pluggable* interface, but what kind of interface it is exactly, and how these interfaces are *plugged* together, may depend from one component framework to another.

As an example, the declarative binding language *Darwin* [54] uses exactly this scheme to describe components and their interaction. With Drawin one can define hierarchies of compositions of interconnected components. The central abstractions used in Darwin are (i) components and (ii) services (means by which components interact). Components in Darwin are both context indenpendently implemented and tested. Darwin distinguishes *provided* and *required* services. The service access points are called plugs. Component interaction is established by connecting service outputs (provided plugs) with service inputs (required plugs). A composite filter component pipeline modelled in Darwin is shown in Fig. 2.1. The figure illustrates the composition of n filters. The output (denoted by a filled circle) of each filter is bound to the input of its successor. The input port of the first filter is bound to the input port of the

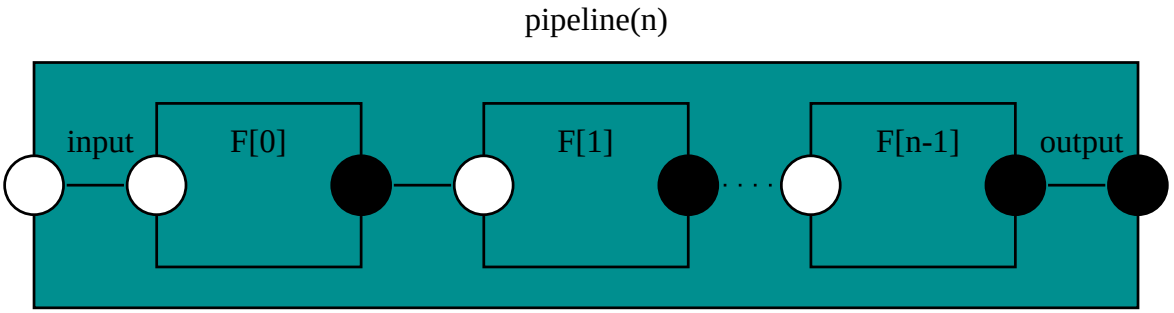

Figure 2.1: Composite component filter in Darwin.

pipeline and the output of the last filter is bound to the output of the pipeline. Darwin provides a *bind* statement to connect provided and required ports.

A component framework is a collection of software components with a software architecture that determines the interfaces that components may have and the rules governing their composition. [107]

This definition closes the loop. The essential point is that components are not used in isolation, but in context with a software architecture that determines how components are plugged together. The framework, the software architecture, and the rules of composition can vary quite dramatically. The critera may be, how fine-grained or coarse-grained, how rigorous or flexible, how domain-specific or general purpose these frameworks, architectures and rules are.

Szyperski [110] describes a component framework as a set of interfaces and rules of interactions that govern how components *plugged into* the framework may interact. He pointed out that an overgeneralization of that scheme has to be avoided in order to keep actual reuse practicable.

What is not covered by this a definition is "glue". The way we glue components depends on the programming model. It is sometimes claimed that programming changes into *scripting* if we have reached a certain level of abstractions. Recently, scripting models (e.g. Visual Basic, Python, and even Haskell [86]) have become very popular to build component based software.

If we look at real glue we discover it main property that it tends to be armorphous and we argue that software glue is armorphous as well. Some kinds of glue can only be used in particular situations to bind certain kinds of materials, whereas other kinds of glue can be used to create structures of objects that were never really designed to be composed. Software glue exposes a similar behaviour and can therefore serve a variety of needs.

Unfortunately, it is not easy to classify componentware is a meaningful way. There

are many different techniques, mechanisms and approaches that any scheme to compare component systems will emphasize some important attributes while ignoring others. Instead of classifying componentware we will consider components, frameworks and glue in turn, and try to identify common attributes, distinguishing attributes, trends and problems. In the end, we optain a picture that reflects the state-of-the-art in componentware.

2.2 Components

Existing component models can be characterized by a set of common attributes. Unfortunately, these attributes cover only partwise all variables a component may depend on. The number of attributes of component models is as high as the number of different kinds of component models. In fact, if we only consider the common attributes of components we have to be aware that the differences between two component models may still be huge.

We can ask: Are components fine grain or coarse grain? Are they source code or pre-compiled? Are they platform-specific/homogeneous or platform-independent/heterogeneous? Are they white-box or black-box entities? Are they stateful or stateless? Do components have standard interfaces?

2.2.1 Scale and Granularity of components

More specially, one can ask, are components bigger or smaller than objects, or are they the same size? Can they as big as whole applications?

First, a component is most useful if it offers the "right" set of interfaces and has no restricting context dependencies. For components to be deployed successfully in new contexts their granularity as well as their mutual dependencies have to be carefully controlled from the outset. Partitioning a design into components of the right weight is a subtle process that has large impact on the success of the resulting components.

Given our definition above, we argue that components can have arbitrary granularity. A good example for fine grain components are mixins [15, 14], which can be composed to define, to extend, or to specialize object classes to build a family of modified classes. For example, a mixin might be defined that add the capability to react on keystrokes to a graphic control. This mixin could be applied to any kind of graphic control to create a new graphic control that can receive input focus.

Another class of fine grain components are dialog elements like buttons or labels used in component based user interface systems. These components have a very limited functionality and they usually react only on a few events.

Components implemented obeying the Component Object Model (COM) [60] from Microsoft should also be fine grained. This model advises explicitly the application

programmer to design the interface of components as simple as possible [98]. The simpler the interface of a component the better it is composable. This corresponds closely to the fact that plug-ins like Apple's QuickTime are indeed also fine-grained components. The most popular plug-in architectures are those of modern WWW-browsers like Netscape.

Typically, fine-grain components are implemented as procedures or classes, and are used to build, or configure applications or parts of applications.

Coarse grain components, on the other side, can be as big as whole applications. Good examples are the *part editors* of OpenDoc [33] or embedding an Excel spreadsheet inside a Word document using OLE [16]. Unix programs like grep, awk or sed can also be considered as components. Using the shell pipe operator we can combine these filter programs to construct really amazing text manipulations under the condition that the all filters agree on a common representation convention like ASCII.

Coarse grain components are good candidates to build distributed applications where the components may run on different nodes. Client/Server applications are a typical example.

With DCOM (Distributed Component Object Model) [17] from Microsoft we can build applications using distributed components. DCOM is an extension of COM. DCOM supports multithreading as well as really distributed components. All mechanisms are transparent for the user. To build applications with COM or DCOM a lot of tools are available (e.g. MIDL Microsoft's own IDL compiler for COM). These tools help to automate several errorprone tasks like data conversion and networking.

The best size of a component depends on many different aspects like the level of abstraction, costs and benefits of deployment, component shipping and extension, component instantiation, and component loading. At least, the scale and granularity of components influence most their successful deployment. If we use the right architecture, component dependencies will be made explicit and the effort to control these dependencies will be reduced to a minimum. We always have to keep in mind that maximizing reusability leads to an explosion of context dependencies and therefore, it will minimize the use. Furthermore, granularity influences to an certain degree non-functional properties like performance and security.

2.2.2 Binary or source code components

Components tend to be pre-compiled since this is more consistent with the notion that components should be black-box entities. Usually, components are bundled up in component libraries (as proposed by McIlroy [56]).

Szyperski claims that software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system. He argues that insisting on a binary form of components is essential to allow an independent development of components and their robust intragration in new contexts even if

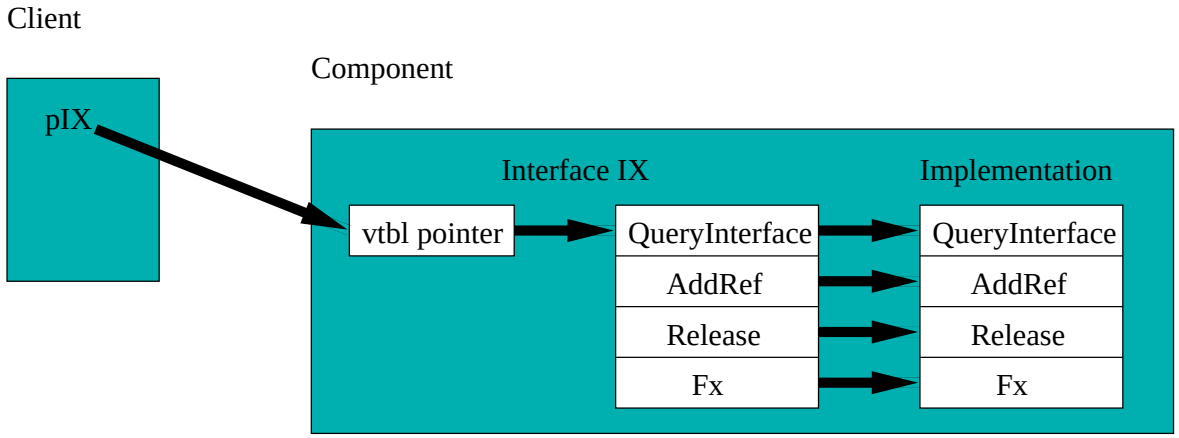

Figure 2.2: Binary representation of a COM interface.

this will rule out many useful software abstraction, such as type declarations, macros, and templates.

The most popular specification for binary components is Microsoft's COM. COM in its core is very simple. The sole fundamental entity that COM does define is an *interface*. On the binary level, an interface is represented as a pointer to a table containing pointers to the interface methods (Fig. 2.2). COM neither specifies how a particular programming language may be bound to it nor defines that one has to use a specific implementation technique.

Usually, COM components are bundled up in a DLL (dynamic link library) or in a standalone application (EXE). The DLL as well as the standalone application can host more than one component. The mechanisms to load a component are in both cases very similar, except that components loaded from a standalone application will run in a different address space while components loaded from a DLL will share the same address space with the loader of the component. The COM system is responsible for loading components, but the component developer has to equip the components with the necessary COM-interface methods like *CreateInstance*.

Further examples of systems that represent components in a more or less binary form are Delphi [12] and JavaBeans [109, 69]. Unfortunately, Delphi's component library is in its nature a static link library since used components have to be linked to the final application resulting in a monolithic and often huge executable. JavaBeans components, on the other side, are shipped within archive files (.jar) which are loaded at run-time and the system extracts the needed beans out of the archive. In both cases components may be implemented with one or more classes.

A glaring exception, nevertheless useful, is that of C++ templates, and particularly the Standard Template Library [70], which is a collection of container classes and algorithms implementing common data structures and procedures (e.g., lists, queues,

sorting etc.). The main reason STL is an exception is that C++ templates have been design in such a way that source code must be generated and compiled when templates are instantiated. Other languages (like Eiffel [58] an Ada [6, 8]) offer similar or equivalent *generic* components as pre-compiled, black-box classes.

2.2.3 Homogeneous or heterogeneous?

Fine grain components tend to be homogeneous, since these components are typically composed to form an application built within a single programming environment (such as Smalltalk [37], Oberon [95], Eiffel [58], Delphi [12], Java [7]). Furthermore, fine grain components tend to have procedural interfaces that are programming language dependent. Usually fine grain components provide a very simple configuration interface consisting only of one or two configurable event handlers. Note, for example, in COM there is no configuration interface available, components must be configured at runtime.

Typical examples for fine grain components are dialog elements like buttons and labels. They have a homogeneous structure since they have been built for a particular component system and they provide only a very simple configuration interface. Actions these components have to perform if they receive the corresponding event have to be specified by the application programmer. For example, in the case of Delphi or JavaBeans the same programming language is used for both programming the component and specifying the action code.

Coarse grain components more typically do not present procedural interfaces, but interfaces based on streams, events, RPC (remote procedural call), or other higher-level communication mechanisms. The "output stream" of a Unix program can be "piped" into a the "input stream" of another Unix program independently of the programming language used to implement them, as long as the second program can understand the byte stream produced by the first. It is important to note that it is often difficult to check the correctness of the filter combination in advance in the sense of a static type analysis. Two components may fail to work together due to different assumptions about the interpretation of the data stream. Shaw [106] calls this an *architectural style mismatch*.

Today, heterogeneous component approaches are less frequently used. Delphi, for example, is a system that offers support to integrate foreign components. The Delphi system generates for these components proxy-components that can be used like plain Delphi components while the foreign component itself is located outside the Delphi environment. The only difference in using foreign components is that they are usually not linked to the final application. In the case of Visual Basic controls such components have to be registered in a system database. This is often a source for problems then shipping an application means that one can not only put the application in the distribution, moreover, also all foreign components have to be redistributed (sometimes this imposes serious license problems).

DCOM can also be seen as a model that is open for heterogeneous components. Components may run on different nodes with an arbitrary architecture as long as DCOM is supported for this architecture. All data conversion, marshalling, and networking is done by DCOM.

The OMG [79] has defined the *Common Object Request Broker Architecture* (short CORBA) which enables to build software with a wide variety of languages, implementations, and platform. CORBA is not a binary standard, but everything is carefully standardized to allow individual vendors to profit using CORBA. Interoperability in CORBA is achieved by a three parts: a set of invocation interfaces, the object request broker (ORB), and a set of object adapters. To integrate a new language one has to define the appropriate bindings to OMG IDL. Such bindings exist for several languages, including C, C++, Smalltalk, and Java.

Clearly, whether components are homogeneous or heterogeneous depends on the problem domain. Although there is still a great interest in component technology for building standalone applications, there is increasing demand for component approaches that will (i) allow applications to talk to each other, and (ii) to combine services provides by diverse applications running on heterogeneous hardware and software platforms. For this reasons, industry standards like CORBA and COM will take on increasing importance. The reader should note as well that the specification of JavaBeans [109] defines explicitly that the bean model may be mapped transparently to any other existing component model.

2.2.4 "White-box" or "black-box" components?

We already stated that components are black-box entities, but clearly this is a relative concept. Even a human being needs a suitable environment to exists or to keep alive. Without the right equipment it is not possible to survive on the moon or at the ground of an ocean. Components behave in the same manner. Smalltalk object cannot be simply exported out of the Smalltalk environment. In order to support *component export* or *component exchange* from or between two component environments (frameworks), we need appropriate mechanisms that provide these kinds of operations.

Even if we bridge environments, we must be careful to fulfill all assumptions. Smalltalk and C++ components cannot work productively together if the Smalltalk garbage collector does not know which C++ objects have references to Smalltalk objects.

Even hidden assumptions are very often present in homogeneous environments. Objects and procedures may only work if environments have been properly initialized. Procedural interfaces are too weak to express these kinds of dependencies, and few component approaches go beyond this.

This problem gets more critical in the case of COM. In COM a proper initialization is less important. The real problem is the *reference counting* for interface pointers.

If an application does not correctly count the usages of an interface (call functions *AddRef* and *Release*) then it may happen that a component will never be released from memory, if a component is released too early from memory this may lead to system crashes. There exists no safe method to solve these kinds of problems. Some class libraries (MFC – Microsoft Foundation Library for C++) already provide some support to control the reference counting mechanism, but it is very limited and these problems can still occur (e.g., circular use of interfaces in two components). Peyton Jones, et al. [86] avoid reference counting in their solution. Instead of handing out the counting procedures to the programmer, Haskell's garbage collector will call *Release* when an interface has become inaccessible. This is a form of *finalization*.

The view that the user of a component is responsible that all necessary initializations are done and all conditions for using the component are satisfied is inadequate. What is needed here is an appropriate support of the component framework providing the right programming model. Using this model, the component programmer can focus on the development the component while the framework keeps track that all assumptions are fulfilled. The component framework has to guarantee a set of invariants such that a component user can focus on solving the problem rather than on satisfying all side conditions.

In fact, a software component must be a unit of composition with contractually specified interfaces and explicit context dependencies. Therefore, it is less important if components are black-box entities or not. The contractual specification of component interfaces guarantees that a component can be deployed independently in new contexts. An instantiation of a component will use the contract as a template to satisfy all assumptions and conditions made to use such a component in a given context. The client and the provider of a component may renegotiate further deployment conditions.

2.2.5 Stateful or stateless components

In section 2.1 we claimed that components are "static abstractions", hence stateless. In fact, in many environments, the distinction between components and their instances is blurred. Consider a user interface builder in which we are connecting buttons, text fields and labels to a form. Each of these "components" clearly has a state (a value, a position, a color, etc.).

The confusion arises because we are not really working with the abstract components (the button class, etc.) but with their instances. Builders tend to work with concrete instances, whereas programming languages deal explicitly with classes. However, even with builders there is typically a distinction between *design time* and *run-time*. Many component methods can only be used at design time while others can only be executed at run-time. Some component models define very precisely how a component and its methods have to work in both cases (e.g., Delphi components have to check its property *ComponentState* to determine whether an operation can be performed or not

```
constructor TAnyComponent.Create( Owner : TComponent );
begin
   { Test for design-mode before calling the inherited constructor. }
   if csDesigning in Owner.ComponentState then
      InitializeForDesignMode;

   { Calling the inherited constructor initializes Self.ComponentState. }
   inherited Create( Owner );

   { Test for design time. }
   if csDesigning in ComponentState then
      DesignTime
   else
      RunTime;

   ...
end;
```

Figure 2.3: Testing ComponentState within the component constructor in Delphi.

[51], see Fig. 2.3). Checking the state of the component is the way to prevent *contract violations*.

JavaBeans and Delphi components are the best candidates for stateful components. Using so-called *properties*, a component can be configured both at design time and at run-time. These systems use a special file to store the settings. This allows us to configure a component without writing any code (except event handlers). At run-time the configuration is restored loading the values out of the configuration file.

Modern programming environments provide a set of possibilities to configure components as simply as possible. So-called *property editors* are an example. Property editors are themselves components, but they differ from application components since they will not be linked to the final application. Therefore, a component does not only include code that defines its behaviour but also a set of surrounding code (procedures, components) that will only be used at design time.

Following Szyperski, we argue that all mainstream component software approaches have to introduce a notion of state. States are not only used to represent component properties like color, size, or position, but they can also be used to apply more sophisticated programming techniques like re-entrance. Recursive method calls or callbacks may break a contractual specification. With states it is possible to check some pre- and post-conditions under which a client may be granted access to a service of a component.

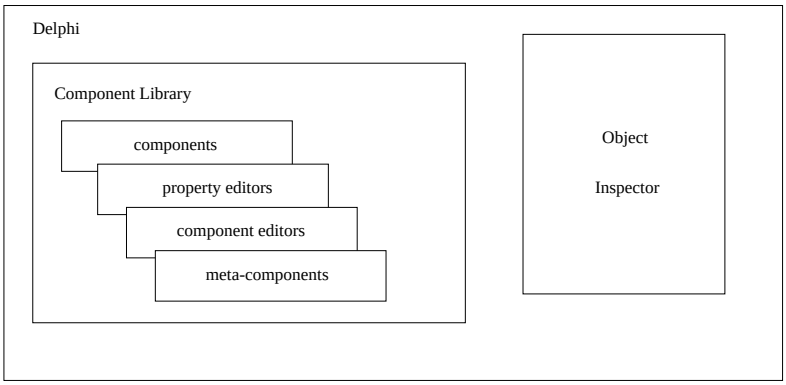

Figure 2.4: Meta-components in Delphi at design time.

2.2.6 Meta-components

The process of making information about a system available within that system is called *reification*. If the system makes use of this information in its normal course of execution it is said to be *reflective*. Using this information to manipulate the behaviour of that system is called *meta-programming*.

Only few component models provide services to support meta-programming. Java 1.1 has introduced the *Java Core Reflection Service* which extends the available runtime type information. This service, however, does not allow interception or manipulation at the meta-level. A similar approach is available for COM where a component can be equipped with a *type library* that support dynamic inspection of all interfaces revealed in the library.

As mentioned above, the Delphi system uses *property editors* to manipulate the state of components. In fact, these components are really *meta-components* (Fig. 2.4). Meta-components like meta-objects are used to affect the behaviour of other components and its environment [51, 98]. Usually, these meta-components serve no purpose at run-time and exist only at design time.

Although we can use meta-components in Delphi this concept is not explicitly mentioned in the system description. Delphi is implemented using Delphi. This makes it easy to modify the Delphi environment using user defined Delphi components. For example, it is possible to integrate a meta-component that controls the naming of components or one can build new inspector components (like the standard object inspector) [51]. This make Delphi besides other features an open system that allows one to integrate arbitrary extensions. To integrate new meta-components one only has to recompile the component library. Once this is done, all extensions are available immediately and these meta-components can take full advantage of the designer objects and run-time type information to reveal Delphi's internals.

2.2.7 Interface standards and standard interfaces

Interface standards are still an open issue in object-oriented as well as component-oriented software development and in connection with components this problem gets even a greater importance. Component connection needs to follow standards to make it at all likely that any two components have compatible *connectors*. A standard should specify just as much about the interfacing of certain components in order to allow as many clients and vendors to work together.

Now the question is, should the standards come before the product and markets, or vice versa? This question cannot simply be answered. In fact, there are examples for both cases. COM and JavaBeans have established a working market first while the OMG has defined a standard (CORBA, OMA) to build a market. We argue that the first way has some advantages especially because standards are usually derived form working applications. Future refinements of these standards are driven by experiences due to the deployment of applications obeying these standards. An example of this strategy is OLE [16], now being available in a second version.

In COM it is possible to remove a component from an application and replace it with another components at run-time. As long as the new component supports the same interfaces as the old component, the application still works. But it is not enough that the new component provides the same interfaces, moreover, the run-time behaviour has to be the same for this interfaces. Any additional behaviour that the new component provides must only be available using new interfaces not known by the original application.

In general, standards are useful to create common models that enable interoperation in principle. Furthermore, standards can be used to agree on concrete interface specifications, enabling effective composition. Finally, standards can also be used to impose overall architectures that assign components their place in composition and interoperation.

Unfortunately, only few systems support the notion of standard interfaces. COM components have standard interfaces. The COM specification requires that each published interface is frozen and never changed again. This guarantees that an application gets already the same functionality if it asks for a particular interface. COM uses so-called *globally unique identifiers* (GUI), a scheme invented by the Open Software Foundation to address a particular interface. Such an identifier is a one-to-one correspondence with an implementation. For this reason, if an application asks for an interface denoted by a specific GUI then the application will always get the same implementation. If there is no such implementation (e.g., the GUI is unknown in the system) then a null pointer is returned to the client.

That standard interfaces are indeed very important can be shown for *QuickReport*, a component package available for Delphi. The different versions of QuickReport are not compatible. Newer versions have a different sets of published properties and several

method and event specifications have changed too. The newer components cannot simply replace the older. What is missing is a scheme like in COM that forbids to change published interfaces of components.

2.2.8 Version management

In a component world, the number of versions of components that exists in parallel can be very high. Many vendors may provide so-called upward-compatible enhanced versions of a successful component. Unfortunately, traditional version management is driven by the assumption that a component evolves at a single source. But different vendors have different interpretations how a component has to provide its service even if it implements the same interface.

Only few component infrastructures address the component versioning problem properly. Newer versions of components must not violate the original specifications. Applications that use the old interfaces of a component will use also the old interfaces of a new component. Only newer application that have knowledge about new interfaces will use the newly provided interfaces and possibly the old. In fact, version management is closely related to the standardization of interfaces for components.

When is it necessary to build a new version? This question can be simply answered. If one of the following conditions is changed then one has to publish a new version:

- The number of functions in the interface has changed.
- The number of parameters in a function has changed.
- The types of parameters in a function have changed.
- The type of return value has changed.
- The meaning of functions and parameter has changed.

This list represents the more or less obvious reasons to release a new version. Some programming models impose additional conditions that are based on the implementation layout or the order of functions and parameters. We argue that such conditions break extensibility. We will use the view of Dami [28], who described extensibility as the possibility to add new functionality to an existing piece of code without affecting the previous behaviour. Therefore, in our opinion, a good component model will also be open to extensibility where the order or the implementation layout of functions and parameters is unimportant.

2.2.9 Typing

Ideally, all conditions of a contract should be stated explicitly and formally as part of an interface specification. Furthermore, it would be highly desirable to have tools to check automatically clients and providers against the contract specifications and in the case of a contract violation to reject the interoperation of both. The most component approaches, therefore, equip the interface specifications with type annotations. One argument for this decision is that only fully and explicitly typed interfaces can benefit from type checking. Furthermore, an independent development of both the client and the provider side is more or less impossible without appropriate type information.

In contrast, Darwin [54] does not require that interfaces be fully and explicitly typed. The Darwin tool infers the type of interface services (plugs) where an explicit typing is missing. Compatibility tests are platform dependent, and therefore, these tests are not part of the Darwin system.

If we use an explicit typing scheme, we open the component approach for substitutability and polymorphism. If two different components support the same interface, a client can use either of them without breaking the contractual specification. One component can substitute another. Therefore, a client can treat these different components polymorphically. Component models that support multiple interfaces encourage polymorphism. The more interfaces a component supports, the smaller these interfaces need to be. And the smaller an interface can be, the smaller its context dependencies will be resulting in a higher degree of reuse.

One component approach that supports polymorphism is COM. For example, one can write an application that implements a viewer that displays bitmaps. The bitmaps are implemented as COM components supporting an interface named *IDisplay*. The viewer interacts with the components only through their *IDisplay* interface. Now, suppose some requirements change and a viewer is needed that is able to display JPEG's. Insteed of writing a new version of the viewer, all we have to do is to implement a JPEG COM component that implements the *IDisplay* interface. The new component will substitute the old one, and the viewer is now able to display JPEG's. The viewer uses polymorphism to treat the different components in the same manner. It is important to note that this does not happen by accident. Equal COM interfaces are always treated uniformly even if this requires a careful planning in advance. Component development requires us to foresee further applications of an interface. Component development emphasizes extensibility.

2.3 Frameworks

A *framework*, in the most general sense, is a structure or a skeleton for a project. An object-oriented framework is a collection of cooperating classes, some of which may be

abstract, that defines the skeleton – and hence the architecture – of an application. A component framework defines the architecture of an application that can be completed by instantiating software components.

Frameworks need not necessarily be domain specific, but they are usually concept specific. For example, a framework for OpenDoc parts does not say much about the specific functions of parts, but embodies the concepts that make a piece of software an OpenDoc part.

Frameworks give components their meaning. In other words, a component without a framework is not a component at all. For example, consider user interface components. Their value lies not in the functionality offered be any individual component, but in the fact that they have been designed to work together in a consistent way to produce rich user interfaces.

Not all frameworks are component-oriented. Object-oriented frameworks fully concentrate on classes and inheritance. The programming model imposed by such frameworks is typically a form of white-box reuse; the application programmer instantiating an application from a framework often must be intimately familiar with implementation details of framework classes in order to define meaningful subclasses. Object composition, on the other hand, if based on forwarding is a concept based on black-box reuse. Frameworks that support object composition are accordingly called black-box frameworks. Examples are COM [60], Visual Basic [59], Delphi [12] and JavaBeans[109].

Modern development environments emphasize *visual programming*. These environments use mainly so-called *forms* for application development (e.g., Visual Basic, Delphi). A form itself is a component. The application programmer builds an application using the *Drag and Drop* method to place components out of the component palette (component library) within the form component. In the most general sense, this is (visual) composition. Inheritance is still present in these system, but in contrast to classical object-oriented frameworks inheritance is merely used to maintain the actual developed form. If one adds a new form to an application, the system will derive its skeleton from a general component base class that provides the necessary infrastructure for the newly created form component. In fact, the only situation where the application programmer gets in contact with inheritance is the specification of event handlers. Event handlers have to have access to the protected features of the component class. Fortunately, using the visual programming style one can ignore the fact that inheritance is used to construct a new form. One programs more or less in a black-box manner.

The most general form of black-box composition is defined by COM. COM emphasizes the specification of interfaces. Reuse by inheritance does not exist by default. In order to reuse existing code, composition has to be used, with the exception that one can use inheritance to define interfaces (as for definition of Java interfaces [7]).

COM supports two kinds of composition: *containment* and *aggregation*. Containment means that an outer component maintains a pointer to an inner component and

can therefore passes services provided by the inner component in a controlled way (the outer component may guard or wrap the inner component's services). In the case of aggregation the outer component maintains also a pointer to the inner component, but the interfaces of the inner component are passed directly to clients. The outer component does not have any control to the inner component's services.

We can identify several different forms of software composition. Which forms are supported depends on the component framework. Nierstrasz and Dami [74] define software composition as "the process of constructing applications by interconnecting software components through their plugs". In other words, composition is achieved by enabling a communication between components through their composition interfaces [57] (this view is more or less similar to that used in Darwin [54]).

Sametinger [99] identifies two basic forms of composition – *internal composition* and *external composition*. Internal composition denotes the process to compile and to link source code to a system. This form is mainly used in Delphi. Textual composition (instantiation of macros and templates) counts also to this form of composition. External composition is in our opinion the most natural form of software composition. Components act independently, i.e., run on their own. Components may communicate by means of interprocess communication (e.g., remote procedure calls). In fact, components may run in the same process space as well as in a different one. External composition is by such systems like Darwin, Visual Basic, and JavaBeans.

There exits many finer grained classifications of software composition (e.g., *functional composition*, *blackboard composition* [74] and *binding of communication channels* [77]), but we argue that these forms can be regarded as concrete applications either of internal or external composition. It is also possible that both forms coexist in the same system (e.g., Delphi's wrapping of Visual Basic components).

2.4 Glue

Naturally it is not enough to have components and frameworks, but one needs a way to bind components together. The binding technology, or *glue*, can take various forms, depending on the nature and granularity of the components [27]. Composition, or gluing, may occur at design time, link-time or run-time. Glue may be very rigid and static (like the syntactic expansion that occurs when C++ templates are composed), or very flexible and dynamic (like that supported by Tcl [123] and other scripting approaches). In fact, in software composition or software assembly the developer's role is to provide a small amount of "glue" that establishes the component interaction. The need to implement much additional code becomes obsolete.

What is the right programming model for gluing? We argue that scripting approaches like Perl, Python or JavaScript are the right candidates. In fact, scripting is not new. In the UNIX environment this is a de facto standard to develop small

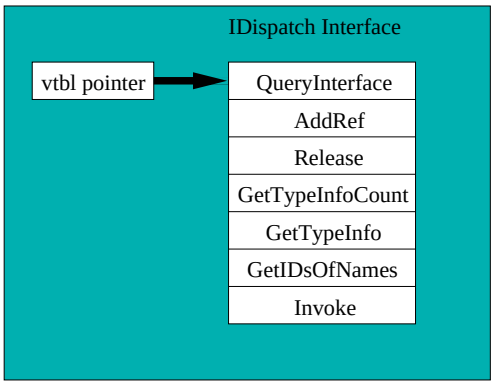

Figure 2.5: The *IDispatch* interface.

applications by combining command line tools gluing their standard input and output streams. Small scripts act as adaptors to bridge the gap between the single tools.

Scripting is a higher-level form of programming. In traditional programming languages elements like lists and dictionaries are missing while modern scripting languages provide them by default. Programming becomes easier and more effective. Programming an efficient string analysis is much simpler in Perl than in C, because Perl is trimmed for string operations. On the other side, however, not all scripting approaches provide the same functionality or emphasize the same aspects of programming. Python is a so-called general purpose scripting language that has also support for string manipulation, but this is not the main objective such that string manipulation in Python is more or less clumsy.

The most popular scripting language for personal computers is Visual Basic. In fact, one can consider Visual Basic as the first component-oriented programming system for Windows. Visual Basic uses so-called Visual Basic controls. These controls use *Automation* [98] a mechanism built on top of COM. Visual Basic controls are *Automation servers* that implement an *IDispatch* interface. A client can communicate with a control through its *IDispatch* interface (see Fig. 2.5). What makes the *IDispatch* interface interesting is the fact that it implements one single standard interface that can be used to ask for a specific interface the control implements. The *IDispatch* interface can be seen as a meta-interface that supports a kind of reflection by which Visual Basic controls become scriptable.

Visual Basic uses a dynamic typing scheme. The system does not have to know what arguments or result type a component method has before a program is allowed to call this method. A so-called *VARIANT* structure is used to make this possible. This structure encompasses all supported application types. Fortunately, the programmer does not get in touch with this structure. Furthermore, a Visual Basic programmer uses a programming paradigm that is more like that used in Smalltalk. Both the *IDispatch*

interface and the *VARIANT* structure make a control scriptable and Visual Basic a scripting language.

Is there a standard glue? There is an ongoing discussion if DCOM or CORBA should become the standard for gluing distributed objects. CORBA was supposed to be a heterogeneous glue, but due to the fact that different ORBs were not able to talk to each other CORBA could not be used as glue at all. Today this situation has changed, but it is left open whether CORBA can play the role of glue or not. It is more likely that we will have some additional layers of middleware that will enable to use different glue technologies. In the case of DCOM it holds more or less the same because it is only implemented for one platform.

The main problem concerning component-oriented software development is that there is no general accepted definition of scripting of composition or a definition for standard glue. This results mainly in the fact that most developer still use traditional techniques to implement their systems.

2.5 Open problems

Despite predictions to the contrary, a true "component market" has not really emerged since the late eighties [27]. Software components are not bought and sold like light bulbs and compact disks. This is partly because standards for component interfaces are only now taking hold, and partly because we have not yet found good economic models for making money from developing components. For example, if we want to produce 100 telephones we have to buy all necessary parts, each 100 times, before we can assemble the devices. On the other hand, if we develop software using the component approach we have to buy the needed components usually once even if we want to sell our solution 100 times. This is possible because we can simply copy (duplicate) the components as often as needed. Other engineering disciplines do not allow this. From the application programmer's point of view there is no drawback, but for the component developer this is very unlikely especially because he gets only payed once even if his product is used multiple times.

To further complicate matters, it is not always in the best interest of developers to sell their components: a component framework may be a strategic asset for a company that needs to be able to rapidly adapt products to changing customer needs [38].

Although a well-designed component framework, together with associated tools can dramatically improve productivity in both development and maintenance, the task of developing frameworks in the first place is somewhat a black art. One difficulty is that most mature software engineering methods focus on how to develop applications fulfilling specific requirements, not techniques for developing generic and reusable components. The situation is slowly improving, mainly because we are getting better experience at building frameworks, and new component sets tend to build on and improve

on existing approaches.

Most of the first "builder" tools focussed on user interface construction, mainly because it is natural to use a direct manipulation interface to instantiate, layout and connect such components. The builder paradigm, however, also applies to other domains, and JavaBeans [109] goes in this direction. What is not clear is how far direct manipulation can be pushed for general purpose programming tasks. Certainly the paradigm of wiring components together works well for some programming-in-the-small tasks, but quickly leads to incomprehensible "screen spaghetti" for complex tasks. It is also true that visual programming is intuitive for certainly kinds of simple tasks, but is more clumsy than text for describing non-trivial algorithms. A combination of a visual building together with a programming language or a scripting language, as offered by Visual Basic, Delphi or JavaBeans seems necessary for a general-purpose approach.

Another serious problem is that we have no good ways to talk about software architectures (or very few good ways [107]). We have seen that an essential ingredient of a component framework is an architecture that supports component composition. But architectures are notoriously hard to describe in a precise way. When we look at the source code of an application, it is easy to see the individual procedures and classes, but where is the architecture? (It is hard to see the forest for the trees.) Until we find simple ways to describe architectures, it will continue to be hard to learn how to use components effectively.

What becomes more important than ever is a precise notion of *contract specification* for components. Current specification techniques emphasize functional properties and leave non-functional properties more or less undefined. In order to deploy an imdependently developed component successfully in a new context all provided and required services must be known in advance. For example, we need to know re-entrance conditions, both the sequential and the concurrent case must be covered, or which resources are needed by a component. Current component approaches cover only selected aspects of contractual specifications for interfaces. At least all actual component approaches ignore non-functional properties. Unfortunately, it is not yet clear how one can represent such properties efficiently.

The component approach does not necessarily need to be object-oriented. For examples, to develop a COM component we can simply use C. There is no need to apply any object-oriented technique. But using an object-oriented technique can often simplify the task. We argue that the object-oriented technique is the basis and the component approach is its extension – objects and components have their designated role and a component system must provide abstractions for them appropriately. Furthermore, if we want to model mobility, objects can provide a suitable abstraction.

2.6 Conclusion

For the development of present-day applications, programming languages supporting higher order abstractions are needed. We argue that these higher order abstractions will be components. Unfortunately, most of the currently available programming languages and systems fail to provide sufficient support for specifying and implementing components.

Object-oriented programming addresses some of the needs of present-day applications, but offers only limited support for viewing applications as configurations of adaptable and reusable software components.

Components are self-contained configurable entities which can be composed to build an application. Unfortunately, most object-oriented techniques fail to provide suitable abstractions for general component specification and component composition (composition mechanisms) [76]. In order to get a system for composition, where components can be specified and implemented, but also components written in other languages/systems can be used, we need a new language – a composition language. This language will combine concepts and paradigms of existing languages and systems and will provide means for an abstract model for software composition [75, 76]. Furthermore, since there is a great demand that modern application development also means that new applications must be able to run in distributed environments, the model for software composition must also support the definition and use of concurrent and distributed components.

Components cannot be used in isolation, but according to a software architecture that determines how components are plugged together. Now, when do we call a software development environment a *composition environment*? We argue, a composition environment must be built of three parts: i) a reusable component library, ii) a component framework determining the software architecture, and iii) an open and flexible composition language. Most of the effort in component technology was spent on the first two parts. Now it is crucial to find an appropriate model to compose existing components together. In our opinion the π-calculus or the $\pi\mathcal{L}$-calculus which is presented in this thesis is a powerful system in order to fill the missing gap in an elegant and natural way.

Nierstrasz et al. [75] have shown that objects provide an organizational paradigm for *decomposing* large applications into coorperating objects as a reuse paradigm for *composing* applications from pre-packaged software components. Furthermore, they identified the crucial fact that object-oriented mechanisms for composition and reuse must be cleanly integrated with other features, such as concurrency, persistence and distribution.

In their argumentation, Nierstrasz et al. [75] focus solely on reuse. On the other side, Dami [28] pointed out that *extensibility* is another important aspect for software composition. In general, "plug-compatibility" usually imposes a certain discipline for

the use of the component plugs (e.g., plug arguments are identified by their position and have therefore always be specified in the same order).

Summarizing the properties of existing component approaches and the requirements discussed above, we can say that a general-purpose composition language must be formal to support reasoning about component configurations. In particular, a composition language should support the following features (Nierstrasz and Meijler [76] defined a similar set of requirements), each of which benefit from formal semantics: *active objects*, *components*, *glue*, a *formal object model*, and *reflection support*.

Chapter 3

Using the π-calculus to model compositional abstractions

In this section we present, how the π-calculus be used to model compositional abstraction. This work is based on experiment we made to model objects in Pict [52, 104].

Pict is an experimental programming language whose language features are all defined by syntactic transformation to a core language that implements the asynchronous simplified π-calculus. Pict is as much an attempt to turn the π-calculus into a full-blown programming language as it is a platform for experimenting with modelling of language features [91] and a platform for experimenting with type disciplines and type inference schemes for the π-calculus [90]. In Appendix A we present the syntax and semantics of the most important elements of Pict that we shall use to model compositional abstractions.

3.1 Towards an object model

What is the right kind of object model for software composition? What are the necessary features such a object model must provide? Where is the barrier between object model and more sophisticated compositional abstractions?

To give an answer to these questions is not easy. Furthermore, current object-oriented languages typically provide an ad hoc collection of mechanisms for constructing and composing objects, and they are based on ad hoc semantic foundations (if any at all) [73]. A language for composing open systems, however, should be based on a rigorous semantic foundation in which concurrency, communication, abstraction, and composition are primitive.

The ad hoc nature of object-oriented languages can be manifested in three ways:

1. The granularity and nature of software abstractions may be restricted: the designer of a software component may be forced (unnaturally) to define it as an

object. Useful abstractions may be finer (e.g., mixins) or coarser (e.g., modules) or even higher-order (e.g., a synchronization policy).

2. The abstraction mechanisms themselves may be ad hoc and inflexible: programmers typically have only limited facilities for defining which features are visible to which clients, how binding of features (static or dynamic) should be resolved, or what kinds of entities may be composed.

3. Language features are informally specified or even implementation dependent. Combinations of features may exhibit unpredictable behaviour in different implementations.

Given the ad hoc way in which software composition is supported in existing languages, we identify the need for a rigorous semantic foundation for modelling the composition of concurrent object systems from software components. What we also need to achieve is simplicity and unification of concepts: if we can understand all aspects of our object model in terms of a small set of primitives, then we are able to cleanly integrate these features and avoid semantic interference [73].

As a first step towards the definition of a compositional object model, we need to identify some basic properties of the object model that we believe are essential also for software composition. These basic features are strongly influenced by the object models of C++ [108], Java [7], Eiffel [58], and Object Pascal [12]. The basic model has to include the following features: *instance variables*, *instance methods*, *class variables*, *class methods*, *self-reference of objects*, *inheritance*, *genericity* and *polymorphism*, and *static* and *dynamic binding*.

At a higher level of abstraction we can identify additional features that we consider as basic compositional abstractions like *synchronization policies*, *mixins*, and some sort of reflection support or more precisely *meta-objects* and *meta-object protocols*.

We will not introduce visibility specifiers as used in the most object-oriented languages. Moreover, we will use some standard rules that will assign each feature a default visibility. Instance variables are private by default and instance methods are public. A subclass will inherit all public features of its superclass (protected inheritance is treated as a special case of public inheritance). A subclass can redefine any of its inherited features under the condition that this redefinition obeys the subtying rules of the implementation language Pict. We will not model multiple inheritance, although, it would be possible to add this mechanism. But due to the known problems with multiple inheritance [108] and due to the fact that we want to avoid inheritance in the software composition process we restrict ourselves to single inheritance.

We start our examination with an implementation of a procedural stack class in Scheme (Figure 3.1) of Friedman et al. [35] that illustrates the principle structure of a core programming model which supports objects and classes. This implementation

```
(define makeStack
   (let ((Pushed 0))
      (lambda ()
         (let ((Stk '()) (LocalPushed 0))
            (lambda (Message)
               (case Message
                  ((empty?) (lambda () (null? Stk)))
                  ((push!) (lambda (Value)
                           (set! Pushed (+ Pushed 1))
                           (set! LocalPushed (+ LocalPushed 1))
                           (set! Stk (cons Value Stk))))
                  ((pop!) (lambda ()
                           (if (null? Stk)
                              (error "Stack: Underflow")
                              (begin
                                 (set! Pushed (- Pushed 1))
                                 (set! LocalPushed (- LocalPushed 1))
                              (set! Stk (cdr Stk))))))
                  ((top) (lambda ()
                           (if (null? Stk)
                              (error "Stack: Underflow")
                              (car Stk))))
                  ((localPushed) (lambda () LocalPushed))
                  ((pushed) (lambda () Pushed))
                  (else (error "Stack: Message not understood" Message))))))))
```

Figure 3.1: Procedural stack class in Scheme.

uses the fact that in languages with first class functions (or procedures), it is possible to represent objects as functions or procedures.

The procedure `makeStack` implements a generator for the class `stack`. Each time this procedure is invoked it returns a new instance of the class `stack` – a stack object. This implementation provides two basic abstractions: class and object and uses the simplest approach to implement a *message passing* protocol: a procedure (anonymous lambda abstraction) representing an object is passed a message that selects the operation to be performed on the object.

Each instance and class may have its own state, which is maintained as the bindings of one or more *instance variables* or *class variables*. In Figure 3.1, there are two instance variables, `Stk` and `LocalPushed`, and one class variable, `Pushed`. Class variables maintain usually information that is common to all instances of a class, whereas each instance of a class has its own value for every instance variable. The reader should note

```
class stack = {
    private common
        Pushed : Integer = 0;
    private
        LocalPushed : Integer = 0;
        Stk : List of Integer = nil;
    public
        function empty() : Boolean { return null(Stk); }
        procedure push( Value : Integer )
            { Pushed = Pushed + 1; LocalPushed = LocalPushed + 1; cons( Value, Stk ); }
        procedure pop()
            { if empty() then raise( Underflow, "pop on empty stack" );
              else
                  Pushed = Pushed - 1; LocalPushed = LocalPushed - 1; cdr( Stk );
              end; }
        function top()
            { if empty() then raise( Underflow, "top on empty stack" );
              else
                  return car( Stk );
              end; }
        function localPushed() { return LocalPushed; }
    public common
        function pushed() { return Pushed; }
}
```

Figure 3.2: Class **stack**.

that class variables or class methods have proven their usefulness and often ease the implementation of certain problems, e.g. the singleton design pattern [36, 19]. This is at least one justification to incorporate these features in our component object model.

The following transcript illustrates the use of the procedure `makeStack` and objects located at `s1` and `s2`.

```
>  (define s1 (makeStack))
>  (define s2 (makeStack))
>  ((s1 'push!) 45)
>  ((s2 'push!) 2)
>  ((s2 'push!) 38)
>  ((s1 'localPushed!))
1
```

```
> ((s2 'localPushed!))
 2
> ((s1 'pushed!))
 3
> ((s2 'pushed!))
 3
> ((s2 'top!))
38
```

Unfortunately, the Scheme code in Figure 3.1 is not very readable. In Figure 3.2, the definition of class `stack` is shown in the pseudo code notation. The pseudo code notation that is based on the languages C++ and Java. The class `stack` has the same functionality as the procedure `makeStack` except that the function `pushed` is now a class method which respects more naturally its semantics.

Before we translate the functional model for class `stack` into PICT, we will study the translation of the λ-calculus into the polyadic mini π-calculus. This translation will provide us the necessary information in order to define our first object model in π and PICT respectively.

3.2 Function as processes

In this section, we will informally introduce the *polyadic mini π-calculus* [101] that is the basis for our object encoding and we present the encoding of two λ-calculus reduction strategies into the mini π-calculus. These encodings are based of the work of Milner [63], Pierce and Sangiorgi [87], Ostheimer and Davie [80], and Turner [112]. The aim of this presentation is that it will provide the necessary information to translate a λ-calculus based object model into π.

3.2.1 The polyadic mini π-calculus

The polyadic mini π-calculus is built from the operators of inaction, input prefix, output, parallel composition, restriction, and replication. Small letters $a, b, ..., x, y, ...$ range over the infinite set of names, and $P, Q, R, ...$ over the set of processes:

$$P ::= \mathbf{0} \;\Big|\; a(\tilde{x}).P \;\Big|\; \overline{a}\langle\tilde{x}\rangle \;\Big|\; P_1|P_2 \;\Big|\; (\nu\ a)P \;\Big|\; !a(\tilde{x}).P$$

$\mathbf{0}$ is the inactive process. An input-prefixed process $a(\tilde{x}).P$, where $\tilde{x}$ has pairwise distinct components, waits for a tuple of names $\tilde{y}$ to be sent along a and then behaves like $P\{\tilde{y}/\tilde{x}\}$, where $\{\tilde{y}/\tilde{x}\}$ is the simultaneous substitution of names $\tilde{x}$ with names $\tilde{y}$. An output $\overline{a}\langle\tilde{x}\rangle$ emits names $\tilde{x}$ at a. Parallel composition runs two processes in parallel. The restriction $(\nu\ a)P$ makes name a local to P. A replication $!a(\tilde{x}).P$ stands for a countably infinite number of copies of $a(\tilde{x}).P$ in parallel. In our object encodings we

will often use the special name _ as wildcard symbol. Values bound to this name are unimportant for the following process and will be ignored.

The set of *free names* $\mathrm{fn}(P)$ and the set of *bound names* $\mathrm{bn}(P)$ of a process P are defined in the usual way. The binding operators are the input prefix $a(\tilde{x})$ (which binds $\tilde{x}$) and the restriction $(\nu\ x)$.

The semantics of the polyadic mini π-calculus is presented using a reduction relation. This style of semantics involves defining two relations on processes: a reduction relation, specifying the actual communication behaviour of processes, and a structural congruence relation. The structural congruence relation will allows us to rewrite a process so that the participants of a potential communication can be syntactically juxtaposed.

The reduction relations, given below, describe the reduction of polyadic mini π-terms. The first two rules state that we can reduce under both parallel composition and restriction. The symmetric versions of both rules are redundant, because of the use of structural congruence. The communication rule takes two processes which are willing to communicate along channel a and simultaneously substitutes the free names $\tilde{x}$ with names $\tilde{y}$. The restricted names $\tilde{z}$ may be communicated from process Q to process P (scope extrusion). The communication rule is the only rule which reduces directly a π-term.

The communication rule assumes that processes are in a particular format. The structural congruence rule allows us to rewrite processes such that they have the correct format for the communication rule.

$$\text{PAR}: \ \frac{Q \ \longrightarrow \ R}{P \mid Q \ \longrightarrow \ P \mid R} \qquad \text{RES}: \ \frac{P \ \longrightarrow \ Q}{(\nu\ x)P \ \longrightarrow \ (\nu\ x)Q}$$

$$\text{COM}: \ a(\tilde{x}).P \mid \overline{a}\langle\tilde{y}\rangle \longrightarrow P\{\tilde{y}/\tilde{x}\}$$

$$\text{STRUCT}: \ \frac{P \ \equiv \ P' \qquad P' \ \longrightarrow \ Q' \qquad Q' \ \equiv \ Q}{P \ \longrightarrow \ Q}$$

The structural congruence relation is the smallest congruence relation over processes that satisfies the axioms below:

$$!a(\tilde{x}).P \ \equiv \ a(\tilde{x}).P \mid !a(\tilde{x}).P$$

$$P \mid Q \ \equiv \ Q \mid P \qquad\qquad P \mid \mathbf{0} \ \equiv \ P$$

$$(P \mid Q) \mid R \ \equiv \ P\ (Q \mid R)$$

$$(\nu\ x)P \mid Q \ \equiv \ (\nu\ x)(P \mid Q) \quad x \notin \mathrm{fn}(Q)$$

The replication operator enables processes to have an infinite behaviour. A replicated process can be called arbitrarily often by providing an arbitrary number of copies of it. Such a process acts as a server and its structure is of so general nature that it is helpful to have a higher-level syntax for it. A similar derived form is also a basic element in PICT.

$$\textbf{def } X[\tilde{x}] = P \textbf{ in } Q \stackrel{def}{=} (\nu\ X)(!X(\tilde{x}).P \mid Q)$$

In [64] Milner has demonstrated how data structures could be encoded in the π-calculus. For example, we can define one encoding of booleans as follows:

$$\textbf{def } \mathit{True}[r] = (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{t})$$
$$\textbf{def } \mathit{False}[r] = (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{f})$$

A boolean value is a channel along we send/receive two channels for the next *true* and *false* interaction. Both *True* and *False* do not take any parameter, other than a result channel r. They both create a new channel b that serves as the location of the boolean value and return b along the result channel r. Furthermore, since *True* and *False* are replicated processes they can answer queries about a boolean value b more than once. If we had omitted the replication the resulting processes would yield *linear* booleans.

Now, we will study the encoding of the λ-calculus in the π-calculus. The main reason for this is that the better we understand the encoding of objects and classes in terms of the λ-calculus (see Fig. 3.1) the better we can model these abstractions in the π-calculus.

3.2.2 Encoding λ-terms with call-by-value reduction

Milner [63] has presented an encoding of the λ-calculus into the π-calculus. In [63] Milner showed the encodings for *call-by-name* and *call-by-value* reduction strategies. The syntax for λ-terms is given below:

e	::=	x	Variable
		$\lambda x.e$	Abstraction
		$e\ e$	Application

The core of the encoding of λ-terms into the π-calculus is the translation of function application, whereby function application becomes a particular form a parallel composition and β-reduction is modeled by interaction. Since the syntax of the first-order π-calculus only allows names to be transmitted along channels, the communication of a term is simulated by the communication of a *trigger* to it.

Informally we can describe the encoding as follows. In the pure λ-calculus every term denotes a function that when supplied with an argument yields another function (which might in turn wait for an argument, etc.). Taking this into account, the translation of a λ-term e is a process that is located at a port p waiting for some arguments.

It will wait until it receives along p a trigger x for its arguments and a new port q and evolves to a new process that is waiting at port q. The names p and q are unique ports along e interacts with its environment.

We will assume that the set of λ-calculus variables is a subset of the set of π-calculus variables, which avoids having to rename λ-calculus variables when translating λ-terms). The call-by-value translation $[\![]\!]^V_{\langle p\rangle}$ is defined as follows:

$$\begin{array}{rcl}
[\![x]\!]^V_{\langle p\rangle} & \stackrel{def}{=} & \overline{p}\langle x\rangle \\
[\![\lambda x.e]\!]^V_{\langle p\rangle} & \stackrel{def}{=} & (\nu\ f)(\overline{p}\langle f\rangle \mid\ !f(x,q).[\![e]\!]^V_{\langle q\rangle}) \\
[\![e\ e']\!]^V_{\langle p\rangle} & \stackrel{def}{=} & (\nu\ q)(\nu\ r)([\![e]\!]^V_{\langle q\rangle} \mid q(f).([\![e']\!]^V_{\langle r\rangle} \mid r(x).\overline{f}\langle x,p\rangle))
\end{array}$$

If e is just a variable, then we return immediately the value of this variable along channel p. If e is a λ-abstraction, we first create a new channel f, which we can think of as the location of $\lambda x.e$. We immediately return f along channel p and start the replicated process $!f(x,q).[\![e]\!]^N_{\langle q\rangle}$. This process acts as a server (see definition of higher-level form **def** $X[\tilde{x}] = P$ **in** Q).

We evaluate an application $e\ e'$ left-to-right. We start e and wait for the result located at f, to be sent along channel q. Then we start e' and wait for the result, x, to be sent along r. Now, we have two values: the function f and its argument x. We apply f to x by sending the pair $\langle x,p\rangle$ to f. The function will send its result along p once it is finished.

The reduction of a simple example will show that the encoding yields indeed the desired result:

$$\begin{array}{rcl}
[\![(\lambda x.x)\ y]\!]^V_{\langle p\rangle} & \equiv & (\nu\ q)(\nu\ r)([\![\lambda x.x]\!]^V_{\langle q\rangle} \mid q(f).([\![y]\!]^V_{\langle r\rangle} \mid r(x).\overline{f}\langle x,p\rangle)) \\
& \equiv & (\nu\ q)(\nu\ r)((\nu\ f)(\overline{q}\langle f\rangle \mid\ !f(x,q).[\![x]\!]^V_{\langle q\rangle}) \mid q(f).([\![y]\!]^V_{\langle r\rangle} \mid r(x).\overline{f}\langle x,p\rangle)) \\
& \rightarrow & (\nu\ q)(\nu\ r)((\nu\ f)(!f(x,q).[\![x]\!]^V_{\langle q\rangle}) \mid [\![y]\!]^V_{\langle r\rangle} \mid r(x).\overline{f}\langle x,p\rangle) \\
& \equiv & (\nu\ q)(\nu\ r)((\nu\ f)(!f(x,q).[\![x]\!]^V_{\langle q\rangle}) \mid \overline{r}\langle y\rangle \mid r(x).\overline{f}\langle x,p\rangle) \\
& \rightarrow & (\nu\ q)(\nu\ r)((\nu\ f)(!f(x,q).[\![x]\!]^V_{\langle q\rangle}) \mid \overline{f}\langle y,p\rangle) \\
& \rightarrow & (\nu\ q)(\nu\ r)((\nu\ f)(!f(x,q).[\![x]\!]^V_{\langle q\rangle}) \mid [\![y]\!]^V_{\langle p\rangle}) \\
& \equiv & [\![y]\!]^V_{\langle p\rangle} \mid (\nu\ q)(\nu\ r)(\nu\ f)(!f(x,q).[\![x]\!]^V_{\langle q\rangle}) \\
& \sim & [\![y]\!]^V_{\langle p\rangle}
\end{array}$$

The last step needs some explanation. This step goes beyond $\equiv$; it is a simple case of *strong bisimilarity* [65]. In fact, this step represents the "garbage-collection" of the process $!f(x,q).[\![x]\!]^V_{\langle q\rangle}$ which cannot be used further (f is a restricted name that was consumed in a previous interaction).

The reader should note that the encoding shown above is sequential. There exists also a parallel encoding. The parallel encoding of the λ-application $e\ e'$ is given below.

$$[\![e\ e']\!]^{V\|}_{\langle p\rangle} \stackrel{def}{=} (\nu\ q)(\nu\ r)([\![e]\!]^{V\|}_{\langle q\rangle} \mid [\![e']\!]^{V\|}_{\langle r\rangle} \mid q(f).r(x).\overline{f}\langle x,p\rangle)$$

Now, we evaluate an application $e\ e'$ in parallel. We start e, e', and the process $q(f).r(x).\overline{f}\langle x,p\rangle$ which waits first for the result located at f, to be sent along channel q and second, the result, x, to be sent along r. We apply the received values f to x by sending the pair $\langle x,p\rangle$ to f. The function will send its result along p once it is finished.

We can use both encodings as long as there are no synchronization constraints. In general, it is safe to use the parallel version, because synchronization will be achieved by an ordered sequence of input prefixes. On the other side, however, the sequentiel encoding is closer to the sequentiel nature of the λ-calculus.

3.2.3 Encoding λ-terms with call-by-name reduction

Now, we present the encoding of Ostheimer and Davie [80] for the call-by-name reduction strategy. We use the Ostheimer and Davie encoding rather than Milner's, since it shares much of the structure of the call-by-value encoding.

$$\begin{array}{rcl}
[\![x]\!]^{N}_{\langle p\rangle} & \stackrel{def}{=} & \overline{x}\langle p\rangle \\
[\![\lambda x.e]\!]^{N}_{\langle p\rangle} & \stackrel{def}{=} & (\nu\ f)(\overline{p}\langle f\rangle \mid\ !f(x,q).[\![e]\!]^{N}_{\langle q\rangle}) \\
[\![e\ e']\!]^{N}_{\langle p\rangle} & \stackrel{def}{=} & (\nu\ q)(\nu\ x)([\![e]\!]^{N}_{\langle q\rangle} \mid q(f).(\overline{f}\langle x,p\rangle \mid\ !x(c).[\![e']\!]^{N}_{\langle c\rangle}))
\end{array}$$

The encoding of $\lambda x.e$ is the same as for the call-by-value reduction. For the application $e\ e'$ we start $[\![e]\!]^{N}_{\langle q\rangle}$ and wait for the result f, to be sent along channel q. In contrast to the call-by-value encoding we do not start the evaluation of e'. Moreover, we start a replicated process on the channel x and apply f to the argument x and the result channel p. If f wishes to get the value associated with its argument x it must communicate with the replicated process on x.

The following simple example will show that we are able to reduce the function without evaluating its argument:

$$\begin{array}{rcl}
[\![(\lambda x.x)\ e]\!]^{N}_{\langle p\rangle} & \equiv & (\nu\ q)(\nu\ x)([\![\lambda x.x]\!]^{N}_{\langle q\rangle} \mid q(f).(\overline{f}\langle x,p\rangle \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle})) \\
& \equiv & (\nu\ q)(\nu\ x)((\nu\ f)(\overline{q}\langle f\rangle \mid\ !f(x,q).[\![x]\!]^{N}_{\langle q\rangle}) \mid q(f).(\overline{f}\langle x,p\rangle \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle})) \\
& \rightarrow & (\nu\ q)(\nu\ x)(\nu\ f)(!f(x,q).[\![x]\!]^{N}_{\langle q\rangle} \mid \overline{f}\langle x,p\rangle \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle}) \\
& \rightarrow & (\nu\ q)(\nu\ x)(\nu\ f)(!f(x,q).[\![x]\!]^{N}_{\langle q\rangle} \mid [\![x]\!]^{N}_{\langle p\rangle} \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle}) \\
& \equiv & (\nu\ q)(\nu\ x)(\nu\ f)(!f(x,q).[\![x]\!]^{N}_{\langle q\rangle} \mid \overline{x}\langle p\rangle \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle}) \\
& \rightarrow & (\nu\ q)(\nu\ x)(\nu\ f)(!f(x,q).[\![x]\!]^{N}_{\langle q\rangle} \mid [\![e]\!]^{N}_{\langle p\rangle} \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle}) \\
& \equiv & [\![e]\!]^{N}_{\langle p\rangle} \mid (\nu\ q)(\nu\ x)(\nu\ f)(!f(x,q).[\![x]\!]^{N}_{\langle q\rangle} \mid\ !x(c).[\![e]\!]^{N}_{\langle c\rangle}) \\
& \sim & [\![e]\!]^{N}_{\langle p\rangle}
\end{array}$$

3.2.4 Using channel sorts for encoding λ-terms

Pierce and Sangiorgi [87] have presented the typed version of this encoding and extended it by distinguishing between the ability to read from a channel, the ability the write to a channel, and the ability both to read and to write. This refinement led to a finer control of the use of channels. Furthermore, this refinement gave rise to a natural subtype relation similar to those in typed λ-calculi. The language PICT supports this form of channel type annotation and we will use them also in our object modelling.

3.3 The Pierce/Turner basic object model

Pierce and Turner [91] have outlined a basic model for objects in PICT, in which objects are modelled as a set of persistent processes representing instance variables and methods. The interface of an object is a record containing the channels of all exported features.

This basic model captures the essentials of concurrent objects: encapsulation, identity, persistence, instantiation, and synchronization. On the other side, however, this model did not cover some other common features of object-oriented programming languages that can be found in most of the better known languages like self-references of objects, dynamic binding, inheritance, overriding, genericity, and class variables.

3.3.1 Process groups as objects

In process calculi like the π-calculus it is possible to view a concurrent system as a kind of *process community*. Such a community is usually an unstructured collection of autonomous agents that use arbitrary patterns to communicate over channels. In practice, however, programs have significantly more structure and we can always identify some *invariants* the correct behaviour of the whole system depends on. The maintenance of these invariants tends to be local to a small groups of processes that cooperate with the rest of the system by some abstraction encapsulating their internal state.

Once, we have identified such groups of processes, communications across group boundaries can seen as messages from one group to another. The higher-level syntactic form **def** is a way to explicitly define a group of processes. In fact, a process group defined with **def** can be thought of as an object. The **def** construct helps to encapsulate the internal state of the object (by using restricted names within the process group).

3.3.2 Process-based vs. channel-based encoding

If we have a group of processes that maintains some local state, then we can think of that process group as an encoding of an updatable data structure. Turner [112] showed two possibilities to encode an updatable data structure: a process-based and

a channel-based encoding. For example, the process $\overline{Cell}\langle x, read, update\rangle$ represents a reference cell whose current contents is located at x. The channels *read* and *update* can be used to read or to modify the contents of the reference cell.

$$\begin{aligned}&\textbf{def } Cell[x, read, update] = \\ &\qquad \overline{read}\langle x\rangle.\overline{Cell}\langle x, read, update\rangle \;+\; update(n).\overline{Cell}\langle n, read, update\rangle\end{aligned}$$

This encoding ensures that a *read* and *update* request cannot be executed concurrently. This is possible due to the use of the summation operator which, in this case, is used to implement a mixed-guarded choice. The summation operator is part of the *synchronous π-calculus* [64]. The language PICT, however, is based on the asynchronous mini π-calculus that does not provide the summation operator.

Now, if we want to use the process-based encoding of updatable data structures the question arises whether we can encode mixed-choice in the asynchronous mini π-calculus or not? Palamidessi [81] showed that it is not possible to encode the full π-calculus into the asynchronous π-calculus. In particular, Palamidessi showed that in symmetric networks, it is not possible to solve the leader election problem by using only the asynchronous π-calculus, i.e. to guarantee that all processes will reach a common agreement (elect the leader) in a finite amount of time. On the other side, however, this is possible in the full π-calculus. For example, if we have a parallel composition of "symmetric" choices

$$P \mid Q \stackrel{def}{=} \overline{x}\langle a\rangle.P_0 \;+\; y(b).P_1 \mid x(b).Q_0 \;+\; \overline{y}\langle c\rangle.Q_1$$

where symmetric means that both process P and Q are identical under structural congruence modulo α-conversion, then a reduction will always lead to asymmetric systems either $P_0 \mid Q_0\{_b\backslash^a\}$ or $P_1\{_b\backslash^c\} \mid Q_1$. In contrast, the above system cannot be defined in the asynchronous π-calculus. Instead, due to the lack of synchronous output, a corresponding system with concurrently enabled input- and output-actions would behave confluently

$$P \mid Q \stackrel{def}{=} (\overline{x}\langle a\rangle \mid y(b).P_1) \mid (x(b).Q_1 \mid \overline{y}\langle c\rangle)$$

and we have for process P that either

$$P \xrightarrow{\overline{x}\langle a\rangle} (\mathbf{0} \mid y(b).P_1) \xrightarrow{y(b).P_1} (\mathbf{0} \mid P_1\{c/x\}) \text{ or } P \xrightarrow{y(b).P_1} (\overline{x}\langle a\rangle \mid P_1\{c/x\}) \xrightarrow{\overline{x}\langle a\rangle} (\mathbf{0} \mid P_1\{c/x\})$$

Similarly,

$$Q \xrightarrow{\overline{x}\langle a\rangle} (\mathbf{0} \mid y(b).Q_1) \xrightarrow{y(b).Q_1} (\mathbf{0} \mid Q_1\{c/x\}) \text{ or } Q \xrightarrow{y(b).Q_1} (\overline{x}\langle a\rangle \mid Q_1\{c/x\}) \xrightarrow{\overline{x}\langle a\rangle} (\mathbf{0} \mid Q_1\{c/x\})$$

Since both P and Q behave confluently, the symmetry $P \mid Q$ is preserved. In other words, no leader could be elected in a finite amount of time. But this violates the Palamidessi's uniformity requirement and therefore, it is not possible to encode mixed-choice into the asynchronous mini π-calculus.

Recently, Nestmann [71] proposed a "good" encoding for guarded choice. In particular, he presented the encoding for input-guarded choice, separate input- and output-guarded choice, and mixed-guarded choice. Nestmann based his investigation on the encoding of the input-guarded choice [72] that can be modelled in the asynchronous π-calculus. To encode mixed-guarded choice Nestmann proposed two solutions: (i) randomization and (ii) a "bakery" algorithm [50]. Basically, both mechanisms are used to break the symmetry (resolve cyclic dependencies among processes). Nevertheless, both solutions imply that either "uniformity" or "divergence-freedom" have to be dropped, if we do not want to leave the chosen framework which on the other side confirms Palamidessi's results [71].

An alternative encoding of a reference cell that does not use the summation operator, is shown below.

$$\textbf{def}\ \mathit{ChannelCell}[x, r] = (\nu\ \mathit{ref})(\overline{r}\langle \mathit{ref}\rangle \mid \overline{\mathit{ref}}\langle x\rangle)$$

Each reference cell is represented using a single restricted channel ref. Given an initial value located at x and a result channel r, the process $\mathit{ChannelCell}$ builds a new reference cell by creating a new channel ref and starting two local processes $\overline{r}\langle \mathit{ref}\rangle$ and $\overline{\mathit{ref}}\langle x\rangle$. The former returns the channel ref along channel r to the caller while the latter initializes the reference cell with x.

Now, we need two processes that implement the read and the update operation, respectively.

$$\textbf{def}\ \mathit{Read}[\mathit{ref}, r] = \mathit{ref}(\mathit{val}).(\overline{r}\langle \mathit{val}\rangle \mid \overline{\mathit{ref}}\langle \mathit{val}\rangle)$$
$$\textbf{def}\ \mathit{Update}[\mathit{ref}, \mathit{val}, r] = \mathit{ref}(\_).(\overline{\mathit{ref}}\langle \mathit{val}\rangle \mid \overline{r}\langle\rangle)$$

The process Read, given two parameter ref and r, reads a value val from ref (the current contents of the reference cell) and immediately sends this value back along result channel r. In parallel, the just received value val is sent back along channel ref. Similarly, given the parameters ref, val, and r, the process Update reads the current contents of the reference cell from ref and replaces it with val. Completion is singled by sending a empty tuple along r. The reader should note that Update reads the current contents of the reference cell into $\_$ – the wildcard name. This means that we are not interested in the contents, but this step is needed to replace the contents of the reference cell.

The processes $\mathit{ChannelCell}$, Read, and Update preserve the invariant that there is at most one active writer to the channel ref. Both Read and Update start with reading the actual contents from channel ref. A successful read has the effect that all other

Read and *Update* operations are temporarily blocked until the current process releases *ref* (sends a value along *ref*). Using this scheme, there will never be any interference between concurrent *Read* and *Update* operations.

What is left open, is an appropriate packing strategy that allows a group of *ChannelCell*, *Read*, and *Update* to be built. We can think of such a group as an implementation of an abstract data type having an interface that contains methods to read and to update its value.

3.3.3 Objects as records

Basically, using a pure process calculus, groups of processes exist nowhere but in the programmer's mind. However, the polyadic mini π-calculus already includes basic facilities for manipulating collections of channels. In the polyadic mini π-calculus we can construct and transmit tuples of channels (in fact, the definition of the higher-level form **def** uses already tuples). Having available a little more structure – the ability to refer explicitly to a group of processes – may suffice to view them as rudimentary *objects*. A tuple is an object, or more precisely, the interface to an object.

In the π-calculus, there is no way that one process can directly affect or refer to another process. A process can only send messages along some channels where, by convention, the other process listens. Similarly, referring to a group of processes means that we can send messages to a collection of channels where these processes are listening. An attractive notion to model such a facility is the standard notion of *records*. Records allow one to selectively address one member of a group by using its name. Viewing each individual channel as explicitly named "service access point", we can bundle them together in a record that provide a well-defined interface for accessing the related services. Furthermore, this packaging gives rise for a *higher-order* style of programming with objects, since a complete interface of one object may be manipulated as a single value.

3.3.4 The object model

An object in Pict is modelled as record. Pierce and Turner have proposed a basic encoding of objects with records [91]. For example, a reference cell with two methods **get** and **set** can be implemented as follows:

```
def ref [init] =
  let
    new Contents
    run Contents!init
  in
    record
      set = abs [v,c] > Contents?_ > (Contents!v | c![]) end,
       get = abs [r] > Contents?v > (Contents!v | r!v) end
    end
  end
```

A reference cell maintains an internal channel `Contents`, which is used to store the current state of the object, and provides two request channels `set` and `get` to set a new state and to read the current state, respectively. The initial state of the reference cell is given by the `init` parameter. In order to protect other processes against reading and writing `Contents`, its declaration and initialization is wrapped in a local block.

Each request channel is the interface to another process. These processes are defined as anonymous process abstractions, and are the only means by which it is possible to query or to change the state of the object.

The process `ref` is an object generator that can be seen as a reference cell factory. The following `val` declaration creates a reference cell with the initial value 50:

```
val cell = ref [50]
```

Requests to an object are performed by the usual dot notation:

```
run cell.set[20]; prInt [cell.get[]];
```

Now, we can implement the class `stack` in Pict. The complete code is given in Figure 3.3. The reader should note that the class variable `Pushed` is implemented as "global" process. The reason is that the basic object model only supports object based features. However, the basic model captures the essentials of concurrent objects.

```
def stack [] =
    let
        new localPushed, Stk
        run localPushed!0
        run Stk!(nil[])
    in
        record
            empty = abs[c] > Stk?aList > Stk!aList | c!(null[aList]) end,
            push = abs[x,c] >
                        Pushed?m >
                            localPushed?n >
                                Stk?l > Stk!(cons[x,l]) | localPushed!(n+1) | Pushed!(m+1) | c![]
                            end,
            pop = abs[c] >
                        Stk?l > if null[l]
                            then
                                raise![exitOnExn,"Stack: Underflow",c]
                            else
                                Pushed?m >
                                    localPushed?n > Stk!(cdr[l]) | localPushed!(n-1) | Pushed!(m-1) | c![]
                            end
                        end,
            top = abs[r] >
                        Stk?l > if null[l]
                            then
                                raise![exitOnExn,"Stack: Underflow",r]
                            else
                                Stk!l | r!(car[l])
                            end
                        end,
            localPushed = abs[r] > localPushed?v > localPushed!v | r!v end,
        end
    end

new Pushed
run Pushed!0
```

Figure 3.3: A stack object in PICT.

3.4 Explicit metaobjects

In this section we present some extensions to the basic object model resulting from experiences modelling object-oriented abstractions in PICT [52, 104, 105].

3.4.1 Modelling class variables

As a first extension we add class variables and class methods to the basic model.

A straightforward mapping of these features is to define them in global scope as processes (see Figure 3.3), but this violates data encapsulation and allows every client to access these features. The most natural solution to this problem is to introduce explicit metaobjects to encapsulate the logic for creating and initializing instances of a class. Metaobjects [48] are a commonly used mechanism in various object-oriented programming languages to encapsulate the inter- pret of language features behind the interface of an object. In this case we use metaobjects to encapsulate and restrict access to class variables and methods. Class variables and class methods are modelled as instance variables and exported methods of the metaobject, respectively.

We use metaobjects not only to model shared class features, but more generally to create, initialize, and control the behaviour of objects. In the basic model of Pierce and Turner, object creation is modelled by a generator process in global scope. Moving this generator process inside the metaobject is a first step towards modelling inheritance and self-references. A metaobject for the class `stack` called `StackClass` is shown in Figure 3.4.

`StackClass` is declared as a unique global channel representing the metaobject for a stack class. Since the class variable `Pushed` is declared within the scope of the class declaration, all methods can directly access the value of `Pushed`.

There are two class methods: `pushed` and `create`. The class method `pushed` returns the total number of pushed items (for all instances of class `stack`), whereas the class method `create` returns a new object of the stack class.

The reader should note that we can add a generic type parameter `T` to the definition of the `create` method. This allows us to create stack objects for arbitrary data types. The metaobject itself does not have to be generic. The the following we denote with

$$\text{create} = \text{abs } [:\text{T}:][\text{r}]$$

the ability to create generic objects using T as actual type parameter.

```
val StackClass =                                        {- global metaobject channel -}
    let
        new Pushed
        run Pushed!0                                    {- private class variable -}
    in
        record
            pushed = abs [r] >                          {- public class method -}
                Pushed?value > (Pushed!value | r!value)
            end,
            create = abs [r] >                          {- creation interface -}
                let
                    new localPushed, Stk
                    run localPushed!0
                    run Stk!(nil[])
                in
                    record
                        empty = abs[c] > Stk?aList > Stk!aList | c!(null[aList]) end,
                        push = abs[x,c] >
                            Pushed?m >
                                localPushed?n >
                                    Stk?l > Stk!(cons[x,l]) | localPushed!(n+1) | Pushed!(m+1) | c![]
                                end,
                        pop = abs[c] >
                            Stk?l > if null[l]
                                then
                                    raise![exitOnExn,"Stack: Underflow",c]
                                else
                                    Pushed?m >
                                        localPushed?n > Stk!(cdr[l]) | localPushed!(n-1) | Pushed!(m-1) | c![]
                                end
                            end,
                        top = abs[r] >
                            Stk?l > if null[l]
                                then
                                    raise![exitOnExn,"Stack: Underflow",r]
                                else
                                    Stk!l | r!(car[l])
                                end
                            end,
                        localPushed = abs[r] > localPushed?v > localPushed!v | r!v end,
                end
        end
    end
```

Figure 3.4: A metaobject for class `stack` in PICT.

3.4.2 Modelling inheritance by dynamic binding of `Self`

The pseudo-variable `Self` is needed to model dynamic binding. To model this feature, we make use of a PICT library process that implements so-called *reference cells*. A reference cell is an object that provides set and get methods to set and retrieve stored values, respectively. `Self` is modelled as a reference cell that is set just once in the `create` method of the metaobject. In order to initialize `Self`, we first have to assign the new fresh object to a temporary channel (this is the generator process), and second to define a *fixed point operator* which delivers the minimal fixed point - `Self`:

```
create = abs [:T:][r] >
  r!( let
    val Self = emptyRef[:AObjectType:][]
    new temporary - make a new channel for Self -
    run temporary!( ...
      object creation
      ... )
  in
    Self.set[temporary]; {- bind Self -}
    Self.get[]           {- return current value of Self -}
  end)
```

In a first approach we model inheritance by delegation (as in Self [113] and Sina [3]): each object owns an instance of its direct superclass. This means that only the exported methods of the superclass can be accessed by the subclass instance. Modelling dynamic binding requires special care. In the absence of dynamic binding of `Self`, if an inherited ancestor method calls a method redefined by the subclass, the original and not the redefined method will be called since `Self` within the superclass instance refers to the superclass object, but not to the subclass object. To achieve dynamic binding, we need a superclass instance in which `Self` refers to the subclass instance [26] (Figure 3.5).

We now introduce "intermediate object" in which all methods and instance variables of a class are defined, but `Self` is unbound: all methods have an additional first parameter `Self`. The metaobject of each class defines a process `CreateIntermediate` (comparable with a generator in [26]) where the intermediate object of the class is defined as follows:

```
def CreateIntermediate [] =
  let
    new localPushed, Stk
    run localPushed!0
    run Stk!(nil[])
```

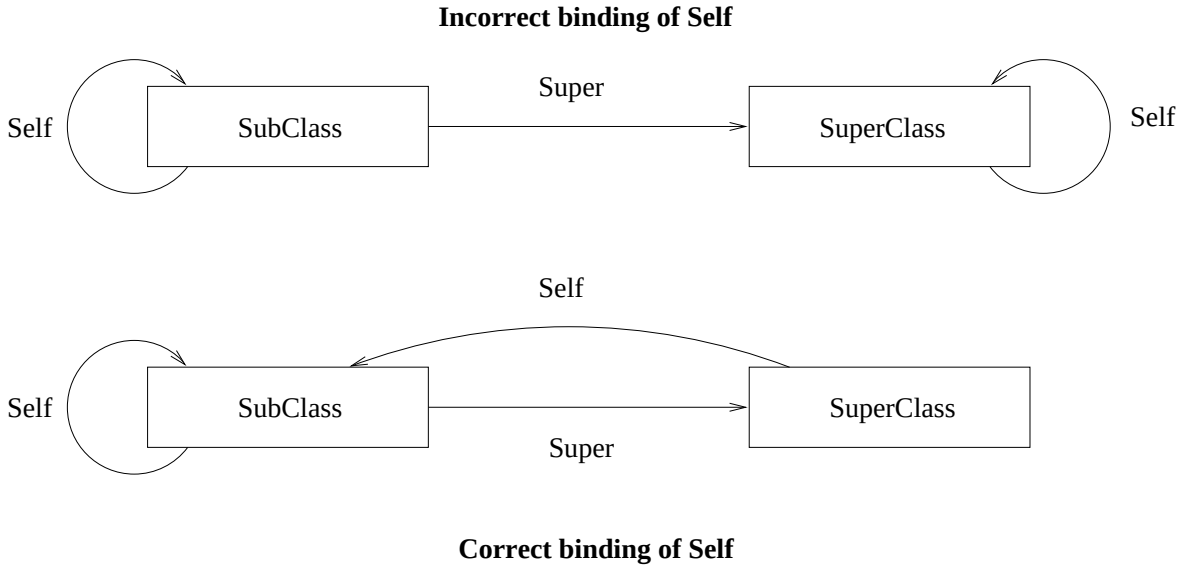

Figure 3.5: Binding of `Self` with inheritance.

```
  in
    record
      empty = abs[Self, c] > ... {- Self is an explicit parameter -}
      push = abs[Self, x,c] > ...
      pop = abs[Self, c] > ...
      top = abs[Self, r] > ...
      localPushed = abs[Self, r] > ...
    end
  end
```

In the `Create` method of the metaobject, an intermediate object is created, each exported method is bound to a method defined in the intermediate object, and the correct binding of `Self` is established. As in the previous model, an empty reference cell is used to model self- reference. The method `Create` is defined as follows:

```
def Create [] =
  let
    val StackIntermediate = CreateIntermediate []
    val Self = emptyRef []
    val NewInstance =
      record
        empty = abs[c] = StackIntermediate.empty[Self.get[],c] end,
        push = abs[x,c] = StackIntermediate.push[Self.get[],x,r] end,
```

```
        pop = abs[c] = StackIntermediate.pop[Self.get[],c] end,
        top = abs[r] = StackIntermediate.top[Self.get[],r] end,
        localPushed = abs[r] =
             StackIntermediate.localPushed[Self.get[],r] end
    end
in
  Self.set[NewInstance];
  Self.get[]
end
```

Now, in addition to exporting the `Create` method and all other public class methods, the metaobject exports the method `CreateIntermediate`, which returns a fresh copy of an intermediate object of the class.

Inheritance is now straightforward to model. In order to reuse the methods defined in an ancestor class, the metaobject of a class gets a fresh copy of the intermediate object of its direct superclass. This intermediate object is then used to define the intermediate object of the class itself. It is possible to (i) override methods, (ii) define new methods, and (iii) call inherited methods. Figure 3.5 illustrates the final architecture of the object model for the π-calculus using Pict as implementation language.

3.5 Results and shortcomings

The shown encodings illustrate that the π-calculus is expressive enough for modelling standard object-oriented programming language features in a convenient way. Walker [120] has shown that POOL [5] can be modelled in the π-calculus, but in his approach, no subtyping or inheritance is supported. Subtyping and a notion of Self can be modelled with the "Calculus of Objects" of Vasconcelos [117]. Barrio [9] has given a nearly complete representation of active objects in the calculus, but dynamic binding and a notion of Self are still missing. Our encodings show that inheritance, dynamic binding, and self-reference can also be conveniently modelled with the π-calculus with the aid of processes representing metaobjects.

Although metaobjects are usually associated with MOPs, we only defined a basic MOP for our Pict object encodings. Two major questions arise: what kind of MOPs do we need in a composition language, and what are the consequences for the underlying type system? To our knowledge, most of the languages supporting run-time MOPs are not statically typed. It is therefore a challenging task to see what kind of MOP can be defined with the current type system of Pict, or how the type system should be extended in order to support run-time reflection using metaobjects.

Modelling object-oriented features in the π-calculus is tedious work, akin to programming in a "concurrent assembler". The programming language Pict [89] simplifies this work somewhat by providing syntax for a large number of common, basic

programming abstractions, like Booleans and integers, control structures, functions, expressions, and statements. Still, to model objects as processes, one is often obliged to forsake natural abstractions and explicitly describe behaviour in low-level, operational terms. For example, to specify the reference cell given in section 3.3 or a stack presented in Figure 3.3, we had to explicitly create and manipulate the reply channel used to deliver the results to clients.

It is possible to specify objects (and methods) in PICT without explicitly mentioning reply channels, but the abstractions needed to do so are not immediately obvious [104]. Therefore we need a less primitive, intermediate calculus that is more convenient for modelling concurrent objects. For example, one solution to this problem can be to use a so-called "guarded object calculus" (GOC) [78] in which an objects is modelled as a set of functions that read and write a local tuple space of messages representing the object's state. Whenever an operation is called on an object, an input guard grabs the needed resources from the tuple space. After the calculation, an output trigger restores resources.

The use of guards and triggers for modelling objects has the advantage that (i) it is possible to specify any kind of operation in GOC style and (ii) objects behave correctly in the presence of multiple clients. On the other hand it is still an open question how to model other abstractions, such as local method calls, self-references and inheritance.

Our overall goal in this work is to develop a formal model of software composition and an executable *composition language* [76] for specifying components, composition abstractions, and applications as compositions of software components. A composition language for open systems should not only have its formal semantics specified in terms of communicating processes, but should really support concurrent and distributed behaviour. The run-time system of current implementation of PICT only runs on a single processor; it is not possible to specify real distribution of processes. As a first step towards real distribution, is a small system which implements a subset of the PICT programming language supporting communication between distributed nodes [118]. What we need, however, is a distributed abstract machine as run-time system for the composition language, comparable to that used for Java [39]. Furthermore, a distributed abstract machine for software composition could be built on top of an existing intercomponent communication system (e.g., COM [60, 17] or CORBA [79]).

The type system of PICT integrates a number of features found in recent work on theoretical foundations for typed object-oriented languages [112] and allows the definition of polymorphic data structures and processes, what we heavily use in our encodings. However, our results show that the current type system is too restrictive for an efficient implementation of metaobject protocols and lacks of a support for runtime type information.

However, when implementing more sophisticated abstractions like McHale's *generic synchronization policies* [55], we can discover an interesting property of the type system of PICT: it is not only possible to define generic *classes*, but also generic *methods*. To

our knowledge, no strongly typed object-oriented programming language supports such a feature.

Chapter 4

The $\pi\mathcal{L}$-calculus

A general purpose composition language based on a formal semantic foundation will facilitate precise specification of glue abstractions and compositions, and will support reasoning about their behaviour. The semantic foundation, however, must address our glue requirements, namely that we are able to model concurrency, communication, reified communications (i.e., messages as first-class entities), abstraction of arbitrary behaviour, and polymorphic interfaces. With such a glue language it should be easy to model common glue patterns, i.e., those concerned with adaptation (of interfaces and behaviour) and composition (of multiple components).

The informal requirements, given above, suggest an approach in which we use a formal process calculus as a "core language" for a more usable and practical glue language. The (asynchronous) π-calculus has many of the features we need, and has been used successfully as a core language for Pict [92]. Although the π-calculus can be used to model composition mechanisms [105], it is inconvenient for modeling general glue abstractions due to the dependence on positional parameters in communications. For example, generic readers/writers synchronization policies cannot be directly coded without wrapping method arguments in order to treat an arbitrary number of arguments as one value [116].

Dami has tackled a similar problem in the context of the λ-calculus, and has proposed λN [28, 30], a calculus in which parameters are identified by names rather than positions. The resulting flexibility and extensibility can also be seen, for example, in HTML forms, whose fields are encoded as named (rather than positional) parameters in URLs, in Python [115], where functions can be defined to take arguments by keywords, and in Visual Basic [59], where *named arguments* can be used to break the order of possibly optional parameters.

In this chapter we develop the $\pi\mathcal{L}$-calculus, an offspring of an asynchronous fragment of the π-calculus. The asynchronous sublanguage was proposed first by Boudol [13] and Honda and Tokoro [44]. Sangiorgi [101] extended the proposal by allowing polyadic communication.

Based on the idea of Dami, in the $\pi\mathcal{L}$-calculus the communication of tuples is replaced with the communication of labeled parameters. In fact, in the $\pi\mathcal{L}$-calculus parameters are identified by names rather than positions.

In this chapter we develop the basic theory for $\pi\mathcal{L}$ while in chapter 5 a polymorphic type system for $\pi\mathcal{L}$ is given.

4.1 Towards labelled communication

In [28, 30], Dami has studied an extended lambda calculus called λN (lambda calculus with Names) in which *names* instead of *positions* are used for interaction between components. In the standard lambda calculus variables are used for naming parameters, but the names do not belong to the semantics of functions. Two lambda terms are considered equivalent modulo α-substitution, which consists of changing the names of bound variables. Lambda expressions that can be converted into one another by α-substitution are called α-equivalent. In fact, by using only de Bruijn indices [31], names disappear totally, because arguments to functions are uniquely identified by their position. In the standard lambda calculus the functions $\lambda(xy)x$ and $\lambda(yx)y$ are equivalent, but $\lambda(xy)x$ and $\lambda(yx)x$ are different. In λN, however, the first two functions are different while the latter two are equivalent. This property of λN is called by Dami *extensibility*, which is of great interest for modeling a large number of programming abstractions, particular object-oriented and component-oriented abstractions.

Dami [28] describes extensibility as the possibility to add new functionality to an existing piece of code without affecting the previous behaviour. In fact, we can always replace an environment defining a given set of names by a bigger environment, defining more names. All lookup operations involving the original set of names are still valid in the new environment.

To illustrate the new expressive power of λN consider the Church encoding of booleans and the *not* function in the standard λ-calculus and in the λN-calculus:

$$\begin{aligned} \mathbf{True} &= \lambda true.\lambda false.true \\ \mathbf{False} &= \lambda true.\lambda false.false \\ \mathbf{Not} &= \lambda arg.\lambda true.\lambda false.arg\ false\ true \end{aligned}$$

These encodings have the desired property that the application **Not True** yields **False**. But now we want a three-value logic with an *unknown* value. Since the number of arguments as well as their position is significant, everything has to be recoded:

$$\begin{aligned} \mathbf{True_U} &= \lambda true.\lambda false.\lambda unknown.true \\ \mathbf{False_U} &= \lambda true.\lambda false.\lambda unknown.false \\ \mathbf{Unknown_U} &= \lambda true.\lambda false.\lambda unknown.unknown \\ \mathbf{Not_U} &= \lambda arg.\lambda true.\lambda false.\lambda unknown.arg\ false\ true\ unknown \end{aligned}$$

Unfortunately, this encoding is not compatible with the previous one. The application of $\mathbf{Not_U}$ to **True** does not yield the desired result because $\mathbf{Not_U}$ **True** does not reduce to **False** and $\mathbf{False_U}$, respectively.

In contrast, in λN the encoding is extensible. There we have a *bind expression*, written $(l = b)$, which binds term b to label l. When such a binding is applied to a λN abstraction then all *named parameters* with label l are substituted by the term b in the abstraction. The order in which the bindings are applied to the abstraction is unimportant. The result is always the same. This leads to the desired extensibility property. A given λN term can easily be extended by providing an additional binding which represents the new feature. Dami has demonstrated an extensible encoding of booleans in the λN-calculus in [28, 30].

Now we present how booleans and the *not* function can be encoded using the asynchronous polyadic π-calculus [101]. We use the same scheme as in section 3.2. The following processes represent the encoding:

$$\begin{aligned}
\mathbf{True}(r) &= (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{t}) \\
\mathbf{False}(r) &= (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{f}) \\
\mathbf{Not}(b,r) &= (\nu\ t,f)(b(c).\overline{c}(t,f) \mid t.\mathbf{False}(r) \mid f.\mathbf{True}(r))
\end{aligned}$$

The boolean values **True** and **False** are the same as in section 3.2. The process **Not** takes two arguments: (i) a channel b that serves as the location of the original boolean value and (ii) a result channel r along **Not** returns the location of the negated boolean value. Internally, **Not** creates two new channels t and f and send them along b. In parallel, **Not** starts two processes that listen at t and f, respectively. If the boolean value signals at t – the interaction for value *true* – then **Not** returns along channel r the value *false*. If the boolean value signals at f – the interaction for value *false* – then **Not** returns along channel r the value *true*.

These encodings have the desired property that an application of **Not** to **True** behaves identically to **False**. We use the notion of *asynchronous bisimulation* [4] to denote that processes in the asynchronous π-calculus behave equally. It is important to note that the asynchronous bisimulation is a congruence because unlike in the synchronous version of the π-calculus the bisimulation is preserved by all operators. We have two versions of bisimulation: (i) the *strong asynchronous bisimulation*, written $\sim_a$, which takes all transitions into account and (ii) the *weak asynchronous bisimulation*, written $\approx_a$, which abstracts from silent actions (τ-actions) that are considered as unobservable. An additional property of asynchronous bisimulation is that all forms of name instantiation (e.g. ground, early, late, and open) coincide [4, 40], basically because this bisimulation is preserved by name instantiation. So we have:

$$(\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)) \quad \approx_a \quad \mathbf{False}(r)$$

To illustrate that the left process is indeed equivalent with $\mathbf{False}(r)$, we show the reduction of it yielding the desired result.

$$\begin{array}{ll}
 & (\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)) \\
\equiv & (\nu\ a)((\nu\ t,f)(a(c).\overline{c}(t,f) \mid t.\mathbf{False}(r) \mid f.\mathbf{True}(r)) \mid (\nu\ b)(\overline{a}(b) \mid !b(t,f).\overline{t})) \\
\rightarrow & (\nu\ b)((\nu\ t,f)(\overline{b}(t,f) \mid t.\mathbf{False}(r) \mid f.\mathbf{True}(r)) \mid !b(t,f).\overline{t}) \\
\equiv & (\nu\ b)((\nu\ t,f)(\overline{b}(t,f) \mid t.\mathbf{False}(r) \mid f.\mathbf{True}(r)) \mid b(t,f).\overline{t} \mid !b(t,f).\overline{t}) \\
\rightarrow & (\nu\ b)((\nu\ t,f)(t.\mathbf{False}(r) \mid f.\mathbf{True}(r)) \mid \overline{t} \mid !b(t,f).\overline{t}) \\
\rightarrow & (\nu\ b)((\nu\ t,f)(\mathbf{False}(r) \mid f.\mathbf{True}(r)) \mid !b(t,f).\overline{t}) \\
\sim_a & \mathbf{False}(r)
\end{array}$$

Again, when we want to model a three-value logic in the asynchronous polyadic π-calculus we also have to recode everything, because a boolean value is now a channel b listening for a *triple* of channels (t,f,u) where each element represents the next corresponding interaction:

$$\begin{array}{rcl}
\mathbf{True_U}(r) & = & (\nu\ b)(\overline{r}(b) \mid !b(t,f,u).\overline{t}) \\
\mathbf{False_U}(r) & = & (\nu\ b)(\overline{r}(b) \mid !b(t,f,u).\overline{f}) \\
\mathbf{Unknown_U}(r) & = & (\nu\ b)(\overline{r}(b) \mid !b(t,f,u).\overline{u}) \\
\mathbf{Not_U}(b,r) & = & (\nu\ t,f,u)(b(c).\overline{c}(t,f,u) \\
 & & \quad \mid t.\mathbf{False_U}(r) \mid f.\mathbf{True_U}(r) \mid u.\mathbf{Unknown_U}(r))
\end{array}$$

As in case of the λ-calculus, this encoding is not compatible with the previous one. Moreover, we cannot apply $\mathbf{Not_U}$ to $\mathbf{True}$. The reason is that a communication can occur only if the sender and the receiver agree on the arity [64] (channels must have the same sort [64, 100]).

$$(\nu\ a)(\mathbf{Not_U}(a,r) \mid \mathbf{True}(a)) \quad \not\approx_a \quad \mathbf{False}(r)$$

The left process cannot be reduced to a process that is equivalent to $\mathbf{False}(r)$:

$$\begin{array}{ll}
 & (\nu\ a)(\mathbf{Not_U}(a,r) \mid \mathbf{True}(a)) \\
\equiv & (\nu\ a)((\nu\ t,f,u)(a(c).\overline{c}(t,f,u) \mid t.\mathbf{False_U}(r) \mid f.\mathbf{True_U}(r) \mid u.\mathbf{Unknown_U}(r)) \\
 & \quad \mid (\nu\ b)(\overline{a}(b) \mid !b(t,f).\overline{t})) \\
\rightarrow & (\nu\ b)((\nu\ t,f,u)(\overline{b}(t,f,u) \mid t.\mathbf{False_U}(r) \mid f.\mathbf{True_U}(r) \mid u.\mathbf{Unknown_U}(r)) \\
 & \quad \mid !b(t,f).\overline{t}) \\
\equiv & (\nu\ b)((\nu\ t,f,u)(\overline{b}(t,f,u) \mid t.\mathbf{False_U}(r) \mid f.\mathbf{True}(r_U) \mid u.\mathbf{Unknown_U}(r)) \\
 & \quad \mid b(t,f).\overline{t} \mid !b(t,f).\overline{t}) \\
\not\approx_a & \mathbf{False}(r)
\end{array}$$

The subprocesses $\overline{b}(t, f, u)$ and $b(t, f).\overline{t}$ cannot communicate because they have a different arity. Therefore, the process $(\nu\ a)(\mathbf{Not_U}(a, r) \mid \mathbf{True}(a))$ cannot be reduced to a process that is equivalent to $\mathbf{False}(r)$ (a reduction to $\mathbf{False_U}(r)$ does also not exist).

To solve this problem we need to replace communication of tuples that impose a discipline in order (position dependency) and arity. Like Dami, we introduce an explicit naming scheme to address parameters by names. In $\pi\mathcal{L}$ we use so-called *forms* in place of tuples. Therefore, in the $\pi\mathcal{L}$-calculus the communication of tuples is replaced by forms (mappings from labels to names).

Using forms, boolean values and the *not* function can be encoded as follows:

$$\begin{aligned}
\mathbf{True}(r) &= (\nu\ b)(\overline{r}(\langle val = b\rangle) \mid\ !b(X).\overline{X_{true}}) \\
\mathbf{False}(r) &= (\nu\ b)(\overline{r}(\langle val = b\rangle) \mid\ !b(X).\overline{X_{false}}) \\
\mathbf{Not}(c, r) &= c(Y).((\nu\ b)(\overline{r}(\langle val = b\rangle) \mid\ !b(X).\overline{Y_{val}}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle)))
\end{aligned}$$

The agents **True** and **False** are quite similar to their counterparts in the π-calculus except that we now exchange forms. Both **True** and **False** create a new channel b and return along channel r the binding $\langle val = b\rangle$. In parallel, both processes start a subprocess that is listing on the channel b for a form X. If the processes receive X, then they signal at X_{true} and X_{false}, respectively. X_l is a projection from labels to names and if a form X contains a binding for label l, then X_l yields the corresponding name, otherwise $\mathcal{E}$ called *empty binding*.

Unlike the π-process **Not**, the $\pi\mathcal{L}$-agent **Not** does not have to perform a test on the boolean value. For example, the π-process can be rewritten as

$$\mathbf{Not}(b, r) = \text{if } b \text{ then } \mathbf{False}(r) \text{ else } \mathbf{True}(r)$$

to stress the fact that we perform a test operation. In the case of $\pi\mathcal{L}$-agent **Not**, there is no such operation. Moreover, we simply update a received form X by rebinding *true* and *false* and sending the new form along the continuation channel denoted by Y_{val}. In fact, this corresponds to a simple form of *message interception*. By simply adding a new set of bindings we get the desired result. The reader should note that the projection X_l always yields the name for rightmost binding containing label l if such a binding for l exists.

In λN we must always use labeled variables. In $\pi\mathcal{L}$ we introduce *form variables*. These variables do not have a label, and therefore they match a complete form received in a communication. Form variables treat the received form as an opaque value. The agent using these variables is not interested in the concrete bindings. This allows an agent both to send and to receive arbitrary forms.

In the encoding of **Not** we have used X and Y as form variables. These variables match a complete form sent along b and c, respectively. The role of X is to match the bindings for the interactions for the values *true* and *false*. Y contains only one binding – the location of a boolean value addressed by label *val*.

In $\pi\mathcal{L}$, unlike in π, we need to obey some naming discipline for labels. Whenever two $\pi\mathcal{L}$-agents are willing to communicate they must also agree on the set of labels they want to use. In the case of the encoding of the boolean values and the *not* function, the $\pi\mathcal{L}$-agents use the label *val* to address a boolean. One can choose arbitrary label names as long as both agents use the same. In contrast, in the π-calculus we have *formal* and *actual* parameters which do not need to be the same in the sender as well as in the receiver. The parameters are simply matched up by their position.

Now, when do two $\pi\mathcal{L}$-agents behave equally? As for the asynchronous π-calculus we develop a notion of bisimulation for the $\pi\mathcal{L}$-calculus. The notion of bisimulation for $\pi\mathcal{L}$ is based on the asynchronous bisimulation for the asynchronous π-calculus [4]. In Section 4.4 we shall present its development on top of a labelled transition system. As for the asynchronous π-calculus we have two forms of bisimulation for $\pi\mathcal{L}$: (i) *strong labelled bisimulation*, written $\overset{\mathcal{L}}{\sim}$, to denote that two $\pi\mathcal{L}$-agents behave equally under all actions and (ii) *weak labelled bisimulation*, written $\overset{\mathcal{L}}{\approx}$, which abstracts from silent actions (τ-actions). In the case of the application of the $\pi\mathcal{L}$-agent **Not** to **True** we have:

$$(\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)) \quad \overset{\mathcal{L}}{\approx} \quad \mathbf{False}(r)$$

By reducing the left agent it can be shown that it is indeed equivalent with $\mathbf{False}(r)$:

$$\begin{array}{ll} & (\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)) \\ \equiv & (\nu\ a)(a(Y).((\nu\ b)(\overline{r}(\langle val{=}b\rangle) \mid\ !b(X).\overline{Y_{val}}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle)))) \\ & \qquad \mid (\nu\ b)(\overline{a}(\langle val{=}b\rangle) \mid\ !b(X).\overline{X_{true}})) \\ \rightarrow & (\nu\ b)((\nu\ b')(\overline{r}(\langle val{=}b'\rangle) \mid\ !b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle))) \\ & \qquad \mid\ !b(X).\overline{X_{true}}) \\ \overset{\mathcal{L}}{\approx} & \mathbf{False}(r) \end{array}$$

The last step is surprising, but if we put both agent $(\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a))$ and agent $\mathbf{False}(r)$ in an arbitrary context, for example:

$$\mathcal{C}[\cdot] \quad \overset{def}{=} \quad (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid [\cdot])$$

then both agents will signal along channel f and therefore, they are equivalent. We have

$$\begin{array}{ll} & \mathcal{C}[((\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)))] \\ = & (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid ((\nu\ a)(\mathbf{Not}(a,r) \mid \mathbf{True}(a)))) \\ \equiv & (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) \\ & \qquad \mid ((\nu\ a)(a(Y).((\nu\ b)(\overline{r}(\langle val{=}b\rangle) \end{array}$$

$$
\begin{array}{lll}
 & & \quad \mid !b(X).\overline{Y_{val}}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle))) \\
 & & \quad \mid (\nu\ b)(\overline{a}(\langle val{=}b\rangle) \mid !b(X).\overline{X_{true}})))) \\
\rightarrow & (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) & \\
 & \quad \mid ((\nu\ b)((\nu\ b')(\overline{r}(\langle val{=}b'\rangle) & \\
 & \quad \quad \mid !b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle))) & \\
 & \quad \mid !b(X).\overline{X_{true}}))) & \\
\rightarrow & (\nu\ t,f)(\nu\ b')(\overline{b'}(\langle true{=}t\rangle\langle false{=}f\rangle) & \\
 & \quad \mid ((\nu\ b)(!b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \quad \mid !b(X).\overline{X_{true}}))) & \\
\equiv & (\nu\ t,f)(\nu\ b')(\overline{b'}(\langle true{=}t\rangle\langle false{=}f\rangle) & \\
 & \quad \mid ((\nu\ b)(!b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \quad \mid b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \quad \mid !b(X).\overline{X_{true}}))) & \\
\rightarrow & (\nu\ t,f)(\nu\ b')(\nu\ b)(!b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \mid \overline{b}((\langle true{=}t\rangle\langle false{=}f\rangle)\langle true{=}f\rangle\langle false{=}t\rangle) & \\
 & \quad \mid !b(X).\overline{X_{true}}) & \\
\equiv & (\nu\ t,f)(\nu\ b')(\nu\ b)(!b'(X).\overline{(\langle val{=}b\rangle)_{val}}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \mid \overline{b}((\langle true{=}t\rangle\langle false{=}f\rangle)\langle true{=}f\rangle\langle false{=}t\rangle) & \\
 & \quad \mid b(X).\overline{X_{true}} \mid !b(X).\overline{X_{true}}) & \\
\rightarrow & (\nu\ t,f)(\nu\ b')(\nu\ b)(!b'(X).\overline{b}(X\langle true{=}X_{false}\rangle\langle false{=}X_{true}\rangle) & \\
 & \quad \mid \overline{f} \mid !b(X).\overline{X_{true}}) & \\
\overset{\mathcal{L}}{\sim} & \overline{f} &
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathcal{C}[\mathbf{False}(r)] \\
= & (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid \mathbf{False}(r)) \\
\equiv & (\nu\ r)(\nu\ t,f)(r(X).\overline{X_{val}}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid (\nu\ b)(\overline{r}(\langle val{=}b\rangle) \mid !b(X).\overline{X_{false}})) \\
\rightarrow & (\nu\ b)(\nu\ t,f)(\overline{b}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid !b(X).\overline{X_{false}}) \\
\equiv & (\nu\ b)(\nu\ t,f)(\overline{b}(\langle true{=}t\rangle\langle false{=}f\rangle) \mid b(X).\overline{X_{false}} \mid !b(X).\overline{X_{false}}) \\
\rightarrow & (\nu\ b)(\nu\ t,f)(\overline{f} \mid !b(X).\overline{X_{false}}) \\
\overset{\mathcal{L}}{\sim} & \overline{f}
\end{array}
$$

□

It is important to note that both agents would also behave equally if we had defined

context $\mathcal{C}[\cdot]$ as follow:

$$\mathcal{C}[\cdot] \stackrel{def}{=} (\nu\ r)(\nu\ f)(r(X).\overline{X_{val}}(\langle \mathit{false} = f\rangle) \mid [\cdot])$$

We say that the form $\langle \mathit{false} = f\rangle$ is complete with repect to $(\nu\ a)(\mathbf{Not}(a, r) \mid \mathbf{True}(a))$, because it contains at least a binding for label *false*. That is, $(\nu\ a)(\mathbf{Not}(a, r) \mid \mathbf{True}(a))$ and all its derivatives will never yield any $\mathcal{E}$ in context $\mathcal{C}[\cdot]$.

As in the case of the λN-calculus, the extension to a three-value logic is now compatible with the encoding of the two-valued logic. Moreover, we also do not need to recode the *not* function because it is already prepared to handle other bindings than *true* and *false*. In fact, the chosen encoding of **Not** allows us to use **Not** without any change in the three-valued logic. This is possible due to the polymorphic use of *form variables*. Therefore, in the three-valued logic we only have to add an agent for the *unknown* value.

$$\mathbf{Unknown}(r) \quad = \quad (\nu\ b)(\overline{r}(\langle val = b\rangle) \mid !b(X).\overline{X_{unknown}})$$

The application of **Not** to **True** or **False** yields the same results as in the two-valued logic. Moreover, the application of **Not** to **Unknown** behaves also as desired and we have:

$$(\nu\ a)(\mathbf{Not}(a, r) \mid \mathbf{Unknown}(a)) \stackrel{\mathcal{L}}{\approx} \mathbf{Unknown}(r)$$

To illustrate that both agents behave indeed identically, we put both agents in a context

$$\mathcal{C}[\cdot] \stackrel{def}{=} (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle \mathit{unknown} = u\rangle) \mid [\cdot])$$

where u is the location of the next interaction for the value *unknown*. We have

$$\begin{array}{ll}
& \mathcal{C}[((\nu\ a)(\mathbf{Not}(a, r) \mid \mathbf{Unknown}(a)))] \\
= & (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle \mathit{unknown} = u\rangle) \mid ((\nu\ a)(\mathbf{Not}(a, r) \mid \mathbf{Unknown}(a)))) \\
\equiv & (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle \mathit{unknown} = u\rangle) \\
& \qquad \mid ((\nu\ a)(a(Y).((\nu\ b)(\overline{r}(\langle val = b\rangle) \\
& \qquad\qquad \mid !b(X).\overline{Y_{val}}(X\langle \mathit{true} = X_{\mathit{false}}\rangle\langle \mathit{false} = X_{\mathit{true}}\rangle)))) \\
& \qquad\quad \mid (\nu\ b)(\overline{a}(\langle val = b\rangle) \mid !b(X).\overline{X_{unknown}})))) \\
\rightarrow & (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle \mathit{unknown} = u\rangle) \\
& \qquad \mid ((\nu\ b)((\nu\ b')(\overline{r}(\langle val = b'\rangle) \\
& \qquad\qquad \mid !b'(X).\overline{b}(X\langle \mathit{true} = X_{\mathit{false}}\rangle\langle \mathit{false} = X_{\mathit{true}}\rangle))) \\
& \qquad\quad \mid !b(X).\overline{X_{unknown}}))) \\
\rightarrow & (\nu\ u)(\nu\ b')(\overline{b'}(\langle \mathit{unknown} = u\rangle)
\end{array}$$

$$
\begin{array}{rl}
 & \quad \mid ((\nu\ b)(!b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \qquad \mid !b(X).\overline{X_{unknown}}))) \\
\equiv & (\nu\ u)(\nu\ b')(\overline{b'}(\langle unknown = u\rangle) \\
 & \quad \mid ((\nu\ b)(!b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \qquad \mid b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \qquad \mid !b(X).\overline{X_{unknown}}))) \\
\rightarrow & (\nu\ u)(\nu\ b')(\nu\ b)(!b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \quad \mid \overline{b}((\langle unknown = u\rangle)\langle true = \mathcal{E}\rangle\langle false = \mathcal{E}\rangle) \\
 & \quad \mid !b(X).\overline{X_{unknown}}) \\
\equiv & (\nu\ u)(\nu\ b')(\nu\ b)(!b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \quad \mid \overline{b}((\langle unknown = u\rangle)\langle true = \mathcal{E}\rangle\langle false = \mathcal{E}\rangle) \\
 & \quad \mid b(X).\overline{X_{unknown}} \mid !b(X).\overline{X_{unknown}}) \\
\rightarrow & (\nu\ u)(\nu\ b')(\nu\ b)(!b'(X).\overline{b}(X\langle true = X_{false}\rangle\langle false = X_{true}\rangle) \\
 & \quad \mid \overline{u} \mid !b(X).\overline{X_{unknown}}) \\
\overset{\mathcal{L}}{\sim} & \overline{u}
\end{array}
$$

and

$$
\begin{array}{rl}
 & \mathcal{C}[\mathbf{Unknown}(r)] \\
= & (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle unknown = u\rangle) \mid \mathbf{Unknown}(r)) \\
\equiv & (\nu\ r)(\nu\ u)(r(X).\overline{X_{val}}(\langle unknown = u\rangle) \mid (\nu\ b)(\overline{r}(\langle val = b\rangle) \mid !b(X).\overline{X_{unknown}})) \\
\rightarrow & (\nu\ b)(\nu\ u)(\overline{b}(\langle unknown = u\rangle) \mid !b(X).\overline{X_{unknown}}) \\
\equiv & (\nu\ b)(\nu\ u)(\overline{b}(\langle unknown = u\rangle) \mid b(X).\overline{X_{unknown}} \mid !b(X).\overline{X_{unknown}}) \\
\rightarrow & (\nu\ b)(\nu\ u)(\overline{u} \mid !b(X).\overline{X_{unknown}}) \\
\overset{\mathcal{L}}{\sim} & \overline{u}
\end{array}
$$

□

4.2 Syntax of the $\pi\mathcal{L}$-calculus

In the $\pi\mathcal{L}$-calculus we replace the communication of names or tuples of names by communication of so-called *forms*. More precisely, the $\pi\mathcal{L}$-calculus is an offspring of the asynchronous π-calculus [101], where polyadic communication is replaced by monadic communication of forms. Furthermore, we strictly distinguish between constants and variables in the $\pi\mathcal{L}$-calculus.

In the $\pi\mathcal{L}$-calculus names are always constant, i.e. names are constant locations and are not subject of substitution (α-conversion is still possible). On the other side, variables in the $\pi\mathcal{L}$-calculus are represented by so-called *projections*. In fact, projections

are *named formal parameters* that are distributed over a $\pi\mathcal{L}$-term. By explicitly naming the formal parameters we break the position dependency of them and therefore we get a greater flexibility in the $\pi\mathcal{L}$-calculus – extensibility.

4.2.1 Names and forms

The most primitive entity, as in the π-calculus, is a *name*. We use $a, b, c, ...x, y, z$ to range over the set $\mathcal{N}$ of names. As in the π-calculus literature, we use the words "name", "port", and "channel" interchangeably.

Unlike the π-calculus where names are both subject and object of a communication (e.g., in the prefixes '$\overline{y}x$' and '$y(x)$' we say that y is the subject, and x is the object), in the $\pi\mathcal{L}$-calculus names are only used as subject of a communication. The role of the object of a communication is taken by forms. Forms are finite mappings from an infinite set $\mathcal{L}$ of labels to an infinite set $\mathcal{N}^+ = \mathcal{N} \cup \{\mathcal{E}\}$, the set of names extended by $\mathcal{E}$ that denotes the empty binding. We use F, G, H to range over forms, X, Y, Z to range over form variables, and l, m, n to range over $\mathcal{L}$. The syntax for forms is defined as follows:

$$\begin{array}{rcll} F & ::= & X & \textit{form variable} \\ & | & \mathcal{E} & \textit{empty binding} \\ & | & FX & \textit{polymorphic extension} \\ & | & F\langle l{=}V\rangle & \textit{binding extension} \end{array}$$

where

$$\begin{array}{rcll} V & ::= & x & \textit{simple name} \\ & | & X_l & \textit{projection} \end{array}$$

Form variables and *projections* deserve a special attention. In the π-calculus we only have names. A name that occurs as object in an input prefix, for example name y in $x(y).A$, is said to be the location of the place where an actually received value z will go in process A, i.e., in process A name y will be *instantiated* by name z.

In the $\pi\mathcal{L}$-calculus, however, we use form variables as the object part of an input prefix. These form variables are *polymorphic placeholders* for forms and in contrast to the π-calculus, form variables are values and not references. Therefore, form variables cannot be instantiated by the received form. There are simply substituted by the received form.

On the other side, *projections* denote locations of names in the $\pi\mathcal{L}$-calculus. In fact, projections are *named formal process parameters* which can be distributed over a $\pi\mathcal{L}$-term. A projection X_l has to be read as selection of the parameter named by l. Moreover, if X_l occurs in a process $a(X).A$, then X_l will be instantiated by name z if X_l maps to z.

As consequence, unlike in the π-calculus where *name instantiation* (or the substitution of names to names) is done in one step in $\pi\mathcal{L}$ we need two: first we must substitute all form variables X in A for some received form value F, then all projections X_l in A can be instantiated (or substituted) to the name denoted by X_l. We treat, however, form substitution and name projection (instantiation of names) as one atomic action. In other words, if a form variable X in X_l has been substituted by a received form value F, then we immediately perform the name projection on the resulting F_l yielding the name that projection F_l maps. Therefore, in the $\pi\mathcal{L}$-calculus, we say that a $\pi\mathcal{L}$-term is instantiated rather than that names of a term are instantiated.

To formulate, how projection works, we need the notion of *variables of a form* and *closed forms*.

Definition 4.1 (Variables of a form) *The set of variables of a form F, written $\mathcal{V}(F)$, is defined as:*

$$\begin{aligned}
\mathcal{V}(\mathcal{E}) &= \emptyset \\
\mathcal{V}(X) &= \{X\} \\
\mathcal{V}(FX) &= \{X\} \cup \mathcal{V}(F) \\
\mathcal{V}(F\langle l{=}x\rangle) &= \mathcal{V}(F) \\
\mathcal{V}(F\langle l{=}X_k\rangle) &= \{X\} \cup \mathcal{V}(F)
\end{aligned}$$

Definition 4.2 (Closed form) *We say that a form F is closed if it does not contain any form variable, so that $\mathcal{V}(F) = \emptyset$.*

Throughout this thesis we shall use the following abbreviation to denote a closed form:

$$\widetilde{\langle l{=}b\rangle} = \begin{cases} \langle l_1{=}b_1\rangle ... \langle l_n{=}b_n\rangle, \ \textit{for } n \ \geq 1, \ b_1, ..., b_n \in \mathcal{N}^+ \\ \mathcal{E}, \ \textit{for } n \ = 0 \end{cases}$$

Now we can define the notion of *name projection* as follows.

Definition 4.3 (Name projection) *If a form F is closed, then the application of a label $l \in \mathcal{L}$ to form F (mapping from $\mathcal{L}$ to $\mathcal{N}^+$), written F_l, is called name projection and is defined as:*

$$\begin{aligned}
\mathcal{E}_l &= \mathcal{E} \\
(F\langle l{=}x\rangle)_l &= x \\
(F\langle m{=}x\rangle)_l &= F_l \quad \textit{if } m \neq l
\end{aligned}$$

If a binding is defined for label l then F_l yields a, otherwise it yields the empty binding ($\mathcal{E}$). A form may have multiple bindings for label l. In this case F_l extracts the rightmost binding. This allows an agent to overwrite a binding with a new one, preserving the old form.

In the following we define the set of *names* and *labels* of a form, and the equivalence over forms.

Definition 4.4 (Names of a form) *The set of names of a form F, written $\mathcal{N}(F)$, is defined as:*

$$\begin{aligned} \mathcal{N}(X) = \mathcal{N}(\mathcal{E}) &= \emptyset \\ \mathcal{N}(FX) &= \mathcal{N}(F) \\ \mathcal{N}(F\langle l{=}x\rangle) &= \{a\} \cup \mathcal{N}(F) \\ \mathcal{N}(F\langle l{=}X_k\rangle) &= \mathcal{N}(F) \end{aligned}$$

Definition 4.5 (Labels of a form) *The set of labels of a form F, written $\mathcal{L}(F)$, is defined as:*

$$\begin{aligned} \mathcal{L}(X) &= \emptyset \\ \mathcal{L}(\mathcal{E}) &= \emptyset \\ \mathcal{L}(FX) &= \mathcal{L}(F) \\ \mathcal{L}(F\langle l{=}V\rangle) &= \{l\} \cup \mathcal{L}(F) \end{aligned}$$

Definition 4.6 (Equivalence of forms) *Let F and G are closed forms. Then two forms F and G are equivalent, written $F \equiv G$, if for all $l \in \mathcal{L}(F) \cup \mathcal{L}(G)$ it holds:*

$$F_l = G_l$$

Lemma 4.1 *Let $F \equiv G$. Then for all $l \in \mathcal{L}$ it holds $F_l = G_l$.* □

Using definition 4.6 and lemma 4.1 it is always possible to replace a form F, which has multiple bindings for some label $l \in \mathcal{L}(F)$ with a equivalent form G with pairwise distinct labels.

4.2.2 The language

The class $\mathcal{A}$ of $\pi\mathcal{L}$-calculus agents is built using the operators of inaction, input prefix, output, parallel composition, restriction, and replication. We use A, B, C to range over the class of agents. The syntax for agents is defined as follows:

$$\begin{array}{rcll} A & ::= & \mathbf{0} & \textit{inactive agent} \\ & | & A \mid A & \textit{parallel composition} \\ & | & (\nu\ a)A & \textit{restriction} \\ & | & V(X).A & \textit{input (receive form in } X\textit{)} \\ & | & \overline{V}(F) & \textit{output (send form } F\textit{)} \\ & | & !V(X).A & \textit{replication} \end{array}$$

0 is the inactive agent. An input-prefixed agent $V(X).A$ waits for a form F to be sent along channel denoted by value V and then behaves like $A\{F/X\}$, where $\{F/X\}$ is the substitution of all form variables X with form F. An output $\overline{V}(F)$ emits a form F along the channel denoted by value V. Unlike in the π-calculus, the value V in both the input prefix and the output particle can be either a simple name or a projection. Parallel composition runs two agents in parallel. The restriction $(\nu\ a)A$ makes name a local to A, i.e., creates a *fresh* name a with scope A. A replication $!V(X).A$ stands for a countably infinite number of copies of $V(X).A$ in parallel.

4.2.3 A reference cell example

The following example presents the encoding of a *reference cell* based on the basic object model of Pierce and Turner [91] in $\pi\mathcal{L}$. This encoding shows that $\pi\mathcal{L}$ provides a compact formalism which can model records, and is therefore appropriate for handling record-based objects.

$$\begin{array}{l} !\mathit{NewRefCell}(X).(\nu\ \mathit{contents})(\nu\ s)(\nu\ g) \\ \qquad (\ \overline{X_{\mathit{reply}}}(\langle \mathit{set}=s\rangle\langle \mathit{get}=g\rangle) \\ \qquad \mid \overline{\mathit{contents}}(\langle \mathit{val}=X_{\mathit{init}}\rangle) \\ \qquad \mid !s(Y).\mathit{contents}(Z).(\overline{Y_{\mathit{reply}}} \mid \overline{\mathit{contents}}(\langle \mathit{val}=Y_{\mathit{val}}\rangle)) \\ \qquad \mid !g(Y).\mathit{contents}(Z).(\overline{Y_{\mathit{reply}}}(\langle \mathit{val}=Z_{\mathit{val}}\rangle) \mid \overline{\mathit{contents}}(Z))\ ) \end{array}$$

The agent listening on channel *NewRefCell* implements an object generator [26] which yields a new object, if we send a form containing at least a binding for X_{reply} along channel *NewRefCell*. Access to the new object is returned along the channel denoted by X_{reply}. The form $\langle \mathit{set}=s\rangle\langle \mathit{get}=g\rangle$ implements the interface to the newly created object. The methods of the object are implemented by the agents listening on channels s (set method) and g (get method). The agents sending/listening along channel *contents* implement the state of the object. The state and the method implementations are local to the object.

4.2.4 Binders and substitution

Both the input prefix and the restriction operator are binders for names in the π-calculus. In the $\pi\mathcal{L}$-calculus, however, only the operator $(\nu\ a)A$ acts as binder for names occurring free in an agent. In $\pi\mathcal{L}$ the input prefix $V(X)$ is the binding operator for form variables. We use $\mathrm{fn}(A)$ and $\mathrm{bn}(A)$ to denote the set of *free* and *bound names* of an agent and $\mathrm{fv}(A)$ and $\mathrm{bv}(A)$ to denote the set of *free* and *bound form variables* of an agent, respectively.

Definition 4.7

(i) The set of free names of an agent A, written $\mathrm{fn}(A)$*, is inductively given by:*

$$\begin{aligned}
\mathrm{fn}(\mathbf{0}) &= \emptyset,\\
\mathrm{fn}(A_1 \mid A_2) &= \mathrm{fn}(A_1) \cup \mathrm{fn}(A_2),\\
\mathrm{fn}((\nu\ a)A) &= \mathrm{fn}(A) - \{a\},\\
\mathrm{fn}(!a(X).A) = \mathrm{fn}(a(X).A) &= \{a\} \cup \mathrm{fn}(A),\\
\mathrm{fn}(!Y_l(X).A) = \mathrm{fn}(Y_l(X).A) &= \mathrm{fn}(A),\\
\mathrm{fn}(\overline{a}(F)) &= \{a\} \cup \mathcal{N}(F),\\
\mathrm{fn}(\overline{Y_l}(F)) &= \mathcal{N}(F).
\end{aligned}$$

(ii) The set of bound names of an agent A, written $\mathrm{bn}(A)$*, is inductively given by:*

$$\begin{aligned}
\mathrm{bn}(\mathbf{0}) &= \emptyset,\\
\mathrm{bn}(A_1 \mid A_2) &= \mathrm{bn}(A_1) \cup \mathrm{bn}(A_2),\\
\mathrm{bn}((\nu\ a)A) &= \{a\} \cup \mathrm{bn}(A),\\
\mathrm{bn}(!V(X).A) = \mathrm{bn}(V(X).A) &= \mathrm{bn}(A),\\
\mathrm{bn}(\overline{V}(F)) &= \emptyset.
\end{aligned}$$

(iii) The set of names of an agent A, written $\mathrm{n}(A)$*, is given by* $\mathrm{n}(A) = \mathrm{fn}(A) \cup \mathrm{bn}(A)$.

(iv) The set of free variables of an agent A, written $\mathrm{fv}(A)$*, is inductively given by:*

$$\begin{aligned}
\mathrm{fv}(\mathbf{0}) &= \emptyset,\\
\mathrm{fv}(A_1 \mid A_2) &= \mathrm{fv}(A_1) \cup \mathrm{fv}(A_2),\\
\mathrm{fv}((\nu\ a)A) &= \mathrm{fv}(A),\\
\mathrm{fv}(!a(X).A) = \mathrm{fv}(a(X).A) &= \mathrm{fv}(A) - \{X\},\\
\mathrm{fv}(!Y_l(X).A) = \mathrm{fv}(Y_l(X).A) &= (\{Y\} \cup \mathrm{fv}(A)) - \{X\},\\
\mathrm{fv}(\overline{a}(F)) &= \mathcal{V}(F),\\
\mathrm{fv}(\overline{Y_l}(F)) &= \{Y\} \cup \mathcal{V}(F).
\end{aligned}$$

(v) The set of bound variables of an agent A, written $\mathrm{bv}(A)$*, is inductively given by:*

$$\begin{aligned}
\mathrm{bv}(\mathbf{0}) &= \emptyset\\
\mathrm{bv}(A_1 \mid A_2) &= \mathrm{bv}(A_1) \cup \mathrm{bv}(A_2),\\
\mathrm{bv}((\nu\ a)A) &= \mathrm{bv}(A),\\
\mathrm{bv}(!V(X).A) = \mathrm{bn}(V(X).A) &= \{X\} \cup \mathrm{bv}(A),\\
\mathrm{bv}(\overline{V}(F)) &= \emptyset.
\end{aligned}$$

(vi) The set of variables of an agent A, written $\mathrm{v}(A)$*, is given by* $\mathrm{v}(A) = \mathrm{fv}(A) \cup \mathrm{bv}(A)$.

The sets $\mathrm{fv}(A)$ and $\mathrm{bv}(A)$ give rise the the following definition.

Definition 4.8 (Closed agent) *We say that an agent A is closed if it does not contain any free form variable, so that* $\mathrm{fv}(A) = \emptyset$.

We write $A\{F/X\}$ for the substitution of all free occurrences of form variable X with form F in A. We use σ to range over form substitutions. Substitutions have precedence over the operators of the language.

Definition 4.9 (Form substitution) *Let $\sigma = \{F/X\}$ and F be closed. Then the effect of the substitution σ on the agent A, written $A\sigma$, is defined inductively below. To avoid that free names of F become accidentally bound in $A\sigma$ (underneath a restriction operator) we assume that the conflicting names in A have been previously α-converted to fresh names, s.t.* $\mathrm{bn}(A) \cap \mathcal{N}(F) = \emptyset$.

$$
\begin{array}{rcl}
\mathbf{0}\sigma & = & 0 \\
(A_1 \mid A_2)\sigma & = & (A_1\sigma) \mid (A_2\sigma) \\
(!V(X).A)\sigma & = & !(V(X).A)\sigma \\
((\nu\ a)A)\sigma & = & (\nu\ a)(A\sigma) \\
(V(X).A)\sigma & = & (V\sigma)(X).A \\
(V(Y).A)\sigma & = & (V\sigma)(Y).(A\sigma),\ Y \neq X \\
(\overline{V}(G))\sigma & = & \overline{(V\sigma)}(G\sigma)
\end{array}
$$

with

$$
V\sigma = \begin{cases} F_l, \ \textit{if}\ V = X_l \\ V, \ \textit{otherwise} \end{cases} \quad \textit{and} \quad G\sigma = \begin{cases} F, \ \textit{if}\ G = X \\ \mathcal{E}, \ \textit{if}\ G = \mathcal{E} \\ (H\sigma)\langle l{=}V\sigma\rangle, \ \textit{if}\ G = H\langle l{=}V\rangle \end{cases}
$$

In $A\{F/X\}$, by definition F must be closed. Therefore, each time the substitution yields a projection F_l, the projection is immediately replaced by the result of F_l – a simple name or $\mathcal{E}$. We say that a form substitution $A\sigma$ simultaneously substitutes all free occurrences of form variable X by form value F in A and all projections X_l in A by name a if X_l maps a. Using this approach we avoid to introduce a (lazy) "trigger" operation that instantiates, when needed, each X_l in A to its corresponding name (for example, if X_l occurs as the subject of an outermost input prefix or output particle).

Now, in the $\pi\mathcal{L}$-calculus we strictly distinguish between constants and variables. In $\pi\mathcal{L}$, variables are represented by form variables and projections, respectively. The following lemma states that we can identify the agents A and $A\sigma$ if A is closed. In other words, $\mathrm{fn}(A) = \mathrm{fn}(A\sigma)$ and $\mathrm{bn}(A) = \mathrm{bn}(A\sigma)$ if $\mathrm{fv}(A) = \emptyset$.

Lemma 4.2 *For every agent A and $\sigma = \{F/X\}$ with $\mathrm{fv}(A) = \emptyset$, it holds that*

$$A\sigma = A$$

PROOF: By induction on the structure of A. □

Unlike in the π-calculus, there is no general *name substitution* in the $\pi\mathcal{L}$-calculus. Therefore, we have to define explicitly *alpha-conversion* of bound names and bound variables, respectively. We write $A\{\tilde{y}/\tilde{x}\}_\alpha^{\mathcal{N}}$ for the alpha-substitution of bound names in agent A and $A\{Y/X\}_\alpha^{\mathcal{V}}$ for the alpha-substitution of bound variables in agent A. We use $\alpha^{\mathcal{N}}$ or $\alpha^{\mathcal{V}}$ to range over alpha-substitution. Like form substitution, alpha-substitutions have precedence over the operators of the language.

Definition 4.10 (α-substitution) *Let $\alpha^{\mathcal{N}} = \{\tilde{y}/\tilde{x}\}_\alpha^{\mathcal{N}}$ and $\alpha^{\mathcal{V}} = \{Y/X\}_\alpha^{\mathcal{V}}$. Then the effect of the substitution $\alpha^{\mathcal{N}/\mathcal{V}}$ on the agent A, written $A\alpha^{\mathcal{N}/\mathcal{V}}$, is defined inductively below.*

$$\begin{aligned}
\mathbf{0}\alpha^{\mathcal{N}/\mathcal{V}} &= \mathbf{0} \\
(A_1 \mid A_2)\alpha^{\mathcal{N}/\mathcal{V}} &= (A_1\alpha^{\mathcal{N}/\mathcal{V}}) \mid (A_2\alpha^{\mathcal{N}/\mathcal{V}}) \\
(!V(X).A)\alpha^{\mathcal{N}/\mathcal{V}} &= !(V(X).A)\alpha^{\mathcal{N}/\mathcal{V}} \\
((\nu\ a)A)\alpha^{\mathcal{N}} &= (\nu\ a')(A\alpha^{\mathcal{N}}),\ a' = \begin{cases} a, \text{ if } a \notin \tilde{x} \\ y_i, \text{ if } a = x_i, \text{ and } x_i\alpha^{\mathcal{N}} = y_i \end{cases} \\
((\nu\ a)A)\alpha^{\mathcal{V}} &= (\nu\ a)(A\alpha^{\mathcal{V}})
\end{aligned}$$

$$\begin{aligned}
(V(X).A)\alpha^{\mathcal{N}} &= (V\alpha^{\mathcal{N}})(X).A\alpha^{\mathcal{N}} \\
(V(Z).A)\alpha^{\mathcal{V}} &= (V\alpha^{\mathcal{V}})(Z').A\alpha^{\mathcal{V}},\ Z' = \begin{cases} Z, \text{ if } Z \neq X \\ Y, \text{ if } Z = X, \text{ and } X\alpha^{\mathcal{V}} = Y \end{cases} \\
(\overline{V}(F))\alpha^{\mathcal{N}/\mathcal{V}} &= \overline{(V\alpha^{\mathcal{N}/\mathcal{V}})}(F\alpha^{\mathcal{N}/\mathcal{V}})
\end{aligned}$$

with

$$V\alpha^{\mathcal{N}} = \begin{cases} y_i, \text{ if } V = x_i, \text{ and } x_i\alpha^{\mathcal{N}} = y_i \\ V, \text{ otherwise} \end{cases},\ V\alpha^{\mathcal{V}} = \begin{cases} Y_l, \text{ if } V = X_l \text{ and } X\alpha^{\mathcal{V}} = Y \\ V, \text{ otherwise} \end{cases}$$

and

$$F\alpha^{\mathcal{N}} = \begin{cases} (G\alpha^{\mathcal{N}})\langle l{=}V\alpha^{\mathcal{N}}\rangle, \text{ if } F = G\langle l{=}V\rangle \\ F, \text{ otherwise} \end{cases}$$

$$F\alpha^{\mathcal{V}} = \begin{cases} GY, \text{ if } F = GX \text{ and } X\alpha^{\mathcal{V}} = Y \\ (G\alpha^{\mathcal{V}})\langle l{=}V\alpha^{\mathcal{V}}\rangle, \text{ if } F = G\langle l{=}V\rangle \\ F, \text{ otherwise} \end{cases}$$

Finally, we adopt the usual convention of writing $x(X)$ when we mean $x(X).\mathbf{0}$. Additionally, an agent $\overline{x}(\mathcal{E})$ sending an empty form can just be written $\overline{x}$, a form $\langle \mathcal{E}\langle l = x\rangle\rangle$ is just written $\langle l = x\rangle$, and we abbreviate $(\nu\ x)(\nu\ y)A$ with $(\nu\ x, y)A$ and $(\nu\ x_1)...(\nu\ x_n)A$ with $(\nu\ \tilde{x})A$, respectively.

4.3 Operational semantics

The operational semantics of a process algebra is traditionally given in terms of a *labelled transition system* describing the possible evolution of a process. This contrasts with the semantic definition in *term rewriting systems* where an *unlabelled reduction system* is used. The best known term rewriting system is probably the λ-calculus. In the λ-calculus, the reduction of two interacting subterms is only possible if they are in a contiguous position. In process calculi, however, interaction does not depend on a physical contiguity. In other words, in the λ-calculus a *redex* denotes a subterm of a λ-term while a "redex" in a process calculus is usually distributed over the term.

Recently, Milner [63, 64] has proposed a guideline for the definition of a reduction system for process algebras. This proposal was inspired by Berry and Boudol's *Chemical Abstract Machine* [11]. Honda and Yoshida [45] have also used this scheme to formulate their semantic theories for processes. Using the reduction system technique, axioms for a structural congruence relation are introduced prior the definition of the reduction relation. Basically, this allows us to separate the laws which govern the neighbourhood relation among processes from the rules that specify their interaction. Furthermore, this simplifies the presentation of the reduction relation by reducing the number of cases that we have to consider.

The reduction semantics defines the basic mechanisms of computation in a process calculus. The interpretation of the operators is precisely described using the reduction semantics. The reduction relation, however, covers only a part the behaviour of processes; it describes the behaviour of processes relative to a context in which they are contained. In other words, the reduction semantics describes how a process may interact with another, but not how this process (or parts of it) may interact with the environment. Therefore, the reduction relation defines the interaction of processes, i.e., their local evolution.

A labelled transition system describes the possible intraactions of processes with the environment. With labelled transition semantics every possible communication of a process can be determined in a direct way. This allows us to get a simple characterizations of behavioural equivalences. Moreover, with labelled transition semantics proofs benefit from reasoning in a purely structural way.

To make a transition means that a process P can evolve into a process Q, and in doing so perform the action μ. An external experimenter or observer will be able to observe the process evolution while taking the role of the environment. A special

(1) $A \mid B \equiv B \mid A,\ (A \mid B) \mid C \equiv A \mid (B \mid C),\ A \mid \mathbf{0} \equiv A;$

(2) $(\nu\ x)\mathbf{0} \equiv \mathbf{0},\ (\nu\ x)(\nu\ y)A \equiv (\nu\ y)(\nu\ x)A;$

(3) $(\nu\ x)A \mid B \equiv (\nu\ x)(A \mid B)$, if x not free in B;

(4) $!V(X).A \equiv V(X).A \mid !V(X).A;$

(5) $\mathcal{E}(X).A \equiv \mathbf{0},\ \overline{\mathcal{E}}(F) \equiv \mathbf{0}.$

Table 4.1: Structural congruence rules for the $\pi\mathcal{L}$-calculus.

action τ denotes interaction or silent action. Roughly spoken, transitions labelled with τ correspond to the plain reduction relation.

In general, it is not easy to define a labelled transition system. The manipulation of names and the side conditions in the rules are non-trivial. On the other side, if the reduction system is available, the corresponding labelled transition system can be found. Furthermore, by showing the correspondence of both the reduction system and the labelled transition system it is possible to prove the correctness of the latter.

4.3.1 Reduction semantics

The structural congruence relation, $\equiv$, is the smallest congruence relation over agents that satisfies the axioms given in Table 4.1.

The axioms (1)–(4) are standard and are the same as for the π-calculus. The only "new" axiom is (5), which defines the behaviour if an *empty binding* appears in subject position of the leftmost prefix of an agent. In this case the agent is identical with the *inactive agent*. This means a system containing such an agent may reach a deadlock.

In general, if the name $\mathcal{E}$ occurs as subject in the leftmost prefix of an agent, this may be interpreted as a run-time error. However, this is too restrictive, in the sense that this view excludes some programs that may be useful in some contexts.

In [28, 29] Dami has identified a similar problem with the λ-calculus. Dami proposed a liberal approach to errors for the λ-calculus. An error, written ε, can be passed around as any other value. Using a lazy evaluation strategy, an error occurring inside a term is not necessarily propagated to the top level. A term is considered to be "erroneous" if and only if it always generates ε after a finite number of interactions with its context.

Now, in the $\pi\mathcal{L}$-calculus the assembly of agents may be partially incorrect, but an error in the assembled agent can be tolerated as long as there are contexts which

$$\text{PAR: } \frac{A \longrightarrow A'}{A \mid B \longrightarrow A' \mid B} \qquad \text{RES: } \frac{A \longrightarrow A'}{(\nu\ x)A \longrightarrow (\nu\ x)A'}$$

$$\text{COM: } x(X).A \mid \overline{x}.(F) \longrightarrow A\{F/X\},\ \textit{if}\ \mathcal{V}(F) = \emptyset$$

$$\text{STRUCT: } \frac{A \equiv A' \quad A' \longrightarrow B' \quad B' \equiv B}{A \longrightarrow B}$$

Table 4.2: Reduction system for the $\pi\mathcal{L}$-calculus.

can use it without reaching the error. Note that usual type systems will reject such assemblies as soon as they detect a potential error.

Using a process calculi, an error corresponds to the fact that a process system cannot further reduce and has reached a deadlock, respectively. The axiom (5) represents exactly this interpretation of error in a process calculus. We simply say, if we have a form that does not contain a specific binding, then the agent requesting this binding cannot further evolve – the agent blocks and therefore becomes inactive.

A programming language based on the $\pi\mathcal{L}$-calculus may use a different approach. If this language provides an exception handling construct, then sending or receiving along name $\mathcal{E}$ can raise an exception. In this case, an agent would not become inactive. Moreover, the system can provide the programmer with a meaningful message.

Table 4.2 describes the reduction of $\pi\mathcal{L}$-terms. In fact, the *reduction* rules define the interaction of $\pi\mathcal{L}$-agents.

The first two rules state that we can reduce under both parallel composition and restriction. (The symmetric rule for parallel composition is redundant, because of the use of structural congruence.)

The communication rule takes two agents which are willing to communicate on the channel x, and substitutes all form variables X with form F in A. Communication is only allowed for *closed* forms (side condition $\mathcal{V}(F) = \emptyset$). The communication rule is the only rule which directly reduces a $\pi\mathcal{L}$-term. Furthermore, a reduction is not allowed underneath a input prefix. Prefixing is the construct that allows sequentialization.

The communication assumes that agents are in a particular format. The structural congruence rule allows us to rewrite agents so that they have the correct format for the communication rules.

4.3.2 Labelled transition semantics

The reduction relation defines how agents may interact with each other; it defines the interaction. The intraaction, however, is not covered by the reduction relation. To

define, how agents may interact with the environment, we use a *labelled transition system* that describes the possible interactions with other systems.

From the reduction relation we know that only communication reduces $\pi\mathcal{L}$-terms. To establish communication we need a pair consisting of an output particle and an input prefix, where both use the same name as subject. Then, we naturally have two kinds of potential interaction, *input* and *output*, and they are represented by agents of the form

$$\begin{array}{rll} (\nu\ \tilde{x})(a(X).P \mid ...) & (a \notin \tilde{x}) & \textit{input} \\ (\nu\ \tilde{x})(\overline{a}(\langle l_1{=}b_1\rangle ... \langle l_n{=}b_n\rangle) \mid ...) & (a, b_1, ..., b_n \notin \tilde{x}) & \textit{output} \end{array}$$

But there is another kind of output, where we have an additional restriction, e.g. $(\nu\ \tilde{y})$. This is represented by an agent of the form

$$(\nu\ \tilde{y})(\nu\ \tilde{x})(\overline{a}(\langle l_1{=}b_1\rangle ... \langle l_n{=}b_n\rangle) \mid ...) \quad (a, b_1, ..., b_n \notin \tilde{x}) \qquad \textit{restricted output}$$

In this kind of action it holds that $\tilde{y} \subseteq \mathrm{fn}(\overline{a}(\langle l_1{=}b_1\rangle ... \langle l_n{=}b_n\rangle) - a$. In fact, $\tilde{y}$ represents private names which are emitted from the agent, i.e., carried out from their current scope (*scope extrusion*). We will strictly handle *output* and *restricted output* as different actions.

As in CCS [62] and the π-calculus [65], a transition in the $\pi\mathcal{L}$-calculus is of the form

$$A \xrightarrow{\mu} A'$$

Intuitively, this transition means that A can evolve into A', and in doing so perform the action μ. We use μ to range over *actions* that have the following structure:

$$\begin{array}{rcll} \mu & = & \tau & \text{silent action} \\ & \mid & a(\widetilde{\langle l{=}b\rangle}) & \text{input action} \\ & \mid & \overline{a}(\widetilde{\langle l{=}b\rangle}) & \text{output action} \\ & \mid & (\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle}) & \text{restricted output action} \end{array}$$

In the case of input and output, a is the *subject* part, whereas $\widetilde{\langle l{=}b\rangle}$ is the *object* part of the action.

Input and output describe interactions between an agent A and its environment, while the silent action τ is used as placeholder for an internal action in which one subagent of A communicates with another; an external observer can see that something is happening (time is passing), but nothing more.

We only allow *closed* forms ($\mathcal{V}(\widetilde{\langle l{=}b\rangle}) = \emptyset$) to be object of an output action. Furthermore, we only consider forms with pairwise distinct labels because an external observer can always replace a form with multiple bindings for some label l with an equivalent one, without changing the behaviour of an agent.

In the input action, $a(X).A \xrightarrow{a(\widetilde{\langle l \mapsto b \rangle})} A$ intuitively means that $a(X).A$ can receive any form value $\langle \widetilde{l = b} \rangle$ along port a, and then evolves into $A\{\langle \widetilde{l = b} \rangle / X\}$. Here, X is the form variable that will be substituted by the received form $\langle \widetilde{l = b} \rangle$. Furthermore, all X_l in A are instantiated to $b \in \mathcal{N}(\langle \widetilde{l = b} \rangle)$ if label l maps b or $\mathcal{E}$ otherwise. In fact, we say that agent A is instantiated by $\langle \widetilde{l = b} \rangle$ if agent A evolves $A\{\langle \widetilde{l = b} \rangle / X\}$.

The prefix $(\nu\ \tilde{x})$ in a *restricted output action* is used to record those names in $\langle \widetilde{l = b} \rangle$ that have been created fresh in A (i.e., $\tilde{x} \cap \mathrm{n}(A) = \emptyset$) and are not yet known to the environment. We always have that $\tilde{x} \subseteq \mathcal{N}(\langle \widetilde{l = b} \rangle)$. Intuitively, $A \xrightarrow{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l \mapsto b \rangle})} A'$ means that A emits private names (i.e., names bound in A) along port a. If some names $\tilde{x}$ are communicated outside of the scope of the ν that binds it, the ν must be moved outwards to include both the sender and the receiver – this is known as *scope extrusion*. To avoid capture of bound names in the receiver, this operation may require a α-conversion of bound names in the receiver.

The silent action, the input action, and the output actions will collectively be called *free* actions, while the restricted output actions will be called *bound* actions. Bound actions carry bound names, i.e. names which cannot be known to the target of the message.

Given an action μ, the bound and free names of μ, written $\mathrm{bn}(\mu)$ and $\mathrm{fn}(\mu)$ are defined as follows:

$$\begin{array}{rcl@{\quad,\quad}rcl}
\mathrm{bn}(a(\langle \widetilde{l = b} \rangle)) & = & \emptyset & \mathrm{fn}(a(\langle \widetilde{l = b} \rangle)) & = & \{a\} \cup \mathcal{N}(\langle \widetilde{l = b} \rangle) \\
\mathrm{bn}(\overline{a}(\langle \widetilde{l = b} \rangle)) & = & \emptyset & \mathrm{fn}(\overline{a}(\langle \widetilde{l = b} \rangle)) & = & \{a\} \cup \mathcal{N}(\langle \widetilde{l = b} \rangle) \\
\mathrm{bn}((\nu\ \tilde{x})\overline{a}(\langle \widetilde{l = b} \rangle)) & = & \{\tilde{x}\} & \mathrm{fn}((\nu\ \tilde{x})\overline{a}(\langle \widetilde{l = b} \rangle)) & = & (\{a\} \cup \mathcal{N}(\langle \widetilde{l = b} \rangle)) - \{\tilde{x}\} \\
\mathrm{bn}(\tau) & = & \emptyset & \mathrm{fn}(\tau) & = & \emptyset
\end{array}$$

The names of μ, written $\mathrm{n}(\mu)$, are $\mathrm{bn}(\mu) \cup \mathrm{fn}(\mu)$.

The $\pi\mathcal{L}$-calculus (standard) *early* transition system is presented in Table 4.3. By *early* we mean, when inferring an action from $a(X).A$ then the variable X is instantiated at the time of inferring the input transition (rule IN). This allows us to define bisimulation without clauses for name-instantiation. By instantiation we mean the mechanism that substitutes X first and then applies all projections X_l for some label l. We have omitted the symmetric versions of rules PAR and COM. Indeed, parallel composition should be understood as commutative operator.

4.4 Observable equivalence of $\pi\mathcal{L}$-terms

An important question in the theory of process calculi is when to processes can be said to exhibit the same behaviour. As in the λ-calculus, the most intuitive way of

$$\text{IN} : a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\} \qquad \text{OUT} : \overline{a}(\widetilde{\langle l=b\rangle}) \xrightarrow{\overline{a}(\widetilde{\langle l=b\rangle})} \mathbf{0}$$

$$\text{OPEN} : \frac{A \xrightarrow{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})} A' \qquad y \neq a \qquad y \in \mathcal{N}(\widetilde{\langle l=b\rangle}) - \tilde{x}}{(\nu\ y)A \xrightarrow{(\nu\ y,\tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})} A'}$$

$$\text{COM} : \frac{A \xrightarrow{\overline{a}(\widetilde{\langle l=b\rangle})} A' \qquad B \xrightarrow{a(\widetilde{\langle l=b\rangle})} B'}{A \mid B \xrightarrow{\tau} A' \mid B'}$$

$$\text{CLOSE} : \frac{A \xrightarrow{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})} A' \qquad B \xrightarrow{a(\widetilde{\langle l=b\rangle})} B' \qquad \tilde{x} \notin \text{fn}(B)}{A \mid B \xrightarrow{\tau} (\nu\ \tilde{x})(A' \mid B')}$$

$$\text{PAR} : \frac{A \xrightarrow{\mu} A' \qquad \text{bn}(\mu) \cap \text{fn}(B) = \emptyset}{A \mid B \xrightarrow{\mu} A' \mid B} \qquad \text{RES} : \frac{A \xrightarrow{\mu} A' \qquad x \notin \text{n}(\mu)}{(\nu\ x)A \xrightarrow{\mu} (\nu\ x)A'}$$

$$\text{REPL} : \frac{a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\}}{!a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}$$

Table 4.3: Labelled transition system for the $\pi\mathcal{L}$-calculus.

defining an equivalence of processes is via some notion of *contextual equivalence*. A *process context* $\mathcal{C}[\cdot]$ is a process expression with a hole into which one can place another process. We say that the processes A and B are equivalent when $\mathcal{C}[A]$ and $\mathcal{C}[B]$ have the same "observable behaviour" for each process context $\mathcal{C}[\cdot]$.

However, the definition of a contextual equivalence between two processes can be difficult to establish. Fortunately, there is an alternative to contextual equivalence which is based on the direct conditions on the processes themselves. Given a labelled transition semantics there is a standard definition of bisimulation equivalence [85, 62] which can be applied to this transition system. Moreover, bisimulation equivalence is widely considered to be the finest equivalence one need to study for transition systems.

Basically, bisimulation defines equivalence as mutual simulation of transitions of processes resulting in equivalent states. Formally, a binary relation $\mathcal{R}$ is a (ground) bisimulation on processes such that $A\ \mathcal{R}\ B$ implies, for arbitrary action μ

(i) whenever $A \xrightarrow{\mu} A'$, then B' exists such that $B \xrightarrow{\mu} B'$ and $A'\ \mathcal{R}\ B'$

(ii) whenever $B \xrightarrow{\mu} B'$, then A' exists such that $A \xrightarrow{\mu} A'$ and $A' \mathcal{R} B'$.

The central idea of bisimulation is that an external observer performs experiments with both processes A and B observing the results in turn in order to match each others process behaviour step-by-step. Furthermore, the definition of bisimulation is given in a *coinductive* style that is, two processes are bisimular if we cannot show that they are not.

Checking the equivalence of processes this way one can think of this as a game played between two persons, the "unbeliever", who thinks that A and B are not equivalent, and the "believer", who thinks that A and B are equivalent. The underlying strategy of this game is that the unbeliever is trying to perform a process transition which cannot be matched by the believer. The unbeliever loses if there are no transitions left for either processes whereas the believer loses if he cannot match a move made by the unbeliever.

In terms of experiments this means, in an output-experiment, an external observer tries to receive a message from the process which is possible if the process has a matching output transition $A \xrightarrow{\overline{a}(\widetilde{\langle l \rightharpoonup b \rangle})} A'$ or $A \xrightarrow{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l \rightharpoonup b \rangle})} A'$. In an input-experiment the observer tries to send a message to the process which only succeeds if the process has a matching input transition $A \xrightarrow{a(\widetilde{\langle l \rightharpoonup b \rangle})} A'$. Finally, an external observer may notice that time is passing while the process performs an silent transition $A \xrightarrow{\tau} A'$ which is usually called a reduction-experiment.

So far, many variants of bisimulation have been proposed (e.g. early, late, open, and barbed bisimulation [65, 66, 100, 102]). All variants, however, distinguish between a *strong* and a *weak* definition of bisimulation. The difference of both is that in the weak case, arbitrary many silent transactions are regarded as equivalent to a single transitions. Therefore, the weak bisimulation is strictly coarser than the strong bisimulation, in the sense that whenever two processes A and B are strongly bisimular, they are also weakly bisimular. In practice, weak bisimulation is often more useful, since we typically want to regard two processes to be equivalent if they have the same *observable* behaviour even if one consumes more time (performs more silent transitions) than the other.

Weak arrows $\Longrightarrow$ denote the reflexive and transitive closure of transitions. We have:

$$\begin{aligned} A \overset{\tau}{\Longrightarrow} A' \quad & \text{iff } A(\xrightarrow{\tau}) * A' \\ A \overset{\mu}{\Longrightarrow} A' \quad & \text{iff } A \overset{\tau}{\Longrightarrow} \cdot \xrightarrow{\mu} \cdot \overset{\tau}{\Longrightarrow} A',\ \mu \neq \tau \end{aligned}$$

4.4.1 Asynchronous interaction

In calculi with synchronous output, the existence of an input transition precisely models the success of an observer's input-experiment. In the synchronous case, input actions

for a process A are only generated if there exists a matching receiver that is enabled inside A. The existence of an input transition such that A evolves to A' reflects precisely the fact that a message offered by the observer has actually been consumed.

The $\pi\mathcal{L}$-calculus is an asynchronous calculus. This implies that the sender of an output message does not know when the message is actually consumed. In other words, at the time of consumption of the message, its sender is not participating in the event anymore. Therefore, an asynchronous observer, in contrast to a synchronous one, cannot directly detect the input actions of the observed agent. Consequently, the $\pi\mathcal{L}$-calculus requires an appropriate semantic framework based on an *asynchronous experimenter* and therefore we need a different notion of input-experiment.

As in the case of the asynchronous π-calculus [43, 13, 4], an asynchronous observer may see indirectly that its messages have been consumed by noticing messages that eventually come back from the process as result of a former input-experiment. Therefore, instead of abandoning input-experiments completely, asynchronous observation captures indirectly input-experiments by performing output-experiments in the context of arbitrary messages.

For the asynchronous π-calculus two different notions of *asynchronous observation* have been proposed. Honda and Tokoro [43, 44, 42] introduced a modified input rule in order to model asynchronous input-experiments explicitly:

$$A \xrightarrow{av} A' \mid \overline{a}v$$

This rule allows that a system can accept an arbitrary message at any time without offering a necessary receptor that can consume it. Therefore, one use the standard technique of performing input-experiments by checking the existence of an input transitions. This approach, however, emphasizes observational behaviour of processes and does not reflect the computational content of processes.

In contrast, Amadio et al. [4] proposed a solution which is based on the standard labelled transition system. Here, the asynchronous style of input-experiments is incorporated into the definition of bisimulation such that inputs of processes have to be simulated only indirectly by observing the output behaviour of a process in context of arbitrary messages. For the $\pi\mathcal{L}$-calculus we follow this approach.

4.4.2 Asynchronous Bisimulation for the $\pi\mathcal{L}$-calculus

Now, in the $\pi\mathcal{L}$-calculus we can define agents like

$$X_l(Y).A \quad \text{or} \quad \overline{X_l}(\langle \widetilde{l=b} \rangle)$$

These agents, however, are not closed. Furthermore, these agents cannot interact with the environment because communication requires that we have a plain name in subject position. Therefore, an observation equivalence in $\pi\mathcal{L}$ will only be established over

closed agents, i.e. we always have a plain name in subject position. This also means that there is no such observation equivalence for open $\pi\mathcal{L}$-agents.

Proposition 4.1 (Closed agents evolve to closed agents) *Let A be a $\pi\mathcal{L}$-agent, $\mathrm{fv}(A) = \emptyset$ and μ be a $\pi\mathcal{L}$-action. Then $A \stackrel{\mu}{\Longrightarrow} A'$ implies $\mathrm{fv}(A') = \emptyset$.*

PROOF: We proceed by induction on the structure of A. We consider the most significant case $A = a(X).A_1$.

Then we have $A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A_1\{\widetilde{\langle l=b\rangle}/X\}$. Furthermore, $\mathrm{fv}(A) = \emptyset$ implies that the set $\mathrm{fv}(A_1)$ can be at least a singleton, i.e., $\mathrm{fv}(A_1) = \{X\}$, since the communication removes the binder for X. By definition $\mathcal{V}(\widetilde{\langle l=b\rangle}) = \emptyset$. Therefore, $A_1\{\widetilde{\langle l=b\rangle}/X\}$ does not add any free form variable such that $\mathrm{fv}(A_1\{\widetilde{\langle l=b\rangle}/X\}) = \emptyset$ as required. □

Definition 4.11 ($\mathcal{L}$-bisimulation) *A binary relation $\mathcal{R}$ over closed agents A and B is a strong $\mathcal{L}$-bisimulation if it is symmetric and $A\ \mathcal{R}\ B$ implies*

- *whenever $A \stackrel{\mu}{\longrightarrow} A'$, where μ is either τ or output with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B) = \emptyset$, then B' exists such that $B \stackrel{\mu}{\longrightarrow} B'$ and $A'\ \mathcal{R}\ B'$*
- *$(A \mid \overline{a}(\widetilde{\langle l=b\rangle}))\ \mathcal{R}\ (B \mid \overline{a}(\widetilde{\langle l=b\rangle}))$ for all messages $\overline{a}(\widetilde{\langle l=b\rangle})$.*

Two agents A and B are strongly bisimular, written $A \stackrel{\mathcal{L}}{\sim} B$, if they are related by some strong bisimulation. The notion of weak $\mathcal{L}$-bisimulation is obtained by replacing strong transitions with weak transitions. We write $\stackrel{\mathcal{L}}{\approx}$ for weak $\mathcal{L}$-bisimulation. Two agents A and B are weakly bisimular, written $A \stackrel{\mathcal{L}}{\approx} B$, if there is a weak $\mathcal{L}$-bisimulation $\mathcal{R}$ with $A\ \mathcal{R}\ B$.

We call $\mathcal{R}$ as above a $\mathcal{L}$-bisimulation. Then both $\stackrel{\mathcal{L}}{\sim}$ and $\stackrel{\mathcal{L}}{\approx}$ are the union of all strong and weak $\mathcal{L}$-bisimulations, respectively. Furthermore, both $\stackrel{\mathcal{L}}{\sim}$ and $\stackrel{\mathcal{L}}{\approx}$ require preservation under parallel composition with an output.

Now, we are mainly interested in comparing $\pi\mathcal{L}$-systems by considering only their "observable" behaviour. This means that we abstract from silent actions. Therefore, we take $\stackrel{\mathcal{L}}{\approx}$ – the observation equivalence – as the main equivalence for the $\pi\mathcal{L}$-calculus.

4.4.3 Congruence of $\stackrel{\mathcal{L}}{\approx}$

As in the asynchronous π-calculus, the lack of summation and matching allows us to establish congruence of $\stackrel{\mathcal{L}}{\approx}$. Furthermore, unlike in the π-calculus, in the $\pi\mathcal{L}$-calculus, names are are always constant, i.e., we do not have name substitution. Therefore, if we have an input-prefixed agent, like $a(X).A$, then only the form variable X is substituted

by a received form $\langle\widetilde{l=b}\rangle$. This substitution does not change any name in A. As a consequence, if $\mathrm{fv}(A) = \emptyset$, we can add an arbitrary number of input prefixes, like $a(X)$ without changing the behaviour of A (see Lemma 4.2).

Proposition 4.2 $\overset{\mathcal{L}}{\approx}$ *is an equivalence relation.*

Proof: Symmetry is by definition, while reflexivity is immediate. The only nontrivial property to show is transitivity. We show that the relation $(\overset{\mathcal{L}}{\approx} \circ \overset{\mathcal{L}}{\approx})$ is a weak $\mathcal{L}$-bisimulation. Suppose that

$$A \overset{\mathcal{L}}{\approx} \circ \overset{\mathcal{L}}{\approx} C$$

Then for some B we have

$$A \overset{\mathcal{L}}{\approx} B \text{ and } B \overset{\mathcal{L}}{\approx} C$$

Consider first the case of τ or *output actions* with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B|C) = \emptyset$:

- Now let $A \overset{\mu}{\Longrightarrow} A'$, μ is either τ or output. Then for some B' we have, since $A \overset{\mathcal{L}}{\approx} B$, $B \overset{\mu}{\Longrightarrow} B'$ and $A' \overset{\mathcal{L}}{\approx} B'$.

- Also since $B \overset{\mathcal{L}}{\approx} C$ we have for some C', $C \overset{\mu}{\Longrightarrow} C'$ and $B' \overset{\mathcal{L}}{\approx} C'$.

Hence $A' \overset{\mathcal{L}}{\approx} \circ \overset{\mathcal{L}}{\approx} C'$. Similarly, if $C \overset{\mu}{\Longrightarrow} C'$ we can find A' such that $A \overset{\mu}{\Longrightarrow} A'$ and $A' \overset{\mathcal{L}}{\approx} \circ \overset{\mathcal{L}}{\approx} C'$.

Consider now the case with composition with output:

- If $A \overset{\mathcal{L}}{\approx} B$, then for all messages $\overline{a}(\langle\widetilde{l=b}\rangle)$ we have by definition $A \mid \overline{a}(\langle\widetilde{l=b}\rangle) \overset{\mathcal{L}}{\approx} B \mid \overline{a}(\langle\widetilde{l=b}\rangle)$.
- Also since $B \overset{\mathcal{L}}{\approx} C$, for all messages $\overline{a}(\langle\widetilde{l=b}\rangle)$ we have by definition $B \mid \overline{a}(\langle\widetilde{l=b}\rangle) \overset{\mathcal{L}}{\approx} C \mid \overline{a}(\langle\widetilde{l=b}\rangle)$.

Hence $A \mid \overline{a}(\langle\widetilde{l=b}\rangle) \overset{\mathcal{L}}{\approx} \circ \overset{\mathcal{L}}{\approx} C \mid \overline{a}(\langle\widetilde{l=b}\rangle)$. Similarly, if we start with C. □

Proposition 4.3 *For any A, B and x,*

$$A \overset{\mathcal{L}}{\approx} B \;\Rightarrow\; (\nu\ x)A \;\overset{\mathcal{L}}{\approx}\; (\nu\ x)B.$$

Proof: We show that the relation

$$\mathcal{R} = \{\ ((\nu\ x)A, (\nu\ x)B)\ \mid\ A \overset{\mathcal{L}}{\approx} B\ \} \cup \overset{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation.

Consider τ or *output actions* with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B) = \emptyset$:

> $(\nu\ x)A \overset{\mu}{\Longrightarrow} (\nu\ x)A'$ is inferred from $A \overset{\mu}{\Longrightarrow} A'$. Since $A \overset{\mathcal{L}}{\approx} B$, this implies $B \overset{\mu}{\Longrightarrow} B'$ with $A' \overset{\mathcal{L}}{\approx} B'$. Then $(\nu\ x)B \overset{\mu}{\Longrightarrow} (\nu\ x)B'$ is the required move, since $((\nu\ x)A', (\nu\ x)B') \in \mathcal{R}$.

Consider *input actions*:

> $(\nu\ x)A \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} (\nu\ x)A'$ is inferred from $A \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$. Since $A \overset{\mathcal{L}}{\approx} B$ and by definition $(A|\overline{a}(\widetilde{\langle l=b\rangle}), B|\overline{a}(\widetilde{\langle l=b\rangle})) \in \mathcal{R}$ for all messages $\overline{a}(\widetilde{\langle l=b\rangle})$, this implies $B \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} B'$ with $A' \overset{\mathcal{L}}{\approx} B'$. Then $(\nu\ x)B \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} (\nu\ x)B'$ is the required move, since $((\nu\ x)A', (\nu\ x)B') \in \mathcal{R}$. □

Proposition 4.4 *For any $A, B,$ and $C,$*

$$A \overset{\mathcal{L}}{\approx} B \ \Rightarrow\ A|C \ \overset{\mathcal{L}}{\approx}\ B|C.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (A|C, B|C)\ \mid\ A \overset{\mathcal{L}}{\approx} B\ \} \cup \overset{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation. Input and output are easy. We only show the case for τ transition. We have two possibilities:

- Using COM, $A|C \overset{\tau}{\Longrightarrow} A'|C'$ is inferred from $A \overset{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ and $C \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$. Since $A \overset{\mathcal{L}}{\approx} B$, this implies $B \overset{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} B'$ with $A' \overset{\mathcal{L}}{\approx} B'$. Then $B|C \overset{\tau}{\Longrightarrow} B'|C'$ with $C \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$ is the required move, since $(A'|C', B'|C') \in \mathcal{R}$. We have a similar reasoning if $A \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ and $C \overset{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$.

- Using CLOSE, $A|C \overset{\tau}{\Longrightarrow} (\nu\ \tilde{x})(A'|C')$ is inferred from $A \overset{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ and $C \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$. Since $A \overset{\mathcal{L}}{\approx} B$, this implies $B \overset{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} B'$ with $A' \overset{\mathcal{L}}{\approx} B'$. Then $B|C \overset{\tau}{\Longrightarrow} (\nu\ \tilde{x})(B'|C')$ with $C \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$ is the required move, since $(A'|C', B'|C') \in \mathcal{R}$. We have a similar reasoning if $A \overset{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ and $C \overset{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$. □

Proposition 4.5 *For any $A, B, a,$ and X,*

$$A \overset{\mathcal{L}}{\approx} B \;\Rightarrow\; a(X).A \;\overset{\mathcal{L}}{\approx}\; a(X).B.$$

PROOF: $a(X).A$ can only move on input. Therefore, in the case of an input action we have $a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\}$. Since $A \overset{\mathcal{L}}{\approx} B$, this implies $a(X).B$ has the same move such that $a(X).B \xrightarrow{a(\widetilde{\langle l=b\rangle})} B\{\widetilde{\langle l=b\rangle}/X\}$ and $(A\{\widetilde{\langle l=b\rangle}/X\}, B\{\widetilde{\langle l=b\rangle}/X\}) \in \mathcal{R}$. Since A and B are closed, i.e. $X \notin \mathrm{fv}(A|B)$, by Lemma 4.2 we have $A\{\widetilde{\langle l=b\rangle}/X\} = A$ and $B\{\widetilde{\langle l=b\rangle}/X\} = B$ and it follows $(A, B) \in \mathcal{R}$. □

In fact, an input prefix in the $\pi\mathcal{L}$-calculus, unlike the π-calculus, is not a binder for names. Therefore, we have $\mathrm{bn}(A) - \mathrm{bn}(a(X).A) = \emptyset$.

Lemma 4.3 $!a(X).A \;\overset{\mathcal{L}}{\approx}\; !a(X).A|a(X).A$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (!a(X).A|C, !a(X).A|a(X).A|C) \ \mid\ C \textit{ arbitrary }\} \cup \overset{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation. The case for output is obvious. We only show the case for input. Remaining cases are similar. Assume $!a(X).A|C \xrightarrow{a(\widetilde{\langle l=b\rangle})} !a(X).A|C'$. But if $C \xrightarrow{a(\widetilde{\langle l=b\rangle})} C'$ the result trivially holds. If not, we should have

$$\mathrm{REPL}\ \frac{a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\}}{\mathrm{PAR}\ \dfrac{!a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}{!a(X).A \mid C \xrightarrow{a(\widetilde{\langle l=b\rangle})} (A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A) \mid C}}$$

but then

$$\mathrm{PAR}\ \frac{a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\}}{\mathrm{PAR}\ \dfrac{a(X).A \mid !a(X).A \xrightarrow{a(\widetilde{\langle l=b\rangle})} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}{(a(X).A \mid !a(X).A) \mid C \xrightarrow{a(\widetilde{\langle l=b\rangle})} (A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A) \mid C}}$$

hence done. □

Using Lemma 4.3, an $\pi\mathcal{L}$-term $!a(X).A$ can be replaced by an arbitrary (as many as needed) number of parallel compositions of $a(X).A$.

Proposition 4.6 *For any A and B,*

$$a(X).A \overset{\mathcal{L}}{\approx} a(X).B \;\Rightarrow\; !a(X).A \;\overset{\mathcal{L}}{\approx}\; !a(X).B.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (!a(X).A|C, !a(X).B|D) \;\mid\; a(X).A \overset{\mathcal{L}}{\approx} a(X).B \text{ and } C \;\overset{\mathcal{L}}{\approx}\; D\} \cup \overset{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation. Output transitions are easy. For τ-transition, only the one between $!a(X).A$ and C is not immediate. Hence assume

$$!a(X).A|C \overset{\tau}{\Longrightarrow} !a(X).A|A\{\widetilde{\langle l{=}b\rangle}/X\}|C'.$$

Then

$$a(X).A|C \;\overset{\tau}{\Longrightarrow}\; A\{\widetilde{\langle l{=}b\rangle}/X\}|C'$$

therefore, by $a(X).A \overset{\mathcal{L}}{\approx} a(X).B$ and Lemma 4.3 there is a D' such that

$$a(X).B|D \;\overset{\tau}{\Longrightarrow}\; B\{\widetilde{\langle l{=}b\rangle}/X\}|D'$$

and $C' \overset{\mathcal{L}}{\approx} D'$, which means

$$!a(X).B|D \overset{\tau}{\Longrightarrow} !a(X).B|B\{\widetilde{\langle l{=}b\rangle}/X\}|D'$$

as required. For input, suppose $!a(X).A|C \overset{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} !a(X).A|A\{\widetilde{\langle l{=}b\rangle}/X\}|C$ but then

$$a(X).B \overset{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B\{\widetilde{\langle l{=}b\rangle}/X\}$$

and $A\{\widetilde{\langle l{=}b\rangle}/X\} \overset{\mathcal{L}}{\approx} B\{\widetilde{\langle l{=}b\rangle}/X\}$ hence

$$!a(X).B|D \overset{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} !a(X).B|B\{\widetilde{\langle l{=}b\rangle}/X\}|D$$

but by Proposition 4.4, $(!a(X).A|A\{\widetilde{\langle l{=}b\rangle}/X\}|C, !a(X).B|B\{\widetilde{\langle l{=}b\rangle}/X\}|D) \in \mathcal{R}$ as desired. □

Summarising Propositions 4.3, 4.4, 4.5, and 4.6, we have:

Proposition 4.7 $\overset{\mathcal{L}}{\approx}$ *is a congruence relation.* □

In fact, like for the asynchronous $\pi - calculus$ [13, 44, 101], this is the expected result, because we have only replaced the communication of tuples in the asynchronous π-calculus by communication of forms. All operators are left unchanged.

4.4.4 Alpha-conversion

In this subsection we show that alpha-convertible agents are weakly $\mathcal{L}$-bisimilar. To prove that $\equiv_\alpha$ is a weak $\mathcal{L}$-bisimulation we use the following lemma.

Lemma 4.4 *Suppose that $A \equiv_\alpha B$.*

If μ is not a bound output and $A \stackrel{\mu}{\Longrightarrow} A'$ then equally for some B' with $A' \equiv_\alpha B'$, $B \stackrel{\mu}{\Longrightarrow} B'$.

If $A \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A'$ and $\tilde{y} \cap \mathrm{n}(B) = \emptyset$ then equally for some B' with $A'\{\tilde{y}/\tilde{x}\}^{\mathcal{N}}_{\alpha} \equiv_\alpha B'$, $B \stackrel{(\nu\ \tilde{y})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B'$.

Proof: The proof is by induction on the depth of inference. We consider in turn each transition rule as the last rule applied in the inference of the antecedent $A \stackrel{\mu}{\Longrightarrow} A'$.

IN: We have $\mu = a(\widetilde{\langle l{=}b\rangle})$, $\mathrm{bn}(\mu) = \emptyset$, $A \equiv a(X).A_1$, and $A' \equiv A_1\{\widetilde{\langle l{=}b\rangle}/X\}$. Since $A \equiv_\alpha B$, B must also be an input-prefixed agent, differing at most in the bound variable of the input prefix. By applying alpha-conversion we can make the prefixes identical, s.t. $B \equiv x(X).B1$. Now, B has a transition $a(\widetilde{\langle l{=}b\rangle})$, s.t. $B \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B'$ with $B' \equiv B1\{\widetilde{\langle l{=}b\rangle}/X\}$ and $A' \equiv_\alpha B'$.

OUT: We have $\mu = \overline{a}(\widetilde{\langle l{=}b\rangle})$, $\mathrm{bn}(\mu) = \emptyset$, $A \equiv \overline{a}(\widetilde{\langle l{=}b\rangle})$, and $A' \equiv \mathbf{0}$. Since $\mathrm{bn}(A) = \emptyset$ and $\mathrm{bv}(A) = \emptyset$ we can replace $A \equiv_\alpha B$ with $A \equiv B$. Hence, it also holds that $B \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B'$ with $A' \equiv_\alpha B'$.

OPEN: We have $\mu = (\nu\ y, \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})$, $\mathrm{bn}(\mu) = \{y\} \cup \tilde{x}$, $A \equiv (\nu\ y)A_1$, and $A' \equiv A_1$. Then by assumption we can prove $A_1 \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A'$, and for some fresh name y we have $y \neq a$ and $y \in \mathcal{N}(\widetilde{\langle l{=}b\rangle}) - \tilde{x}$. Furthermore, it holds $\mathrm{fn}(A) = \mathrm{fn}(A_1)$ and since $A \equiv_\alpha B$ it holds $\mathrm{fn}(A) = \mathrm{fn}(B) = \mathrm{fn}(B_1)$ with $B_1 \equiv_\alpha A_1\{z/y\}^{\mathcal{N}}_{\alpha}$. Since $A_1 \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A'$ we can prove for some fresh name z with $z \neq a$ and $z \in \mathcal{N}(\widetilde{\langle l{=}b\rangle}\{z/y\}^{\mathcal{N}}_{\alpha}) - \tilde{x}$ $B_1 \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B'$ with $B' \equiv_\alpha A'\{z/y\}^{\mathcal{N}}_{\alpha}$. Therefore, it also holds $(\nu\ z)B \stackrel{(\nu\ z,\tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B'$.

COM: Then we have $\mu = \tau$, $A \equiv A_1|A_2$ with $A_1 \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A_1'$, $A_2 \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A_2'$, and $A' \equiv A_1'|A_2'$. Then by assumption we can prove $A_1 \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A_1'$ and $A_2 \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} A_2'$. Since $A \equiv_\alpha B$ it holds $\mathrm{fn}(A) = \mathrm{fn}(B)$. Then with $B_1 \equiv_\alpha A_1$, $B_2 \equiv_\alpha A_2$, $B_1' \equiv_\alpha A_1'$, and $B_2' \equiv_\alpha A_2'$, we can prove $B_1 \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B_1'$ and $B_2 \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} B_2'$. Therefore, it also holds $B \stackrel{\tau}{\Longrightarrow} B'$ with $B \equiv B_1|B_2$ and $B' \equiv B_1'|B_2'$.

The proofs for CLOSE, PAR, RES, and REPL are similar. □

Theorem 4.1 *$\equiv_\alpha$ is a weak $\mathcal{L}$-bisimulation.*

PROOF: If A is a closed agent and $B \equiv_\alpha A$ then B is also closed. Therefore, A and B differing at most in the choice of bound names and variables, repectively. By applying alpha-conversion we can make them identical. Using the preceding lemma every move of A can be matched up by $B \equiv_\alpha A\alpha^{\mathcal{N}/\mathcal{V}}$ for some alpha-substitution $\alpha^{\mathcal{N}/\mathcal{V}}$. □

With Theorem 4.1 it is always possible to identify freely alpha-convertible agents writing $\equiv$ for $\equiv_\alpha$.

4.5 From π-calculus to $\pi\mathcal{L}$ – and back

In this section we present the compilation from the asynchronous polyadic π-calculus into the $\pi\mathcal{L}$-calculus and back. The compilations illustrates that both calculi can faithfully encode each other. Moreover, we show that both compilations preserve the weak asynchronous bisimulation relation, i.e., if two asynchronous π-processes are weakly bisimular, then their compilations are also weakly bisimular in $\pi\mathcal{L}$ and vice versa.

4.5.1 Transition system and bisimulation for the π-calculus

In Section 3.2.1, we have given the syntax and reduction semantics of the mini π-calculus. In order to show the faithfulness of the compilations, we use the 1-bisimulation defined by Amadio et al. [4]. This bisimulation is defined over a labelled transition system. Therefore, before presenting the compilations, we give a labelled transition system for the mini π-calculus.

We use α to range over π-calculus actions that have the following structure:

$$\begin{array}{rcll} \alpha & = & \tau & \text{silent action} \\ & | & a(\tilde{b}) & \text{input action} \\ & | & \overline{a}(\tilde{b}) & \text{output action} \\ & | & (\nu\ \tilde{x})\overline{a}(\tilde{b}) & \text{restricted output action} \end{array}$$

Bound names and free names of an action α, written $\mathrm{bn}(\alpha)$ and $\mathrm{fn}(\alpha)$, respectively, are defined as follows:

$$\begin{array}{rcl} \mathrm{fn}(\tau) = \emptyset & , & \mathrm{bn}(\tau) = \emptyset \\ \mathrm{fn}(a(\tilde{b})) = \{a\} & , & \mathrm{bn}(a(\tilde{b})) = \{\tilde{b}\} \\ \mathrm{fn}(\overline{a}(\tilde{b})) = \{a\} \cup \{\tilde{b}\} & , & \mathrm{bn}(\overline{a}(\tilde{b})) = \emptyset \\ \mathrm{fn}((\nu\ \tilde{x})\overline{a}(\tilde{b})) = (\{a\} \cup \{\tilde{b}\}) - \{\tilde{x}\} & , & \mathrm{bn}((\nu\ \tilde{x})\overline{a}(\tilde{b})) = \{\tilde{x}\} \end{array}$$

$$\text{IN} : a(\tilde{x}).P \xrightarrow{a(\tilde{b})} P\{\tilde{b}/\tilde{x}\} \qquad \text{OUT} : \overline{a}\langle\tilde{b}\rangle \xrightarrow{\overline{a}(\tilde{b})} \mathbf{0}$$

$$\text{OPEN} : \frac{P \xrightarrow{(\nu\ \tilde{x})\overline{a}(\tilde{b})} P' \qquad y \neq a \qquad y \in \tilde{b} - \tilde{x}}{(\nu\ y)P \xrightarrow{(\nu\ y,\tilde{x})\overline{a}(\tilde{b})} P'}$$

$$\text{COM} : \frac{P \xrightarrow{\overline{a}(\tilde{b})} P' \qquad Q \xrightarrow{a(\tilde{b})} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{CLOSE} : \frac{P \xrightarrow{(\nu\ \tilde{x})\overline{a}(\tilde{b})} P' \qquad Q \xrightarrow{a(\tilde{b})} Q' \qquad \tilde{x} \notin \text{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu\ \tilde{x})(P' \mid Q')}$$

$$\text{PAR} : \frac{P \xrightarrow{\alpha} P' \qquad \text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{RES} : \frac{P \xrightarrow{\alpha} P' \qquad x \notin \text{n}(\alpha)}{(\nu\ x)P \xrightarrow{\alpha} (\nu\ x)P'}$$

$$\text{REPL} : \frac{a(\tilde{x}).P \xrightarrow{a(\tilde{b})} P\{\tilde{b}/\tilde{x}\}}{!a(\tilde{x}).P \xrightarrow{a(\tilde{b})} P\{\tilde{b}/\tilde{x}\} \mid !a(\tilde{x}).P}$$

Table 4.4: Labelled transition system for the mini π-calculus.

The names of α, written $\text{n}(\alpha)$, are $\text{bn}(\alpha) \cup \text{fn}(\alpha)$. The labelled transition system for the mini π-calculus is given in Table 4.4. We have omitted the symmetric versions of rules PAR and COM. We will identify alpha-convertible processes, i.e, they have the same transitions.

Weak arrows $\Longrightarrow$ denote the reflexive and transitive closure of transitions. We have:

$$\begin{aligned} P \overset{\tau}{\Longrightarrow} P' \quad & \text{iff } P(\xrightarrow{\tau}) * P' \\ P \overset{\alpha}{\Longrightarrow} P' \quad & \text{iff } P \overset{\tau}{\Longrightarrow} \cdot \xrightarrow{\alpha} \cdot \overset{\tau}{\Longrightarrow} P',\ \alpha \neq \tau \end{aligned}$$

Definition 4.12 (Weak asynchronous 1-bisimulation [4])
A binary relation $\mathcal{R}$ over π-processes P and Q is a weak 1-bisimulation if it is symmetric and $P\ \mathcal{R}\ Q$ implies

- *whenever $P \xrightarrow{\alpha} P'$, where α is either τ or output with $\text{bn}(\alpha) \cap \text{fn}(P|Q) = \emptyset$, then Q' exists such that $Q \overset{\alpha}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$.*

- $(P \mid \overline{a}(\tilde{b}))\ \mathcal{R}\ (Q \mid \overline{a}(\tilde{b}))$ *for all messages* $\overline{a}(\tilde{b})$.

Two processes P and Q are weakly 1-bisimular, written $P \approx Q$, if there is a weak 1-bisimulation $\mathcal{R}$ with $P\ \mathcal{R}\ Q$.

4.5.2 The compilation from π to $\pi\mathcal{L}$-calculus

We now present the translation from the asynchronous polyadic π-calculus into the $\pi\mathcal{L}$-calculus. The basic idea of this translation is the use of de Bruijn indices [31]. More precisely, in an input-prefixed process $a(\tilde{x}).P$, we assign every parameter name x_i a unique non-negative integer i with respect to a fresh form variable X (in fact, we use the parameter's position index) and replace every application of x_i in P by a projection X_{l_i} where l_i maps x_i.

Similarly, in an output-particle $\overline{a}\langle\tilde{b}\rangle$, we replace every b_i with a binding $\langle l_i = b_i\rangle$ where i is a unique non-negative integer (in fact, i is the actual output parameter position). The reader should note that if b_i is bound by an input prefix $a(\tilde{x})$, then the translation replaces b_i by X_{l_j} where j is the position index of b_i in the input prefix (i.e., $b_i = x_j$) and l_j denotes b_i with respect to the fresh form variable X. For example, the π-process

$$a(x_1, x_2, x_3).(\overline{b}(x_1, x_2) \mid \overline{c}(x_1, x_3))$$

is translated into a $\pi\mathcal{L}$-agent

$$a(X).(\overline{b}(\langle l_1 = X_{l_1}\rangle\langle l_2 = X_{l_2}\rangle) \mid \overline{c}(\langle l_1 = X_{l_1}\rangle\langle l_2 = X_{l_3}\rangle))$$

For the translation we use the function $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ that takes a π-process P and returns the corresponding $\pi\mathcal{L}$-agent. Within the translation Γ is used to record the names that have been seen. In fact, Γ is a symbol table that maps restricted names to itself and names that are bound by an input prefix to a corresponding projection. In the translation $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$, Λ is an input counter, i.e., Λ keeps track of all input-prefixes. In fact, Λ denotes the actual fresh form variable that replaces the input parameters $\tilde{x}$ of an input-prefixed process $a(\tilde{x}).P$. For example, in the translation of

$$a(x_1).b(y_1).\overline{c}(x_1, y_1)$$

we start with $\Lambda = 0$ such that first the translated input prefix becomes $a(X^0)$, while Λ is 1 for the second input prefix, so that the translation yields $b(X^1)$. Therefore, the translated agent becomes

$$a(X^0).b(X^1).\overline{c}(\langle l_1 = X^0_{l_1}\rangle\langle l_1 = X^1_{l_1}\rangle)$$

Now, we present the translation function $\mathcal{C}[\![]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ that starts with $\Lambda = 0$ and $\Gamma = \emptyset$ ($\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}$ is the translations function for π-calculus actions). Furthermore, $\mathcal{C}[\![]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ uses a function **map** that maps names to names and projections, respectively:

$$\begin{array}{rcl}
\mathcal{C}[\![\mathbf{0}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & \mathbf{0} \\
\mathcal{C}[\![P \mid Q]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & \mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \\
\mathcal{C}[\![(\nu\ x)P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & (\nu\ x)\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma:(x\mapsto n(x))} \\
\mathcal{C}[\![!a(x_1,...,x_n).P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & !\mathcal{C}[\![a(x_1,...,x_n).P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \\
\mathcal{C}[\![a(x_1,...,x_n).P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & \mathbf{map}(\Gamma,a)(X^{\Lambda+1}).\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda+1,\Gamma:(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+1,n))} \\
\mathcal{C}[\![\overline{a}\langle x_1,...,x_n\rangle]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} & = & \overline{\mathbf{map}(\Gamma,a)}(\langle l_1{=}\mathbf{map}(\Gamma,x_1)\rangle...\langle l_n{=}\mathbf{map}(\Gamma,x_n)\rangle)
\end{array}$$

$$\mathbf{map}(\Gamma,x) = \begin{cases} X^{\Lambda}_{l_n}, & \textit{if } \Gamma = \Gamma_1 : (x \mapsto v(\Lambda,n)) : \Gamma_2 \\ x, & \textit{otherwise} \end{cases}$$

$$\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}} = \begin{cases} \tau, & \tau \\ a(\widetilde{\langle l{=}b\rangle}), & \textit{if } \alpha = a(\tilde{b}) \\ \overline{a}(\widetilde{\langle l{=}b\rangle}), & \textit{if } \alpha = \overline{a}(\tilde{b}) \\ (\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle}), & \textit{if } \alpha = (\nu\ \tilde{x})\overline{a}(\tilde{b}) \end{cases}$$

The extension of Γ may hide existing mappings, i.e., for example, if Γ is extended by $(x \mapsto v(\Lambda,n))$ and Γ already contains a mapping for x, say $(x \mapsto n(x))$ such that $\Gamma = \Gamma_1 : (x \mapsto n(x)) : \Gamma_2$, then the function $\mathbf{map}(\Gamma : (x \mapsto v(\Lambda,n)), x)$ yields $X^{\Lambda}_{l_n}$, which corresponds to the latest mapping $(x \mapsto v(\Lambda,n))$. Furthermore, if Γ defines no mapping for a name x, then $\mathbf{map}(\Gamma,x) = x$. The collection of names $x_1,...,x_n$ for which a mapping is declared in Γ is indicated by $dom(\Gamma)$.

We show now that $\mathcal{C}[\![]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ is faithful, i.e., if two processes P and Q are weakly 1-bisimular, then this implies that $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$.

Lemma 4.5
Let P be a π-process and $\Gamma = (x_1 \mapsto v(\Lambda,1)) : ... : (x_n \mapsto v(\Lambda,n))$, i.e., $\mathcal{C}[\![]\!]^{\pi\mathcal{L}}$ has already processed an input prefix with P as subprocess that binds $x_1,...,x_n$. Then it holds that $\mathcal{C}[\![P\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^{\Lambda}\}$.

Proof: By induction on the structure of P. Furthermore, we assume that if name $a \notin dom(\Gamma)$, then it holds $a\{\tilde{b}/\tilde{x}\} = a$.

- $P = \mathbf{0}$.
  Immediate, since $\mathcal{C}[\![\mathbf{0}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathbf{0}$ and $\mathbf{0}\{\tilde{b}/\tilde{x}\} = \mathbf{0}$ and $\mathbf{0}\{\widetilde{\langle l{=}b\rangle}/X^{\Lambda}\} = \mathbf{0}$.
- $P = a(\tilde{x}).P_1$ where $\tilde{x} = x_1,...,x_n$.
  Case $a \notin dom(\Gamma)$. Then it must be the case that $\mathbf{map}(\Gamma,a) = a$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = a(X^{\Lambda+1}).\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+1,n))}$. Using induction, we can prove that
  $$\begin{array}{rl} & \mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+1,n))} \\ = & \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+1,n))}\{\widetilde{\langle l{=}b\rangle}/X^{\Lambda}\} \end{array}$$

and since $a\{\tilde{b}/\tilde{x}\} = a$ and $a\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = a$ we get

$$\begin{array}{ll} & a(X^{\Lambda+1}).\mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))} \\ = & a(X^{\Lambda+1}).(\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\}) \end{array}$$

Case $a \in dom(\Gamma)$. Then it must be the case that $\mathbf{map}(\Gamma, a) = X^\Lambda_{l_j}$, $a\{\tilde{b}/\tilde{x}\} = b_j$, and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = X^\Lambda_{l_j}(X^{\Lambda+1}).\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))}$. Using induction, we can prove that

$$\begin{array}{ll} & \mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))} \\ = & \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} \end{array}$$

and since $a\{\tilde{b}/\tilde{x}\} = b_j$ and $X^\Lambda_{l_j}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = b_j$ we get

$$\begin{array}{ll} & b_j(X^{\Lambda+1}).\mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))} \\ = & b_j(X^{\Lambda+1}).(\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda+1,(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\}) \end{array}$$

- $P = \overline{a}\langle\tilde{c}\rangle$ where $\tilde{c} = c_1, ..., c_n$.

  Case $a, c_1, ..., c_n \notin dom(\Gamma)$. Then we have $\mathbf{map}(\Gamma, a) = a$, $\mathbf{map}(\Gamma, c_1) = c_1$,..., $\mathbf{map}(\Gamma, c_n) = c_n$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \overline{\mathbf{map}(\Gamma, a)}(\langle l_1{=}\mathbf{map}(\Gamma, c_1)\rangle ... \langle l_n{=}\mathbf{map}(\Gamma, c_n)\rangle)$. Since $a\{\tilde{b}/\tilde{x}\} = a$, $c_1\{\tilde{b}/\tilde{x}\} = c_1$, ... , $c_n\{\tilde{b}/\tilde{x}\} = c_n$ and $a\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = a$, $c_1\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = c_1$, $c_n\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = c_n$ we get

  $$\begin{array}{ll} & \mathcal{C}[\![\overline{a}\langle\tilde{c}\rangle\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \overline{a}(\langle l_1{=}c_1\rangle ... \langle l_n{=}c_n\rangle) \\ = & \mathcal{C}[\![\overline{a}\langle\tilde{c}\rangle]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = \overline{a}(\langle l_1{=}c_1\rangle ... \langle l_n{=}c_n\rangle). \end{array}$$

  Case $a, c_1, ..., c_n \in dom(\Gamma)$. Then we have $\mathbf{map}(\Gamma, a) = X^\Lambda_{l_j}$, $\mathbf{map}(\Gamma, c_1) = X^\Lambda_{l_{k_1}}$,..., $\mathbf{map}(\Gamma, c_n) = X^\Lambda_{l_{k_n}}$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \overline{\mathbf{map}(\Gamma, a)}(\langle l_1{=}\mathbf{map}(\Gamma, c_1)\rangle ... \langle l_n{=}\mathbf{map}(\Gamma, c_n)\rangle)$. Since $a\{\tilde{b}/\tilde{x}\} = b_j$, $c_1\{\tilde{b}/\tilde{x}\} = b_{k_1}$, ... , $c_n\{\tilde{b}/\tilde{x}\} = b_{k_n}$ and $a\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = b_j$, $c_1\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = b_{k_1}$, $c_n\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = b_{k_n}$ we get

  $$\begin{array}{ll} & \mathcal{C}[\![\overline{a}\langle\tilde{c}\rangle\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \overline{b_j}(\langle l_1{=}b_{k_1}\rangle ... \langle l_n{=}b_{k_n}\rangle) \\ = & \mathcal{C}[\![\overline{a}\langle\tilde{c}\rangle]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = \overline{b_j}(\langle l_1{=}b_{k_1}\rangle ... \langle l_n{=}b_{k_n}\rangle). \end{array}$$

- $P = P_1 \mid P_2$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. We can therefore use induction to prove $\mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\}$ and $\mathcal{C}[\![P_2\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\}$ such that

  $$\begin{array}{ll} & \mathcal{C}[\![(P_1 \mid P_2)\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \\ = & \mathcal{C}[\![P_1 \mid P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} \end{array}$$

- $P = (\nu\ x)P_1$.

Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = (\nu\ x)\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. We can therefore use induction to prove $\mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\}$ such that

$$\begin{array}{ll} & \mathcal{C}[\![((\nu\ x)P_1)\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = (\nu\ x)\mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \\ = & \mathcal{C}[\![(\nu\ x)P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} = (\nu\ x)\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}\{\widetilde{\langle l{=}b\rangle}/X^\Lambda\} \end{array}$$

- $P = !a(\tilde{x}).P_1$. Similar to the previous case. □

Lemma 4.6
Let P be a π-process and α a π-action. Then $P \stackrel{\alpha}{\Longrightarrow} P'$ implies $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}}{\Longrightarrow} \mathcal{C}[\![P']\!]^{\pi\mathcal{L}}_{\Lambda',\Gamma'}$.

PROOF: We proceed by induction on the structure of P with $P \not\equiv \mathbf{0}$. Furthermore, we use $\Lambda = 0$ and $\Gamma = \emptyset$ as start conditions.

- $P = a(\tilde{x}).P_1$ where $\tilde{x} = x_1, ..., x_n$.
  Then we have $\mathbf{map}(\emptyset, a) = a$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} = a(X^1).\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}$.
  It holds that $a(\tilde{x}).P_1 \stackrel{a(\tilde{b})}{\Longrightarrow} P_1\{\tilde{b}/\tilde{x}\}$ and
  $$\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}\{\widetilde{\langle l{=}b\rangle}/X^1\}.$$
  We have $\mathcal{C}[\![a(\tilde{b})]\!]^{\pi\mathcal{L}} = a(\widetilde{\langle l{=}b\rangle})$ and by Lemma 4.5
  $$\begin{array}{ll} & \mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))} \\ = & \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}\{\widetilde{\langle l{=}b\rangle}/X^1\}. \end{array}$$
- $P = \overline{a}\langle\tilde{b}\rangle$ where $\tilde{b} = b_1, ..., b_n$.
  Then we have $\mathbf{map}(\emptyset, a) = a$, $\mathbf{map}(\emptyset, b_1) = b_1$,..., $\mathbf{map}(\emptyset, b_n) = b_n$, and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} = \overline{a}(\langle l_1 = b_1\rangle...\langle l_n = b_n\rangle)$. It holds that $\overline{a}\langle\tilde{b}\rangle \stackrel{\overline{a}(\tilde{b})}{\Longrightarrow} \mathbf{0}$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} \mathcal{C}[\![\mathbf{0}]\!]^{\pi\mathcal{L}}_{0,\emptyset}$ with $\mathcal{C}[\![\overline{a}(\tilde{b})]\!]^{\pi\mathcal{L}} = \overline{a}(\widetilde{\langle l{=}b\rangle})$ as required.
- $P = P_1 \mid P_2$.
  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$.
  - Using COM, $P_1 \mid P_2 \stackrel{\tau}{\Longrightarrow} P_1' \mid P_2'$ is inferred from $P_1 \stackrel{\overline{a}(\tilde{b})}{\Longrightarrow} P_1'$ and $P_2 \stackrel{a(\tilde{b})}{\Longrightarrow} P_2'$. Using induction twice, we have
    $$\frac{\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\overline{a}(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \qquad \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{a(\widetilde{\langle l{=}b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}{\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\tau}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1' \mid P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}$$

– Using CLOSE, $P_1 \mid P_2 \stackrel{\tau}{\Longrightarrow} (\nu\ \tilde{x})(P_1' \mid P_2')$ is inferred from $P_1 \stackrel{(\nu\ \tilde{x})\overline{a}(\tilde{b})}{\Longrightarrow} P_1'$ and $P_2 \stackrel{a(\tilde{b})}{\Longrightarrow} P_2'$. Using induction twice, we have

$$\frac{\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \qquad \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}{\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\tau}{\Longrightarrow} (\nu\ \tilde{x})(\mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}} \mid \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}) = \mathcal{C}[\![(\nu\ \tilde{x})(P_1' \mid P_2')]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}$$

- $P = (\nu\ x)P_1$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = (\nu\ x)\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma:(x\mapsto n(x))}$. Using RES, $(\nu\ x)P_1 \stackrel{\alpha}{\Longrightarrow} (\nu\ x)P_1'$ is inferred from $P_1 \stackrel{\alpha}{\Longrightarrow} P_1'$. The result follows using induction

$$\frac{\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\alpha}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}{(\nu\ x)\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\alpha}{\Longrightarrow} (\nu\ x)\mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![(\nu\ x)P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}}$$

- $P = !a(\tilde{x}).P_1$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = !\mathcal{C}[\![a(\tilde{x}).P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. The result follows using induction on $\mathcal{C}[\![a(\tilde{x}).P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$.

□

The following lemma states that the converse also holds.

Lemma 4.7
Let P be a π-process and α a π-action. Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\mu}{\Longrightarrow} P''$ implies there exists P', α such that $P \stackrel{\alpha}{\Longrightarrow} P'$ and $P'' = \mathcal{C}[\![P']\!]^{\pi\mathcal{L}}_{\Lambda',\Gamma'}$, $\mu = \mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}$.

PROOF: We proceed by induction on the structure of P with $P \not\equiv \mathbf{0}$. Furthermore, we use $\Lambda = 0$ and $\Gamma = \emptyset$ as start conditions.

- $P = a(\tilde{x}).P_1$ where $\tilde{x} = x_1, ..., x_n$.

  Then we have $\mathbf{map}(\emptyset, a) = a$ and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} = a(X^1).\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}$.

  It holds that $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} P''\{\widetilde{\langle l=b\rangle}/X^1\}$ and $a(\tilde{x}).P_1 \stackrel{a(\tilde{b})}{\Longrightarrow} P_1\{\tilde{b}/\tilde{x}\}$.

  We have $a(\tilde{b}) = \mathcal{C}[\![a(\widetilde{\langle l=b\rangle})]\!]^{\pi\mathcal{L}}$, $P'' = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}$, and by Lemma 4.5

$$\begin{array}{ll} & \mathcal{C}[\![P_1\{\tilde{b}/\tilde{x}\}]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))} \\ = & P''\{\widetilde{\langle l=b\rangle}/X^1\} \\ = & \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{1,(x_1\mapsto v(1,1)):...:(x_n\mapsto v(1,n))}\{\widetilde{\langle l=b\rangle}/X^1\} \end{array}$$

- $P = \overline{a}\langle\tilde{b}\rangle$ where $\tilde{b} = b_1, ..., b_n$.

  Then we have $\mathbf{map}(\emptyset, a) = a$, $\mathbf{map}(\emptyset, b_1) = b_1$,..., $\mathbf{map}(\emptyset, b_n) = b_n$, and $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} = \overline{a}(\langle l_1 = b_1\rangle ... \langle l_n = b_n\rangle)$. It holds that $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{0,\emptyset} \stackrel{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} P''$ and $\overline{a}\langle\tilde{b}\rangle \stackrel{\overline{a}(\tilde{b})}{\Longrightarrow} \mathbf{0}$. We have $\overline{a}(\tilde{b}) = \mathcal{C}[\![\overline{a}(\widetilde{\langle l=b\rangle})]\!]^{\pi\mathcal{L}}$ and $P'' = \mathcal{C}[\![\mathbf{0}]\!]^{\pi\mathcal{L}}_{0,\emptyset}$ as required.

- $P = P_1 \mid P_2$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$.

  - Using COM, $\mathcal{C}[\![P_1 \mid P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\tau}{\Longrightarrow} \mathcal{C}[\![P_1' \mid P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ is inferred from $\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ and $\mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. Using induction twice and $\overline{a}(\widetilde{\langle l=b\rangle}) = \mathcal{C}[\![\overline{a}(\tilde{b})]\!]^{\pi\mathcal{L}}$, $a(\widetilde{\langle l=b\rangle}) = \mathcal{C}[\![a(\tilde{b})]\!]^{\pi\mathcal{L}}$ and $\tau = \mathcal{C}[\![\tau]\!]^{\pi\mathcal{L}}$, we have

$$\frac{P_1 \stackrel{\overline{a}(\tilde{b})}{\Longrightarrow} P_1' \qquad P_2 \stackrel{a(\tilde{b})}{\Longrightarrow} P_2'}{P_1 \mid P_2 \stackrel{\tau}{\Longrightarrow} P_1' \mid P_2'}$$

  - Using CLOSE, $\mathcal{C}[\![P_1 \mid P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\tau}{\Longrightarrow} \mathcal{C}[\![(\nu\ \tilde{x})(P_1' \mid P_2')]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ is inferred from $\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ and $\mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathcal{C}[\![P_2']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. Using induction twice and $(\nu\ \tilde{x})\overline{a}(\widetilde{\langle l=b\rangle}) = \mathcal{C}[\![(\nu\ \tilde{x})\overline{a}(\tilde{b})]\!]^{\pi\mathcal{L}}$, $a(\widetilde{\langle l=b\rangle}) = \mathcal{C}[\![a(\tilde{b})]\!]^{\pi\mathcal{L}}$ and $\tau = \mathcal{C}[\![\tau]\!]^{\pi\mathcal{L}}$, we have

$$\frac{P_1 \stackrel{(\nu\ \tilde{x})\overline{a}(\tilde{b})}{\Longrightarrow} P_1' \qquad P_2 \stackrel{a(\tilde{b})}{\Longrightarrow} P_2'}{P_1 \mid P_2 \stackrel{\tau}{\Longrightarrow} (\nu\ \tilde{x})(P_1' \mid P_2')}$$

- $P = (\nu\ x)P_1$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = (\nu\ x)\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma:(x\mapsto n(x))}$. Using RES, $\mathcal{C}[\![(\nu\ x)P_1]\!]^{\pi\mathcal{L}}_{\Lambda\Gamma} \stackrel{\mu}{\Longrightarrow} \mathcal{C}[\![(\nu\ x)P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$ is inferred from $\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} \stackrel{\mu}{\Longrightarrow} \mathcal{C}[\![P_1']\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. The result follows using induction and $\alpha = \mathcal{C}[\![\mu]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$

$$\frac{P_1 \stackrel{\alpha}{\Longrightarrow} P_1'}{(\nu\ x)P_1 \stackrel{\alpha}{\Longrightarrow} (\nu\ x)P_1'}$$

- $P = !a(\tilde{x}).P_1$.

  Then $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = !\mathcal{C}[\![a(\tilde{x}).P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$. The result follows using induction on $\mathcal{C}[\![a(\tilde{x}).P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma}$.

□

Lemma 4.8
Let α be a π-action. Then $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \subseteq \mathrm{bn}(\alpha)$.

PROOF: We proceed by induction on the structure of α. We consider the most significant cases.

- Case $\alpha = a(\tilde{b})$.

  Then $\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}} = a(\widetilde{\langle l = b\rangle})$, $\mathrm{bn}(a(\tilde{b})) = \tilde{b}$ and $\mathrm{bn}(a(\widetilde{\langle l = b\rangle}) = \emptyset$ such that $\emptyset \subset \tilde{b}$ as required.

- Case $\alpha = (\nu\ \tilde{x})\overline{a}(\tilde{x})$.

  Then $\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}} = (\nu\ \tilde{x})\overline{a}(\widetilde{\langle l = b\rangle})$, $\mathrm{bn}((\nu\ \tilde{x})\overline{a}(\tilde{x})) = \tilde{x}$ and $\mathrm{bn}((\nu\ \tilde{x})\overline{a}(\widetilde{\langle l = b\rangle})) = \tilde{x}$ such that $\tilde{x} \subseteq \tilde{x}$ as required. □

Lemma 4.9
Let P be a π-process. Then $\mathrm{fn}(P) = \mathrm{fn}(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}})$.

PROOF: We proceed by induction on the structure of P. We consider the most significant case $P = a(\tilde{x}).P_1$ where $\tilde{x} = x_1, ..., x_n$.
Then we have $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma} = a(X^{\Lambda+1}).\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma:(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+11,n))}$ and $\mathrm{fn}(a(\tilde{x}).P_1) = \{a\} \cup (\mathrm{fn}(P_1) - \tilde{x})$. Now, suppose that $\mathrm{fn}(P_1) \cap \tilde{x} \neq \emptyset$, then these names become unbound after $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}$. However, it holds that $x_i \in dom(\Gamma : (x_1 \mapsto v(\Lambda+1,1)) : ... : (x_n \mapsto v(\Lambda+1,n)))$ for i in $1, ..., n$. Therefore, $\mathbf{map}(\Gamma : (x_1 \mapsto v(\Lambda+1,1)) : ... : (x_n \mapsto v(\Lambda+1,x_i))) = X^{\Lambda+1}_{l_i}$, hence the translation removes the name x_i from the resulting agent such that $\mathrm{fn}(\mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}_{\Lambda,\Gamma:(x_1\mapsto v(\Lambda+1,1)):...:(x_n\mapsto v(\Lambda+11,n))}) \cap \tilde{x} = \emptyset$. We have therefore $\mathrm{fn}(P) = \mathrm{fn}(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}})$ as required. □

Lemma 4.10
Let P and Q be π-processes and α be a π-action. Then $\mathrm{bn}(\alpha) \cap \mathrm{fn}(P|Q) = \emptyset$ *implies* $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \cap \mathrm{fn}(\mathcal{C}[\![P|Q]\!]^{\pi\mathcal{L}}) = \emptyset$.

PROOF: Suppose $\mathrm{bn}(\alpha) \cap \mathrm{bn}(P|Q) = \emptyset$ then by Lemma 4.8 we know that $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \subseteq \mathrm{bn}(\alpha)$, i.e., the translation $\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}$ may remove binders from α. On the other side, we known by Lemma 4.9 that $\mathrm{fn}(P) = \mathrm{fn}(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}})$. Hence, $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \cap \mathrm{fn}(\mathcal{C}[\![P|Q]\!]^{\pi\mathcal{L}}) = \emptyset$ since it only depends on the set $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}})$ which is smaller than $\mathrm{bn}(\alpha)$. □

Theorem 4.2 *For each pair of π-processes P and Q, $\mathcal{C}[\![]\!]^{\pi\mathcal{L}}$ is an injective mapping, such that if $P \approx Q$, then it holds*

$$\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (\mathcal{C}[\![P]\!]^{\pi\mathcal{L}}, \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}})\ \mid\ P \approx Q\ \} \cup \overset{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation.
Consider first the case τ or *output actions*:

> By Lemma 4.10, we have $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \cap \mathrm{fn}(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \mid \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}) = \emptyset$.
>
> Now let $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mu}{\Longrightarrow} P''$. By Lemma 4.7,
>
> $$P \overset{\alpha}{\Longrightarrow} P' \text{ with } \mu = \mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}} \text{ and } P'' = \mathcal{C}[\![P']\!]^{\pi\mathcal{L}}.$$
>
> The definition of $\approx$ guarantees that there exists Q' such that $Q \overset{\alpha}{\Longrightarrow} Q'$ and $P' \approx Q'$. By Lemma 4.6, $\mathcal{C}[\![Q]\!]^{\pi\mathcal{L}} \overset{\mu}{\Longrightarrow} \mathcal{C}[\![Q']\!]^{\pi\mathcal{L}}$. Thus we have $P''\ \mathcal{R}\ \mathcal{C}[\![Q']\!]^{\pi\mathcal{L}}$.

Consider now the case with composition with output:

> Since $P \approx Q$, we have $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}$. Then for all messages $\overline{a}(\langle\widetilde{l=b}\rangle)$ we have by definition $(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \mid \overline{a}(\langle\widetilde{l=b}\rangle), \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}} \mid \overline{a}(\langle\widetilde{l=b}\rangle)) \in \mathcal{R}$. □

The following theorem shows that weak $\mathcal{L}$-bisimilarity in $\pi\mathcal{L}$ of translated π-processes implies weak 1-bisimilarity in π for these processes.

Theorem 4.3 *For each pair of π-processes P and Q, if $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}$, then it holds $P \approx Q$.*

PROOF: We show that the relation

$$\mathcal{R} = \{\ (P, Q)\ \mid\ \mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}\ \} \cup \approx$$

is a weak 1-bisimulation.
Consider first the case τ or *output actions*:

> By Lemma 4.9, we have $\mathrm{fn}(P \mid Q) = \mathrm{fn}(\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \mid \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}})$ and by Lemma 4.8, we have $\mathrm{bn}(\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}) \subseteq \mathrm{bn}(\alpha)$. Now, it is possible that $\mathrm{bn}(\alpha) \cap \mathrm{fn}(P|Q) = \tilde{b} \neq \emptyset$ which means that α carries bound names (locations) $\tilde{b}$ that are not yet known to the target of α. However, the choice of names is unimportant as long as they denote the same location. We can take therefore fresh names $\tilde{b'}$ with $\tilde{b'} \cap \mathrm{n}(P \mid Q) = \emptyset$ and α-convert the action α such that $\mathrm{bn}(\alpha\{\tilde{b'}/\tilde{b}\}) \cap \mathrm{fn}(P|Q) = \emptyset$.
>
> Suppose $\mathrm{bn}(\alpha) \cap \mathrm{fn}(P|Q) = \emptyset$. Now let $P \overset{\alpha}{\Longrightarrow} P'$. By Lemma 4.6,
>
> $$\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mu}{\Longrightarrow} \mathcal{C}[\![P']\!]^{\pi\mathcal{L}} \text{ with } \mu = \mathcal{C}[\![\alpha]\!]^{\pi\mathcal{L}}.$$

The definition of $\overset{\mathcal{L}}{\approx}$ guarantees that there exists $\mathcal{C}[\![Q']\!]^{\pi\mathcal{L}}$ such that $\mathcal{C}[\![Q]\!]^{\pi\mathcal{L}} \overset{\mu}{\Longrightarrow} Q''$ and $\mathcal{C}[\![P']\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} Q''$. By Lemma 4.7, $Q \overset{\alpha}{\Longrightarrow} Q'$ and $Q'' = \mathcal{C}[\![Q']\!]^{\pi\mathcal{L}}$. Thus we have $P' \mathcal{R} Q'$.

Consider now the case with composition with output:

Since $\mathcal{C}[\![P]\!]^{\pi\mathcal{L}} \overset{\mathcal{L}}{\approx} \mathcal{C}[\![Q]\!]^{\pi\mathcal{L}}$, we have $P \approx Q$. Then for all messages $\overline{a}(\tilde{b})$ we have by definition $(P \mid \overline{a}(\tilde{b}), Q \mid \overline{a}(\tilde{b})) \in \mathcal{R}$. □

4.5.3 The compilation from $\pi\mathcal{L}$ to π-calculus

We now present the translation from the $\pi\mathcal{L}$-calculus into the asynchronous polyadic π-calculus. The basic idea of this translation is that we represent $\pi\mathcal{L}$-forms as polyadic-π-processes. More precisely, a $\pi\mathcal{L}$-form is translated into a π-process listening at a channel f (the form location) for a channel L that represents the actual projection label, a result channel R along which the result of the projection is returned, and an *error-continuation* channel E that is the location of the continuation if the actual label is not defined for the form located at f. For example, the form

$$\langle l = s \rangle \langle n = t \rangle$$

is translated into the following replicated process:

$$\begin{array}{l} P \equiv\, !f(L,R,E). \\ \quad (\nu\ n,c)(\ \overline{L}(c,n) \\ \quad\quad \mid n().\overline{R}(t) \\ \quad\quad \mid c().(\nu\ f')(\ \overline{f'}(L,R,E) \\ \quad\quad\quad\quad \mid f'(L,R,E). \\ \quad\quad\quad\quad\quad (\nu\ l,c')(\ \overline{L}(l,c') \\ \quad\quad\quad\quad\quad\quad \mid l().\overline{R}(s) \\ \quad\quad\quad\quad\quad\quad \mid c'().(\nu\ f'')(\ \overline{f''}(L,R,E) \mid f''(L,R,E).\overline{E}()))))) \end{array}$$

where f denotes the location of the form $\langle l = s \rangle \langle n = t \rangle$. The output-particles $\overline{L}(c,n)$ and $\overline{L}(l,c')$ initiate the test for a label, i.e., if L denotes label l then $l().\overline{R}(s)$ is triggered as continuation. Similarly, if L denotes label n then $n().\overline{R}(t)$ is triggered as continuation. This scheme corresponds roughly to Milner's encoding of boolean values [64]. If L denotes neither l nor n then $\overline{E}()$ is triggered to indicate that a runtime error has occurred (*undefined label*).

Labels are also encoded as replicated processes that wait for a tuple of continuation channels and signal along their designated label channel. Without lose of generality, we map labels to names. In case there is a conflict with an existing name in the agent to be translated we α-convert this agent using fresh names. For the form $\langle l = s \rangle \langle n = t \rangle$ we need to define two label processes where we use x_1 as continuation channel for label l and x_2 as continuation channel for label n, respectively:

$$!l(x_1, x_2).\overline{x_1}()$$
$$!n(x_1, x_2).\overline{x_2}()$$

The translation of agents is relatively straightforward. In the translation of an input-prefixed $\pi\mathcal{L}$-agent $a(X).A$, the input prefix $a(X)$ is also used as input prefix for the π-process. However, in the π-process X is now a name.

An output-particle $\overline{a}(F)$ (for simplicity we use a simple name here) is translated into a parallel π-process $(\nu\ f)(\overline{a}(f) \mid P)$ where $\overline{a}(f)$ emits along the original channel a the location of the form process (located at channel f). The right-hand side process P implements the form.

Finally, a projection X_l is translated into a parallel composition of an output-particle $\overline{X}(l, r, Error)$ that triggers the label projection process for X to send along a fresh channel r the value of the projection if label l is defined and an input-prefixed process $r(v_l).P$ that instantiates a name v_l in P with the value received along channel r. For example, the $\pi\mathcal{L}$-agent

$$\overline{a}(\langle l = s\rangle\langle n = t\rangle) \mid a(X).\overline{X_l}(\langle m = X_n\rangle)$$

is translated into the following π-process:

$$\begin{array}{l}
(\nu\ Error)(\nu\ l)(\nu\ n)(\nu\ m)\\
\quad(\ !Error().\mathbf{0} \mid !n_1(x_1,x_2,x_3).\overline{x_1}() \mid !n_2(x_1,x_2,x_3).\overline{x_2}() \mid !n_3(x_1,x_2,x_3).\overline{x_3}()\\
\quad\mid (\nu\ f)(\ \overline{a}(f) \mid !f(L,R,E).\\
\qquad\quad(\nu\ n,c)(\ \overline{L}(c,n,c)\\
\qquad\qquad\qquad\mid n().\overline{R}(t)\\
\qquad\qquad\qquad\mid c().(\nu\ f')(\ \overline{f'}(L,R,E)\\
\qquad\qquad\qquad\qquad\mid f'(L,R,E).\\
\qquad\qquad\qquad\qquad\qquad(\nu\ l,c')(\ \overline{L}(l,c',c')\\
\qquad\qquad\qquad\qquad\qquad\qquad\mid l().\overline{R}(s)\\
\qquad\qquad\qquad\qquad\qquad\qquad\mid c'().(\nu\ f'')(\ \overline{f''}(L,R,E)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mid f''(L,R,E).\overline{E}())))))\\
\mid a(X).(\nu\ r)(\overline{X}(l,r,Error)\\
\qquad\quad\mid r(v_l).(\nu\ f)(\overline{v_l}(f)\\
\qquad\qquad\qquad\mid !f(L,R,E).\\
\qquad\qquad\qquad\quad(\nu\ m,c)(\ \overline{L}(c,c,m)\\
\qquad\qquad\qquad\qquad\qquad\mid m().(\nu\ r')(\overline{X}(k,r',Error) \mid r(v_k).\overline{R}(v_k))\\
\qquad\qquad\qquad\qquad\qquad\mid c().(\nu\ f)(\ \overline{f}(L,R,E) \mid f(L,R,E).\overline{E}())))))
\end{array}$$

In the translation of projection we use the global channel *Error* as error continuation. In fact, *Error* is the location of the process that handles label applications for labels not defined in the actual form. However, it is important to note that signaling along the error continuation channel can only occur in the presence of polymorphic form extension. For example, a polymorphic form

$$XY\langle l = s\rangle\langle n = t\rangle$$

is translated into the following replicated process:

$$
\begin{array}{l}
!f(L,R,E).\\
\quad(\nu\ n,c)(\ \overline{L}(c,n)\\
\qquad\quad|\ n().\overline{R}(t)\\
\qquad\quad|\ c().\\
\qquad\qquad(\nu\ f')(\ \overline{f'}(L,R,E)\\
\qquad\qquad\quad|\ f'(L,R,E).\\
\qquad\qquad\qquad(\nu\ l,c')(\ \overline{L}(l,c')\\
\qquad\qquad\qquad\quad|\ l().\overline{R}(s)\\
\qquad\qquad\qquad\quad|\ c'().\\
\qquad\qquad\qquad\qquad(\nu\ f'')(\ \overline{f''}(L,R,E)\\
\qquad\qquad\qquad\qquad\quad|\ f''(L,R,E).\\
\qquad\qquad\qquad\qquad\qquad(\nu\ c'')(\ \overline{Y}(L,R,c'')\\
\qquad\qquad\qquad\qquad\qquad\quad|\ c''().\\
\qquad\qquad\qquad\qquad\qquad\qquad(\nu\ f''')(\ \overline{f'''}(L,R,E)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad|\ f'''(L,R,E).\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad(\nu\ c''')(\ \overline{X}(L,R,c''')\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad|\ c''().\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad(\nu\ f^{IV})(\overline{f^{IV}}(L,R,E)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad|\ f^{IV}(L,R,E).\overline{E}())))))))))
\end{array}
$$

Now, if apply a label m to form $XY\langle l{=}s\rangle\langle n{=}t\rangle$, then we check for label m from right to left, i.e., we initiate the test for label n ($\overline{L}(c,n)$) and then for label l ($\overline{L}(l,c')$). Since label m is not defined in $\langle l{=}s\rangle\langle n{=}t\rangle$, we check now form variable Y ($\overline{Y}(L,R,c'')$) and if Y does not define a binding for label m, we forward the request to form variable X ($\overline{X}(L,R,c''')$). If neither Y nor X defines a binding for label m, then $\overline{E}()$ is triggered to indicate a runtime error.

For the translation we use the function $\mathcal{C}[\![]\!]^{\pi}_{\phi}$ that takes a $\pi\mathcal{L}$-agent and returns the corresponding π-process. Within the translation, ϕ is a mapping from the set $\mathcal{L}$ of labels to the set $\mathcal{I}$ of integers. Without lose of generality ϕ maps to $1,...,n$ where n is the number of labels used in the $\pi\mathcal{L}$-agent to be translated.

The translation $\mathcal{C}[\![]\!]^{\pi}_{\phi}$ is only defined on closed agents. This constraint is due to the definition of the weak $\mathcal{L}$-bisimulation which is only defined on closed agents.

The translation $\mathcal{C}[\![]\!]^{\pi}_{\phi}$ has two steps:

1. Collect labels of the agent and assign each label a unique number (established by function ϕ).

2. Translate agent using ϕ.

$$\mathcal{C}[\![A]\!]^{\pi}_{\phi} \;=\; (\nu\ Error)(_{\forall\ l\in\mathcal{LA}(A)}\ \nu\ n_{\phi_{\mathcal{LA}(A)}(l)})\ (!Error().\mathbf{0} \mid \prod^{\forall\ l\in\mathcal{LA}(A)} !n_{\phi_{\mathcal{LA}(A)}(l)}(x_1,...,x_{card(\mathcal{LA}(A))}).\overline{x_{\phi_{\mathcal{LA}(A)}(l)}}() \mid \mathcal{C}_A[\![A]\!]^{\pi}_{\phi}\ )$$

$$\begin{array}{rcl}
\mathcal{C}_A[\![\mathbf{0}]\!]^{\pi}_{\phi} &=& \mathbf{0}\\
\mathcal{C}_A[\![A \mid B]\!]^{\pi}_{\phi} &=& \mathcal{C}_A[\![A]\!]^{\pi}_{\phi} \mid \mathcal{C}_A[\![B]\!]^{\pi}_{\phi}\\
\mathcal{C}_A[\![(\nu\ a)A]\!]^{\pi}_{\phi} &=& (\nu\ a)\mathcal{C}_A[\![A]\!]^{\pi}_{\phi}\\
\mathcal{C}_A[\![!V(X).A]\!]^{\pi}_{\phi} &=& !\mathcal{C}_A[\![V(X).A]\!]^{\pi}_{\phi}
\end{array}$$

$$\mathcal{C}_A[\![V(X).A]\!]^{\pi}_{\phi} \;=\; \begin{cases} x(X).\mathcal{C}_A[\![A]\!]^{\pi}_{\phi}, & if\ V = x\\ (\nu\ r)(\overline{Y}(l,r,Error) \mid r(v_l).v_l(X).\mathcal{C}_A[\![A]\!]^{\pi}_{\phi}, & if\ V = Y_l \end{cases}$$

$$\mathcal{C}_A[\![\overline{V}(F)]\!]^{\pi}_{\phi} \;=\; \begin{cases} (\nu\ f)(\overline{x}(f) \mid !\mathcal{C}_f^{\mathcal{LA}(A)}[\![F]\!]^{\pi}_{\phi}), & if\ V = x\\ (\nu\ r)(\overline{X}(l,r,Error) \mid r(v_l).((\nu\ f)(\overline{v_l}(f) \mid !\mathcal{C}_f^{\mathcal{LA}(A)}[\![F]\!]^{\pi}_{\phi}))), & if\ V = X_l \end{cases}$$

$$\begin{array}{rcl}
\mathcal{C}_f^{\mathcal{LA}(A)}[\![\mathcal{E}]\!]^{\pi}_{\phi} &=& f(L,R,E).\overline{E}()\\
\mathcal{C}_f^{\mathcal{LA}(A)}[\![X]\!]^{\pi}_{\phi} &=& f(L,R,E).\overline{X}(L,R,E)\\
\mathcal{C}_f^{\mathcal{LA}(A)}[\![FX]\!]^{\pi}_{\phi} &=& f(L,R,E).(\nu\ c)(\ \overline{X}(L,R,c)\\
&& \qquad \mid c().(\nu\ f')(\overline{f'}(L,R,E) \mid \mathcal{C}_{f'}^{\mathcal{LA}(A)}[\![F]\!]^{\pi}_{\phi})\ )\\
\mathcal{C}_f^{\mathcal{LA}(A)}[\![F\langle l{=}x\rangle]\!]^{\pi}_{\phi} &=& f(L,R,E).(\nu\ l,c)(\ \mathcal{C}_{label}^{\mathcal{LA}(A)}[\![l,c]\!]^{\pi}_{\phi}\\
&& \qquad \mid l().\overline{R}(x)\\
&& \qquad \mid c().(\nu\ f')(\overline{f'}(L,R,E) \mid \mathcal{C}_{f'}^{\mathcal{LA}(A)}[\![F]\!]^{\pi}_{\phi})\ )\\
\mathcal{C}_f^{\mathcal{LA}(A)}[\![F\langle l{=}X_k\rangle]\!]^{\pi}_{\phi} &=& f(L,R,E).(\nu\ l,c)(\ \mathcal{C}_{label}^{\mathcal{LA}(A)}[\![l,c]\!]^{\pi}_{\phi}\\
&& \qquad \mid l().(\nu\ r)(\overline{X}(k,r,Error) \mid r(v_k)\overline{R}(v_k))\\
&& \qquad \mid c().(\nu\ f')(\overline{f'}(L,R,E) \mid \mathcal{C}_{f'}^{\mathcal{LA}(A)}[\![F]\!]^{\pi}_{\phi})\ )
\end{array}$$

$$\begin{array}{rcl}
\mathcal{C}_{label}^{\mathcal{LA}(A)}[\![l,c]\!]^{\pi}_{\phi} &=& \overline{L}(x_1,...,x_{j-1},x_j,x_{j+1},...,x_n),\\
&& with\ x_j = l;\ x_1,...,x_{j-1},x_{j+1},...,x_n = c\\
&& and\ \phi_{\mathcal{LA}(A)}(l) = j
\end{array}$$

$$\mathcal{C}[\![\mu]\!]^{\pi} = \begin{cases} \tau, & \tau\\ a(f), & if\ \mu = a(\widetilde{\langle l{=}b\rangle})\\ (\nu\ f)\overline{a}(f), & if\ \mu = \overline{a}(\widetilde{\langle l{=}b\rangle})\ or\ \mu = (\nu\ \tilde{x})\overline{a}(\widetilde{\langle l{=}b\rangle}) \end{cases}$$

Table 4.5: The translation $\mathcal{C}[\![]\!]^{\pi}_{\phi}$.

Definition 4.13 (Labels of an agent) *The set of labels of an agent A, written $\mathcal{LA}(A)$, is inductively given by:*

$$\begin{aligned}
\mathcal{LA}(\mathbf{0}) &= \emptyset \\
\mathcal{LA}(A_1|A_2) &= \mathcal{LA}(A_1) \cup \mathcal{LA}(A_2) \\
\mathcal{LA}((\nu\ x)A) &= \mathcal{LA}(A) \\
\mathcal{LA}(a(X).A) &= \mathcal{LA}(A) \\
\mathcal{LA}(Y_l(X).A) &= \{l\} \cup \mathcal{LA}(A) \\
\mathcal{LA}(\overline{a}(F)) &= \mathcal{LF}(F) \\
\mathcal{LA}(\overline{Y_l}(F)) &= \{l\} \cup \mathcal{LF}(F)
\end{aligned}$$

$$\begin{aligned}
\mathcal{LF}(\mathcal{E}) &= \emptyset \\
\mathcal{LF}(\mathcal{FX}) &= \mathcal{LF}(F) \\
\mathcal{LF}(F\langle l{=}a\rangle) &= \{l\} \cup \mathcal{LF}(F) \\
\mathcal{LF}(F\langle l{=}X_k\rangle) &= \{l,k\} \cup \mathcal{LF}(F)
\end{aligned}$$

Now, the function ϕ is parameterized with $\mathcal{LA}(A)$ such that

$$\forall\ l \in \mathcal{LA}(A) \Longrightarrow \phi_{\mathcal{LA}(A)}(l) = i$$

with $i \in 1, ..., card(\mathcal{LA}(A))$.

The translation $\mathcal{C}[\![]\!]^\pi_\phi$ is given in Table 4.5. The translation $\mathcal{C}[\![]\!]^\pi_\phi$ generates a π-process that consists of three parts: (i) the restrictions for the mapped labels (here $(_{\forall\ l\in\mathcal{LA}(A)}\ \nu\ n_{\phi_{\mathcal{LA}(A)}(l)})$ generates a restricted name for all label in $\mathcal{LA}(A)$ where $\phi_{\mathcal{LA}(A)}(l)$ maps a label l to a unique integer $i \in \mathcal{I}$), (ii) the label processes, and (iii) the translated agent. The function $\mathcal{C}[\![]\!]^\pi_\phi$ uses three subfunctions: (i) $\mathcal{C}_A[\![]\!]^\pi_\phi$ that translates a $\pi\mathcal{L}$-agent, (ii) $\mathcal{C}_f^{\mathcal{LA}(A)}[\![]\!]^\pi_\phi$ that translates a $\pi\mathcal{L}$-form into a π-process located at channel f, and (iii) $\mathcal{C}_{label}^{\mathcal{LA}(A)}[\![]\!]^\pi_\phi$ that generates the test output-particle for the actual form.

Finally, $\mathcal{C}[\![]\!]^\pi$ translates $\pi\mathcal{L}$-actions into π-actions. It is important to note that $\mathcal{C}[\![\mu]\!]^\pi$ does not preserve the structure of the action. For example, both $\overline{a}(\langle\widetilde{l{=}b}\rangle)$ and $(\nu\ \tilde{x})\overline{a}(\langle\widetilde{l{=}b}\rangle)$ are translated into $(\nu\ f)\overline{a}(f)$. The reason is that the translation $\mathcal{C}[\![]\!]^\pi_\phi$ adds one level of indirection for the communication of forms. More precisely, a $\pi\mathcal{L}$-form is translated into a replicated π-process located at channel f. We can think of this channel as a "pointer" to a record. In order to access a value of the record we have to send a selector (the channel that maps the corresponding label) along channel f. This operation is comparable with the pointer manipulation operator –> in C/C++.

We show now that $\mathcal{C}[\![]\!]^\pi_\phi$ is faithful, i.e., if two agents A and B are weakly $\mathcal{L}$-bisimular, then this implies that $\mathcal{C}[\![A]\!]^\pi_\phi \approx \mathcal{C}[\![B]\!]^\pi_\phi$.

Lemma 4.11
Let A be a $\pi\mathcal{L}$-agent, $X \in \mathrm{fv}(A)$, X_l a subterm of A, and b be subject of μ. Then if $\mathcal{C}[\![A\{\langle l{=}b\rangle/X\}]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} \mathcal{C}[\![A'\{\langle l{=}b\rangle/X\}]\!]^\pi_\phi$ it holds that

$$\mathcal{C}[\![A]\!]^\pi_\phi\{f/X\} \stackrel{\tau}{\Longrightarrow} \cdot \xrightarrow{(\nu\ r)\overline{f'}(l,r,Error)} \cdot \xrightarrow{r(b)} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^\pi} \cdot \stackrel{\tau}{\Longrightarrow} \mathcal{C}[\![A']\!]^\pi_\phi\{f/X\}$$

with f be the location of $\mathcal{C}^{\mathcal{LA}(A)}_f[\![X]\!]^\pi_\phi$.

Proof: We have, by assumption, that

$$\mathcal{C}[\![A\{\langle l{=}b\rangle/X\}]\!]^\pi_\phi \stackrel{\tau}{\Longrightarrow} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^\pi} \cdot \stackrel{\tau}{\Longrightarrow} \mathcal{C}[\![A'\{\langle l{=}b\rangle/X\}]\!]^\pi_\phi$$

Now, a projection X_l is translated into π-calculus fragment:

$$(\nu\ r)(\overline{X}(l,r,Error) \mid r(v_l).P$$

where P is a π-process that uses v_l. Applying $\{f/X\}$ to this fragment, we get

$$(\nu\ r)(\overline{f}(l,r,Error) \mid r(v_l).P)$$

In order the perform an action along b with is denoted by X_l, the translated agent performs two communications with

$$\xrightarrow{(\nu\ r)\overline{f'}(l,r,Error)} \cdot \xrightarrow{r(b)} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^\pi}$$

as required. □

In general, unlike in the translation from the π-calculus into the $\pi\mathcal{L}$-calculus, for a $\pi\mathcal{L}$-agent A with $X \in \mathrm{fv}(A)$ it does not hold that $\mathcal{C}[\![A\{\widetilde{\langle l{=}b\rangle}/X\}]\!]^\pi_\phi = \mathcal{C}[\![A]\!]^\pi_\phi\{f/X\}$. Moreover, since forms are encoded as replicated processes and a concrete access to name that is bound by a form requires two additional communications (Lemma 4.11), we have that $\mathcal{C}[\![A\{\widetilde{\langle l{=}b\rangle}/X\}]\!]^\pi_\phi \not\approx \mathcal{C}[\![A]\!]^\pi_\phi\{f/X\}$. However, if a $\pi\mathcal{L}$-agent A has a weak transition μ then it holds that $\mathcal{C}[\![A]\!]^\pi_\phi$ has a weak transition $\mathcal{C}[\![\mu]\!]^\pi$ too with the same subject.

Lemma 4.12
Let A be a closed $\pi\mathcal{L}$-agent and μ be a $\pi\mathcal{L}$-action. Then if $A \stackrel{\mu}{\Longrightarrow} A'$ then there exists a π-process P such that $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} P$.

Proof: We proceed by induction on the structure of A with $A \not\equiv \mathbf{0}$. By assumption agent A is closed. Therefore, the outermost subject part of A must be a simple name and the outermost object part does not contain unbound form variables. We consider the most significant cases.

- $A = a(X).A_1$.

  Then we have $\mathcal{C}[\![A]\!]^\pi_\phi = a(X).\mathcal{C}[\![A_1]\!]^\pi_\phi$. It holds that $a(X).A_1 \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A_1\{\widetilde{\langle l=b\rangle}/X\}$ and $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{a(f)}{\Longrightarrow} P$ with $P = \mathcal{C}[\![A_1]\!]^\pi_\phi\{f/X\}$.

- $A = \overline{a}(\widetilde{\langle l=b\rangle})$.

  Then we have $\mathcal{C}[\![A]\!]^\pi_\phi = (\nu\ f)(\overline{a}(f) \mid !\mathcal{C}^{\mathcal{LA}(A)}_f[\![\widetilde{\langle l=b\rangle}]\!]^\pi_\phi)$. It holds that $\overline{a}(\widetilde{\langle l=b\rangle}) \stackrel{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} \mathbf{0}$ and $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{(\nu\ f)\overline{a}(f)}{\Longrightarrow} P$ with $P = 0 \mid !\mathcal{C}^{\mathcal{LA}(A)}_f[\![\widetilde{\langle l=b\rangle}]\!]^\pi_\phi$ □

Lemma 4.13 *Let A be a closed $\pi\mathcal{L}$-agent, μ be a $\pi\mathcal{L}$-action and P be a π-process. Then if $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} P$ implies that there exists an agent A' such that $A \stackrel{\mu}{\Longrightarrow} A'$.*

Proof: We proceed by induction on the structure of A with $A \not\equiv \mathbf{0}$. By assumption agent A is closed. Therefore, the outermost subject part of A must be a simple name and the outermost object part does not contain unbound form variables. We consider the most significant cases.

- $A = a(X).A_1$.

  Then we have $\mathcal{C}[\![A]\!]^\pi_\phi = a(X).\mathcal{C}[\![A_1]\!]^\pi_\phi$. It holds that $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![a(\widetilde{\langle l=b\rangle})]\!]^\pi}{\Longrightarrow} P$ with $P = \mathcal{C}[\![A_1]\!]^\pi_\phi\{f/X\}$, $a(X).A_1 \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ with $A' = A_1\{\widetilde{\langle l=b\rangle}/X\}$ and $a(f) = \mathcal{C}[\![a(\widetilde{\langle l=b\rangle})]\!]^\pi$.

- $A = \overline{a}(\widetilde{\langle l=b\rangle})$.

  Then we have $\mathcal{C}[\![A]\!]^\pi_\phi = (\nu\ f)(\overline{a}(f) \mid !\mathcal{C}^{\mathcal{LA}(A)}_f[\![\widetilde{\langle l=b\rangle}]\!]^\pi_\phi)$. It holds that $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\overline{a}(\widetilde{\langle l=b\rangle})]\!]^\pi}{\Longrightarrow} P$ with $P = 0 \mid !\mathcal{C}^{\mathcal{LA}(A)}_f[\![\widetilde{\langle l=b\rangle}]\!]^\pi_\phi$, $\overline{a}(\widetilde{\langle l=b\rangle}) \stackrel{\overline{a}(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A'$ with $A' = \mathbf{0}$ and $(\nu\ f)\overline{a}(f) = \mathcal{C}[\![\overline{a}(\widetilde{\langle l=b\rangle})]\!]^\pi$. □

Theorem 4.4 *For each pair of closed $\pi\mathcal{L}$-agents A and B, $\mathcal{C}[\![]\!]^\pi_\phi$ is an injective mapping, such that if $A \stackrel{\mathcal{L}}{\approx} B$, then it holds*

$$\mathcal{C}[\![A]\!]^\pi_\phi \ \approx\ \mathcal{C}[\![B]\!]^\pi_\phi$$

Proof: We show that the relation

$$\mathcal{R} = \{\ (\mathcal{C}[\![A]\!]^\pi_\phi, \mathcal{C}[\![B]\!]^\pi_\phi)\ \mid\ A\ \stackrel{\mathcal{L}}{\approx} B\ \} \cup\ \approx$$

is a weak (1)-bisimulation.
Consider first the case τ or *output actions*:

> The condition $\mathrm{bn}(\mathcal{C}[\![\mu]\!]^\pi) \cup \mathrm{fn}(\mathcal{C}[\![A]\!]^\pi_\phi \mid \mathcal{C}[\![B]\!]^\pi_\phi) = \emptyset$ can easily be established by α-convert the $\mathcal{C}[\![\mu]\!]^\pi$ since the names of the bound names in the action are unimportant as long as they denote the same location.
>
> Suppose $\mathrm{bn}(\mathcal{C}[\![\mu]\!]^\pi) \cup \mathrm{fn}(\mathcal{C}[\![A]\!]^\pi_\phi \mid \mathcal{C}[\![B]\!]^\pi_\phi) = \emptyset$. Now let $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} P$. By Lemma 4.13 there exists an agent A' such that $A \stackrel{\mu}{\Longrightarrow} A'$.
>
> The definition of $\stackrel{\mathcal{L}}{\approx}$ guarantees that there exists B' such that $B \stackrel{\mu}{\Longrightarrow} B'$ and $A' \stackrel{\mathcal{L}}{\approx} B'$. By Lemma 4.12, $\mathcal{C}[\![B]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} P'$. Thus we have $P \mathcal{R} P'$.
>
> In the case of an output action, we know, by Lemma 4.11, that next weak transition $\mathcal{C}[\![\mu]\!]^\pi$ cannot occur before all names which are part of μ have been communicated. For each name, we have two communications. Since $P \mathcal{R} P'$, it follows that both must perform the same name requests.

Consider now the case with composition with output:

> Since $A \stackrel{\mathcal{L}}{\approx} B$, we have $\mathcal{C}[\![A]\!]^\pi_\phi \approx \mathcal{C}[\![B]\!]^\pi_\phi$. Then for all messages $\overline{a}(\tilde{b})$ we have by definition $(\mathcal{C}[\![A]\!]^\pi_\phi \mid \overline{a}(\tilde{b}), \mathcal{C}[\![B]\!]^\pi_\phi \mid \overline{a}(\tilde{b})) \in \mathcal{R}$. □

The following theorem shows that weak 1-bisimilarity in π of translated $\pi\mathcal{L}$-agents implies weak $\mathcal{L}$-bisimilarity in $\pi\mathcal{L}$ for these agents.

Theorem 4.5 *For each pair of closed $\pi\mathcal{L}$-agents A and B, if $\mathcal{C}[\![A]\!]^\pi_\phi \approx \mathcal{C}[\![B]\!]^\pi_\phi$, then it holds $A \stackrel{\mathcal{L}}{\approx} B$.*

Proof: We show that the relation

$$\mathcal{R} = \{\ (A, B)\ \mid\ \mathcal{C}[\![A]\!]^\pi_\phi \approx \mathcal{C}[\![B]\!]^\pi_\phi\ \} \cup \stackrel{\mathcal{L}}{\approx}$$

is a weak $\mathcal{L}$-bisimulation.
Consider first the case τ or *output actions*:

> The condition $\mathrm{bn}(\mu) \cup \mathrm{fn}(A \mid B) = \emptyset$ can easily be established by α-convert the μ since the names of the bound names in the action are unimportant as long as they denote the same location.
>
> Suppose $\mathrm{bn}(\mu) \cup \mathrm{fn}(A \mid B) = \emptyset$. Now let $A \stackrel{\mu}{\Longrightarrow} A'$. By Lemma 4.12 there exists a π-process P such that $\mathcal{C}[\![A]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} P$.
>
> The definition of $\approx$ guarantees that there exists Q such that $\mathcal{C}[\![B]\!]^\pi_\phi \stackrel{\mathcal{C}[\![\mu]\!]^\pi}{\Longrightarrow} Q$ and $P \approx Q$. By Lemma 4.13, $B \stackrel{\mu}{\Longrightarrow} B'$. Thus we have $A' \mathcal{R} B'$.

In the case of an output action, we know, by Lemma 4.11, that next weak transition $\mathcal{C}[\![\mu]\!]^\pi$ cannot occur before all names which are part of μ have been communicated. For each name, we have two communications. Since $P \approx Q$, it follows that both must perform the same name requests. Furthermore, P and Q perform more moves than A' and B', but if the next move for P and Q is $\mathcal{C}[\![\mu]\!]^\pi$, then A and B can move for μ too.

Consider now the case with composition with output:

Since $\mathcal{C}[\![A]\!]^\pi_\phi \approx \mathcal{C}[\![B]\!]^\pi_\phi$, we have $A \overset{\mathcal{L}}{\approx} B$. Then for all messages $\overline{a}(\langle \widetilde{l=b} \rangle)$ we have by definition $(A \mid \overline{a}(\langle \widetilde{l=b} \rangle), B \mid \overline{a}(\langle \widetilde{l=b} \rangle)) \in \mathcal{R}$. □

Chapter 5

Types for $\pi\mathcal{L}$

The fundamental purpose of a type system is to prevent the occurrence of *run-time errors* while executing a program, i.e., types impose constraints which help to enforce correctness of a program [24]. However, a type system must be defined in such a way that a programmer can easily predict whether a program will typecheck or not [22]. In other words, a type system should be *transparent*. Furthermore, a type system should be *decidably verifiable*, i.e., there should be an algorithm that can ensure that a program is *well-typed*, and a type system should be *enforceable*, i.e., type declarations should be checked statically as much as possible [22].

The most useful type systems for programming languages are those which can by typechecked automatically (usually at compile time). We can consider then a type as a *specification* and typechecking as the *verification* [21]. However, both need to be defined is such a way that the programmer can always understand any type errors reported by the type system.

However, the process of declaring and manipulating type annotations can become tedious when programming in a typed language. Then we may use *type inference* to insert the missing type information automatically. Furthermore, like in the type systems of ML [61] or Pict [92], for some classes of (simple) programs the programmer does not even have to write any type information. In addition, type inference provides also useful debugging information. For example, if we expect a channel to have one type, and the type inference yields another, this may suggest a failure in the use of this channel.

The $\pi\mathcal{L}$-calculus is inherently polymorphic. In order to preserve the untyped flexibility in the typed $\pi\mathcal{L}$-calculus as much as possible, we define a first-order polymorphic type system as basic type system for the $\pi\mathcal{L}$-calculus. By the attribute polymorphic we mean that our type system supports *atomic subtype polymorphism* and that our type system does not have the types *Top* or *Bottom*.

The design of our type system for the $\pi\mathcal{L}$-calculus is driven by the fact that a type system, while providing static checking, should not impose excessive constraints

[21]. In fact, by subtype polymorphism we gain a system that enables us to define reliable components. Furthermore, from the software methodology point of view we get a system that is enabled for an *ordered extension* of existing software systems.

In this chapter we present an extended typed $\pi\mathcal{L}$-calculus. In a real programming language one needs some *constant* or *basic values* to build useful programs. Numbers, Boolean, and other data structures can be represented in the pure $\pi\mathcal{L}$-calculus using the scheme presented by Milner [64] or Turner [112] for the π-calculus. Therefore, adding *constant values* to the calculus does not change its semantics. However, if constant values are available, then calculations involving such values are more efficient because they can usually be done in one step.

5.1 Types and type contexts for $\pi\mathcal{L}$

In the typed (extended) $\pi\mathcal{L}$-calculus we indicate by *Basic* a collection of *ground types* like *Integer* and *String* for constant values. We use K to range over any such ground types. Furthermore, in the $\pi\mathcal{L}$-calculus we have basically two kinds of data: *channels* (names) and *forms*. Therefore, we have also a channel type constructor and a form type constructor.

Throughout this chapter we will use the term *value* for both *names* and *constants*. We assign every constant a corresponding ground type.

In order to enable us to do type inference, we also allow *type variables*.

5.1.1 Type contexts

The definition of the typed $\pi\mathcal{L}$-calculus is given in a context-free grammar. However, typing constraints are context sensitive. To take the types of variables into account, we use *type assertions* or *judgements*

$$\Gamma \vdash \Im$$

where Γ is a *static typing environment* of the form

$$\Gamma = \{v_1 : T_1, ..., v_n : T_n\}$$

assigning v_i type T_i and no v_i occurring twice. Intuitively, the assertion $\Gamma \vdash \Im$ says that all free variables of $\Im$ are declared in Γ, i.e., $\Im$ is well-formed. The empty environment is denoted by $\emptyset$, and the collection of variables $v_1, ..., v_n$ declared in Γ is indicated by $dom(\Gamma)$. We implicitly assume that all variables mentioned in Γ are distinct. For example, if we mention $\Gamma, v : T$ in a rule, then v is not already mentioned in Γ. The judgements for the typed $\pi\mathcal{L}$-calculus are given in Table 5.1.

In the $\pi\mathcal{L}$-calculus we have the set $\mathrm{fn}(A)$ and $\mathrm{fv}(A)$ denoting the free names and the free variables of an agent, respectively. As far typing is concerned, we will use the

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash K$	K is a well-formed constant type in Γ
$\Gamma \vdash T$	T is a well-formed type in Γ
$\Gamma \vdash A$	A is a well-formed agent in Γ
$\Gamma \vdash F$	F is a well-formed form in Γ
$\Gamma \vdash v : T$	value v has type T in Γ
$\Gamma \vdash F : T$	form F has type T in Γ
$\Gamma \vdash T <: S$	T is a subtype of S in Γ

Table 5.1: Judgments for the typed $\pi\mathcal{L}$-calculus.

term *variable* to denote both sets. In fact, we can define two distinct *namespaces* to record $\mathrm{fn}(A)$ and $\mathrm{fv}(A)$ separately in Γ. However, we leave this distinction implicit in our typing rules. Moreover, we can always apply α-conversion to guarantee that all variables in Γ are distinct.

5.1.2 Typing rules

Typing rules assert the validity of particular judgements on the basis of other judgements that are already known to be valid. The checking process starts usually with $\emptyset \vdash \diamond$, stating that the empty environment is well-formed. Now, the type system for the $\pi\mathcal{L}$-calculus is defined by a collection of rules of the following form:

$$\frac{\begin{array}{l}(\mathit{Rule\ name}) \\ \Gamma_1 \vdash \Im_1 \quad \ldots \quad \Gamma_n \vdash \Im_n \quad (\mathit{Annotations})\end{array}}{\Gamma \vdash \Im}$$

Each typing rule is written as a number of *premises* $\Gamma_i \vdash \Im_i$ above the horizontal line with a single *conclusion* $\Gamma \vdash \Im$ below the line. When all premises are satisfied, the conclusion must hold. All typing rules are named. The name of the typing rule is determined by the conclusion. For example, the rule name (*Input*) denotes the typing rule for the input-prefixed agent. When needed, *annotations* are added that state restricting conditions under which a rule can be applied.

5.1.3 Forms

In the (extended) typed $\pi\mathcal{L}$-calculus we extend the set V of values by constants. The syntax for forms is refined as follows:

$$\begin{array}{rcll} F & ::= & X & \textit{form variable} \\ & | & \mathcal{E} & \textit{empty binding} \\ & | & FX & \textit{polymorphic extension} \\ & | & F\langle l{=}V\rangle & \textit{binding extension} \end{array}$$

where

$$\begin{array}{rcll} V & ::= & c & \textit{constant} \\ & | & x & \textit{simple name} \\ & | & X_l & \textit{projection} \end{array}$$

Now, in the typed $\pi\mathcal{L}$-calculus, we have a different notion of *name projection*. If a label l is not defined in a form F and F substitutes X, then the projection X_l fails. This failure is denoted by *wrong*. In the untyped calculus we consider agents in which this failure occurs as *partially incorrect*. However, in the typed version these agents will not typecheck to guarantee that an agent which has passed the typechecker will not fail.

Definition 5.1 (Typed name projection) *If a form F is closed, i.e., $\mathcal{V}(F) = \emptyset$, and F substitutes form variable X, then the application of a label $l \in \mathcal{L}$ to X (mapping from $\mathcal{L}$ to $\mathcal{N}^+$), written X_l, is called typed name projection and is defined as:*

$$\begin{array}{rcl} \mathcal{E}_l & = & \textit{wrong} \\ (F\langle l{=}a\rangle)_l & = & a \\ (F\langle m{=}a\rangle)_l & = & F_l \quad \textit{if } m \neq l \end{array}$$

5.1.4 Syntax of the typed $\pi\mathcal{L}$-calculus

The syntax of the first-order typed $\pi\mathcal{L}$-calculus is given in Table 5.2. We use T, S, and U to range over types and M, N to range over type variables.

The constant *wrong* stands for an agent in which a runtime type error has occurred, e.g., one in which a projection yields a type error or an attempted communication involves a violation of the direction restriction (input only/output only).

5.1.5 Reduction semantics of the typed $\pi\mathcal{L}$-calculus

As for the untyped version we can define the operational semantics of the typed $\pi\mathcal{L}$-calculus using the relations *structural equivalence* and *reduction relation*. The new semantics differs only in the rules involving type annotations.

The structural congruence relation, $\equiv$, is the smallest congruence relation over typed agents that satisfies the axioms given in Table 5.3.

The reduction rules for the typed $\pi\mathcal{L}$-calculus are defined as follows:

T	$::=$	K	*Basic constant types*
	$\mid$	CT	*Channel type*
	$\mid$	FT	*Form type*
	$\mid$	M	*Type variable*
	$\mid$	$Wrong$	*Type error*
CT	$::=$	$\updownarrow FT$	*Bidirectional channel*
	$\mid$	$\downarrow FT$	*Input only channel*
	$\mid$	$\uparrow FT$	*Output only channel*
FT	$::=$	$\langle\rangle$	*Empty form type*
	$\mid$	$\langle l_1 : T_1 \rangle ... \langle l_n : T_n \rangle$	*Form type*
	$\mid$	$FT \backslash l$	*Form type restriction*
A	$::=$	$\mathbf{0}$	*Inactive agent*
	$\mid$	$A \mid A$	*Parallel composition*
	$\mid$	$!V(X)A$	*Replication*
	$\mid$	$(\nu\ a : T)A$	*Restriction*
	$\mid$	$V(X).A$	*Input (receive form in X)*
	$\mid$	$\overline{V}(F)$	*Output (send form F)*
	$\mid$	$wrong$	*Runtime error*

Table 5.2: Syntax of the typed $\pi\mathcal{L}$-calculus.

$$\text{PAR: } \frac{A \longrightarrow A'}{A \mid B \longrightarrow A' \mid B} \qquad \text{RES: } \frac{A \longrightarrow A'}{(\nu\ x : T)A \longrightarrow (\nu\ x : T)A'}$$

$$\text{COM: } x(X).A \mid \overline{x}.(F) \longrightarrow A\{F/X\},\ \textit{if}\ \mathcal{V}(F) = \emptyset$$

$$\text{STRUCT: } \frac{A \equiv A' \quad A' \longrightarrow B' \quad B' \equiv B}{A \longrightarrow B}$$

The rules are almost identical with the untyped version except that type annotations have been added to the restriction. However, unlike in the polyadic π-calculus, we do not have a rule involving an arity mismatch [87].

(1) $A \mid B \equiv B \mid A,\ (A \mid B) \mid C \equiv A \mid (B \mid C),\ A \mid \mathbf{0} \equiv A$;

(2) $(\nu\ x : T)\mathbf{0} \equiv \mathbf{0},\ (\nu\ x : T)(\nu\ y : S)A \equiv (\nu\ y : S)(\nu\ x : T)A$;

(3) $(\nu\ x : T)A \mid B \equiv (\nu\ x : T)(A \mid B)$, if x not free in B;

(4) $!V(X).A \equiv V(X).A \mid !V(X).A$;

Table 5.3: Structural congruence rules for the $\pi\mathcal{L}$-calculus.

5.2 Basic typing rules

An empty environment is a valid environment. It does not require any assumptions.

$$(\textit{Empty environment})$$
$$\emptyset \vdash \diamond$$

The following rule is used to extend an environment Γ to a longer environment $\Gamma, v : T$, given that T is a valid type in Γ. Note that the premise $\Gamma \vdash T$ implies, inductively, that Γ is valid. Furthermore, we require that v is not already defined in Γ ($v \notin dom(\Gamma)$). Here, v can denote both *channels* and *form variables*.

$$(\textit{Environment } v)$$
$$\frac{\Gamma \vdash T \qquad v \notin dom(\Gamma)}{\Gamma, v : T \vdash \diamond}$$

The rules (*Constant type*) and (*Empty form type*) construct types for *constants* and the empty form type, respectively.

$$(\textit{Constant type})$$
$$\frac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K}$$

$$(\textit{Empty form type})$$
$$\Gamma \vdash \langle\rangle$$

The following rules (*Constant*) and (*Channel*) assign types the *constant values* and *channels*, respectively.

$$(\textit{Constant})$$
$$\frac{\Gamma \vdash K}{\Gamma \vdash c : K}$$

$$\frac{(Channel)\quad \Gamma, x : T \vdash \diamond}{\Gamma, x : T \vdash x : T}$$

5.3 Subtyping rules for types

The following rules capture basic facts of the subtype relation <: of types. The subtype relation consists of two structural rules

$$\frac{(Subtyping\ reflexivity)\quad \Gamma \vdash T}{\Gamma \vdash T <: T}$$

$$\frac{(Subtyping\ transitivity)\quad \Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

stating that it is reflexive and transitive.

The subsumption rule states that if a term has type T, and T is a subtype of S, then the term has also type S. That is, subtyping behaves very much like set inclusion, when type membership is seen as set membership [22]. Here, v stands for both *channels* and *form variables*.

$$\frac{(Subsumption)\quad \Gamma \vdash v : T \qquad \Gamma \vdash T <: S}{\Gamma \vdash v : S}$$

5.4 Typechecking forms

5.4.1 Operations on forms

In fact, forms are *extensible records*. Therefore, a record type model is best suited as typing scheme for forms. We have the following form types:

$\langle\rangle$	type of all forms.
$\langle\rangle\backslash l$	type of all forms which lack field of label l.
$\langle l_1 : T_1\rangle\langle l_2 : T_2\rangle$	form type which has at least bindings for label l_1 and l_2, with values of type T_1 and T_2.
$\langle l_1 : T_1\rangle\backslash l_2$	form type which has at least a binding for label l_1 of type T_1 and no binding for label l_2.

Hence a form type is characterized by a finite collection of *type bindings* (i.e., labeled types) and a finite collection of *restricted labels* (i.e., labels that must not occur in a given form). All bindings are unordered, however, they must have distinct labels.

We have two operations on form types:

- *Binding extension* $FT\langle l:T\rangle$: This type denotes the collection obtained from FT by extending it with a binding l of type T. There are two cases. First, if FT does not already contain a binding for l, i.e., the form to be extended has type $FT\backslash l$, then $\langle l:T\rangle$ extends FT while removing the restriction $\backslash l$. Second, if FT has already a binding for l, then, by definition, $\langle l:T\rangle$ overrides the binding in FT, yielding a form type in which all labels are distinct. The reader should note that binding extension is similar to the **with** operator for records in Pict [92]. Furthermore, there is no constraint on type T. In general, we can use an arbitrary type for T.

- *Polymorphic extension* FT_1FT_2: This is the most problematic operator. Technically, polymorphic extension is *asymmetric record concatenation* [23, 96, 122]. Polymorphic extension takes two forms and returns a new form composed of all bindings in any of its arguments. However, if both forms define the same binding, then the questions arises which type has to be assigned to the result form? Neither in the type systems defined by Wand [122] nor in that defined by Cardelli [23] will typecheck the resulting form. In the former concatenation has no *principle type* while in the latter concatenation is in fundamental conflict with the subsumption rule. Rémy [96] proposed an approach that replaces concatenation with one-field record extension, i.e., concatenation is translated into binding extensions. Harper and Pierce [41] proposed an alternative approach based on symmetric concatenation using *compatibility assumptions*. But the programmer has to specify all necessary constraints explicitly.

  However, we want to keep polymorphic extension as basic operation. Moreover, our main concern is *software composition*. For example, the process of adding a generic synchronization policy [55, 105] to a component can be considered as polymorphic extension. The resulting component must be compatible with the unsynchronized component in order to work uniformly in the same environment. Therefore, we require that the interface type of the policy be a subtype of the component's interface type. In other words, if FT_1 and FT_2 have a non-empty intersection of labels, i.e, $\mathcal{L}(FT_1) \cap \mathcal{L}(FT_2) \neq \emptyset$, then for all conflicting labeled types it must hold that $FT_{2l_i} <: FT_{1l_i}$.

  Unfortunately, we need a further restriction for the type FT_2 to preserve the subtype relation of polymorphic extension of FT_1FT_2. Suppose, we have defined an agent $x(X).A$ that waits along channel x for a form F that is a subtype of $\langle l_1:S_1\rangle\langle 1_2:T_2\rangle\langle l_3:S_3\rangle$ and an output-particle $\overline{x}(XY)$ that emits along x the form

value XY of type $\text{merge}(TS) = \langle l_1 : S_1 \rangle \langle 1_2 : T_2 \rangle \langle l_3 : S_3 \rangle$ with $X : T = \langle l_1 : T_1 \rangle \langle l_2 : T_2 \rangle$, $Y : S = \langle l_1 : S_1 \rangle \langle l_3 : S_3 \rangle$, and $S_1 <: T_1$. In this case both agents will typecheck, since

$$\text{merge}(TS) <: \langle l_1 : S_1 \rangle \langle 1_2 : T_2 \rangle \langle l_3 : S_3 \rangle.$$

However, if the actual types of values X and Y are R and U with $R <: T$ and $U <: S$, $U = \langle l_1 : S_1 \rangle \langle l_2 : U_2 \rangle \langle l_3 : S_3 \rangle$ and $U_2 \not<: T_2$, then the polymorphic extension XY violates the subtype relation, since

$$\text{merge}(RU) = \langle l_1 : S_1 \rangle \langle l_2 : U_2 \rangle \langle l_3 : S_3 \rangle \not<: \langle l_1 : S_1 \rangle \langle 1_2 : T_2 \rangle \langle l_3 : S_3 \rangle.$$

The problem is that the subtype relations $R <: T$ and $U <: S$ are not sufficient. We need an additional restriction on type S and all its subtypes – that is, polymorphic extension of two forms X and Y with type R and U is only allowed, if it holds that $R <: T$, $U <: S$, and $\forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S),\ S <: \langle\rangle \backslash l_k$, which always holds for type S, but adds an constraint to all subtypes of S. Therefore, in order to typecheck the output-particle $\overline{x}(XY)$, it must be the case that $X : T$ with $T <: \langle l_1 : T_1 \rangle \langle l_2 : T_2 \rangle$ and $Y : S$ with $S <: \langle l_1 : S_1 \rangle \langle l_3 : S_3 \rangle \backslash l_2$, and $S_1 <: T_1$.

5.4.2 Form types

The simplest form is $\mathcal{E}$ – the empty binding. This form has no bindings and hence has type $\langle\rangle$ and is consistent with any context Γ:

$$\frac{}{\Gamma \vdash \mathcal{E} : \langle\rangle} \ (Empty\ form)$$

The typing rule for form variables

$$\frac{\Gamma, X : T \vdash \diamond \qquad \Gamma \vdash T <: \langle\rangle}{\Gamma, X : T \vdash X : T} \ (Form\ variable)$$

states that if the context $\Gamma, X : T$ is a well-formed environment and T is a subtype of $\langle\rangle$, then X has type T. Note, the premise $\Gamma \vdash T <: \langle\rangle$ ensures that T is given a valid form type.

In order to typecheck *binding extension* and *polymorphic extension*, we need to introduce the following definitions.

Definition 5.2 (Labels of a form type) *The set of labels of a form type FT, written $\mathcal{L}(FT)$, is defined as:*

$$\begin{aligned} \mathcal{L}(\langle\rangle) &= \emptyset \\ \mathcal{L}(FT\langle l : T \rangle) &= \{l\} \cup \mathcal{L}(FT) \end{aligned}$$

Definition 5.3 (Form type projection) *The application of a label l to a form type FT, written FT_l, is defined as:*

$$\begin{aligned} \langle\rangle_l &= \mathit{Wrong} \\ (FT\langle l{:}T\rangle)_l &= T \\ (FT\langle m{:}T\rangle)_l &= FT_l \ \mathit{if}\ m \neq l \end{aligned}$$

Definition 5.4 *The operation* merge *on two form types T and S, written* merge(TS)*, removes all multiple bindings from the concatenation of T and S. In the resulting form type, all common labeled types have been overridden from right-to-left.*

$$\text{merge}(TS) = \begin{cases} \langle\rangle, \ \mathit{if}\ \mathcal{L}(T) \cup \mathcal{L}(S) = \emptyset & \\ \langle l_1{:}U_1\rangle ... \langle l_n{:}U_n\rangle, & \mathit{with}\ l_1, ..., l_n \in \mathcal{L}(T) \cup \mathcal{L}(S), \\ & \mathit{and}\ U_i = \begin{cases} S_i, \ \mathit{if}\ S_i \neq \mathit{Wrong} \\ T_i, \ \mathit{otherwise} \end{cases} \end{cases}$$

Now, we can define typing rules for *binding extension* and *polymorphic extension*. The typing rule for *binding extension* $F\langle l{=}V\rangle$ checks that F is assigned a form type under Γ (premise $\Gamma \vdash T <: \langle\rangle$). If, in the same context Γ, V has type S, then $F\langle l{=}V\rangle$ has type merge$(T\langle l{:}S\rangle)$. Note, type T, as in Pict [92], may or may not contain label l.

$$\frac{\begin{array}{l}(\mathit{Binding\ extension}) \\ \Gamma \vdash F : T \qquad \Gamma \vdash T <: \langle\rangle \qquad \Gamma \vdash V : S\end{array}}{\Gamma \vdash F\langle l{=}V\rangle : \text{merge}(T\langle l{:}S\rangle)}$$

The typing rule for *polymorphic extension* FX checks that both F and X denote valid forms, i.e., they have been assigned the form types T and S, respectively (premises $\Gamma \vdash T <: \langle\rangle$ and $\forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle\backslash l_k$). The premise $\forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle\backslash l_k$ ensures that all subtypes of S lack also all labels only defined in T (for type S this is always true). Finally, if T and S have a non-empty intersection of labels, then for any such label it must hold that S_{l_i} is a subtype of T_{l_i} which is checked by restriction $\forall\ l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ S_{l_i} <: T_{l_i}$. If all premises are satisfied, FX is given type merge(TS).

$$\frac{\begin{array}{l}(\mathit{Polymorphic\ extension}) \\ \Gamma \vdash F : T \qquad \Gamma \vdash X : S \qquad \Gamma \vdash T <: \langle\rangle \qquad \forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle\backslash l_k \\ \forall\ l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \Gamma \vdash S_{l_i} <: T_{l_i}\end{array}}{\Gamma \vdash FX : \text{merge}(TS)}$$

The typing rule for *projection* ensures that the type of form variable X contains at least the used label l. If X has type $\langle l{:}T\rangle$, then projection X_l has type T under Γ.

$$\frac{\Gamma, X : T \vdash \diamond \qquad \Gamma \vdash T <: \langle\rangle}{\Gamma, X : T \vdash X : T}\ (Form\ variable) \qquad \Gamma \vdash \mathcal{E} : \langle\rangle\ (Empty\ form)$$

$$\frac{\Gamma \vdash F : T \qquad \Gamma \vdash X : S \qquad \Gamma \vdash T <: \langle\rangle \qquad \forall\, l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle \backslash l_k \qquad \forall\, l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \Gamma \vdash S_{l_i} <: T_{l_i}}{\Gamma \vdash FX : \mathrm{merge}(TS)}\ (Polymorphic\ extension)$$

$$\frac{\Gamma \vdash F : T \qquad \Gamma \vdash T <: \langle\rangle \qquad \Gamma \vdash V : S}{\Gamma \vdash F\langle l = V\rangle : \mathrm{merge}(T\langle l : S\rangle)}\ (Binding\ extension)$$

$$\frac{\Gamma \vdash X : \langle l : T\rangle}{\Gamma \vdash X_l : T}\ (Projection)$$

Table 5.4: Form typing rules for the $\pi\mathcal{L}$-calculus.

$$\frac{\Gamma \vdash X : \langle l : T\rangle}{\Gamma \vdash X_l : T}\ (Projection)$$

The typing rules for forms are summarised in Table 5.4.

5.4.3 Form subtyping

In the following we define a collection of rules describing the subtyping behaviour of the form type constructor. The following definitions, especially those involving label restriction, are inspired by the proposal of Cardelli and Mitchell [23]. The first two rules capture the subtype relation between well-formed forms and type $\langle\rangle$. In fact, these rules correspond to the usual subtyping rules for records.

$$\frac{\Gamma \vdash T_1 \quad \ldots \quad \Gamma \vdash T_n \quad l_i\ distinct}{\Gamma \vdash \langle l_1 : T_1\rangle \ldots \langle l_n : T_n\rangle <: \langle\rangle}\ (Form\ subtyping\ empty\ form)$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \ldots \quad \Gamma \vdash T_n <: S_n \quad \Gamma \vdash T_{n+1} \quad \ldots \quad \Gamma \vdash T_{n+m} \quad l_i\ distinct}{\Gamma \vdash \langle l_1 : T_1\rangle \ldots \langle l_{n+m} : T_{n+m}\rangle <: \langle l_1 : S_1\rangle \ldots \langle l_n : S_n\rangle}\ (Form\ subtyping)$$

The rule (*Subtyping restriction*) states that every well-formed restricted form type is a subtype of $\langle\rangle$, whereas rule (*Subtyping restricted types*) state that if form type S is a subtype of form type T, then this relation is preserved under restriction.

$$\frac{\Gamma \vdash T <: \langle\rangle}{\Gamma \vdash T\backslash l <: \langle\rangle} \quad (\textit{Subtyping restriction})$$

$$\frac{\Gamma \vdash T <: \langle\rangle \qquad \Gamma \vdash S <: T}{\Gamma \vdash S\backslash l <: T\backslash l} \quad (\textit{Subtyping restricted types})$$

The last rule for form subtyping states that if form type S is a subtype of form type T, then $S\backslash l$ is also a subtype of T iff $l \notin \mathcal{L}(T)$. In other words, a form type with more label restrictions is a subtype of a form type with fewer restrictions.

$$\frac{\Gamma \vdash T <: \langle\rangle \qquad \Gamma \vdash S <: T \qquad l \notin \mathcal{L}(T)}{\Gamma \vdash S\backslash l <: T} \quad (\textit{Subtyping add restriction})$$

5.5 Typechecking channels

In the following we define a collection of rules describing the subtyping behaviour of the channel type constructors $\updownarrow$, $\uparrow$, and $\downarrow$. Basically, the rules are the same as in Pict [92]. The refinement of the channel type constructor $\updownarrow$ was shown first by Pierce and Sangiorgi [87] in order to manipulate separately input and output capabilities of channels. The constructor $\updownarrow$ is invariant in the subtype relation. Therefore, a channel type $\updownarrow S$ is a subtype of $\updownarrow T$ only if S and T are equivalent.

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: S}{\Gamma \vdash \updownarrow S <: \updownarrow T} \quad (\textit{Channel subtyping})$$

The constructors $\uparrow$ and $\downarrow$ have a more interesting subtyping behaviour: $\uparrow$ is contravariant and $\downarrow$ is covariant. For example, given a channel x being used in a context only to read forms of type T, then is it is safe to replace x by another channel y carrying forms of type S, as long as it holds that S is a subtype of T. Conversely, given a channel x being used in a context only to write forms of type S, then is it is safe to replace x by another channel y carrying forms of type T, as long as it holds that T is a subtype of S.

$$\frac{(\textit{Output channel subtyping})\quad \Gamma \vdash T <: S}{\Gamma \vdash \uparrow S <: \uparrow T}$$

$$\frac{(\textit{Input channel subtyping})\quad \Gamma \vdash S <: T}{\Gamma \vdash \downarrow S <: \downarrow T}$$

Finally, $\updownarrow T$ is a subtype of both $\uparrow T$ and $\downarrow T$. That is, a channel x may be used for both output and input even though only one capability is actually being needed.

$$(\textit{Output channel channel subtyping})\quad \Gamma \vdash \updownarrow T <: \uparrow T$$

$$(\textit{Input channel channel subtyping})\quad \Gamma \vdash \updownarrow T <: \downarrow T$$

5.6 Typechecking agents

Like in the π-calculus, agents have no explicit results in the $\pi\mathcal{L}$-calculus. Interaction with agents is only possible by communicating with them. Therefore, typing judgements for agents take simply the form $\Gamma \vdash A$. In fact, $\Gamma \vdash A$ has to be read as asserting that A uses its free variables consistently with the types given in Γ.

The simplest $\pi\mathcal{L}$-calculus agent is the inactive agent, $\mathbf{0}$. It cannot communicate at all and hence is consistent with any context Γ:

$$(\textit{Null})\quad \Gamma \vdash \mathbf{0}$$

The output agent $\overline{V}(F)$ sends the form F along the channel denoted by V. The Output typing rule

$$\frac{(\textit{Output})\quad \Gamma \vdash V : \uparrow T \qquad \Gamma \vdash F : T \qquad \Gamma \vdash T <: \langle\rangle}{\Gamma \vdash \overline{V}(F)}$$

states that if, in a context Γ, V denotes an output enabled channel carrying a form of type T, then $\overline{V}(F)$ is a well-formed agent. Note, the premise $\Gamma \vdash \mathrm{T} : \langle\rangle$ guarantees that the type of F is a subtype of the empty form, i.e., that F is always a valid form value.

The input-prefixed agent $V(X).A$ receives a form along the channel denoted by V, binding the received form to X in A. The Input typing rule

$$\frac{(\mathit{Input}) \quad \Gamma \vdash V : \downarrow T \qquad \Gamma \vdash T <: \langle\rangle \qquad \Gamma, X : T \vdash A}{\Gamma \vdash V(X).A}$$

checks that V denotes an input enabled channel carrying a form of type T, and that A is well-formed in the context Γ extended with the type of the bound form variable X. Note, the premise $\Gamma \vdash \mathrm{T} : \langle\rangle$ guarantees that T is a subtype of the empty form, i.e., that X is always a valid form value. Furthermore, we assume that X is distinct from all form variables already bound in Γ (it is always possible to satisfy this condition by α-converting the bound form variable).

The typing rule for $A \mid B$ must ensure that A and B use their free variables in a consistent manner. Therefore, we require that A and B are well-formed in the same context Γ. This ensures that any names or form variables which are used in both A and B must have the same type.

$$\frac{(\mathit{Parallel\ composition}) \quad \Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \mid B}$$

The restriction operator $(\nu\ a : T)A$ introduces a new channel a in the scope of A. The typing rule for restriction extends the context Γ by adding a type binding for a. Additionally, we force the type assigned to a to be a channel type.

$$\frac{(\mathit{Restriction}) \quad \Gamma, a : \updownarrow T \vdash A}{\Gamma \vdash (\nu\ a : \updownarrow T)A}$$

Finally, the replication operator, $!V(X).A$, serves to make a countably infinite number of copies of $V(X).A$ in parallel. However, the consistency of $!V(X).A$ depends only on the fact that $V(X).A$ uses its free variables consistently under context Γ:

$$\frac{(\mathit{Replication}) \quad \Gamma \vdash V(X).A}{\Gamma \vdash\ !V(X).A}$$

The typing rules for agents are summarised in Table 5.5.

5.7 Type soundness

The relation between the type system and the operational semantics can be expressed in two forms: *evaluation cannot fail*, and *reduction preserves typing* [93]. The former detects the immediate failure of an agent, while the latter provides a proof that well-formed agents remain well-formed after a successful reduction step.

$$(Null)\quad \Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mid B}\ (Parallel\ composition) \qquad \frac{\Gamma \vdash V(X).A}{\Gamma \vdash\ !V(X).A}\ (Replication)$$

$$\frac{\Gamma, a : \updownarrow T \vdash A}{\Gamma \vdash (\nu\ a : \updownarrow T)A}\ (Restriction)$$

$$\frac{\Gamma \vdash V : \downarrow T \quad \Gamma \vdash T <: \langle\rangle \quad \Gamma, X : T \vdash A}{\Gamma \vdash V(X).A}\ (Input)$$

$$\frac{\Gamma \vdash V : \uparrow T \quad \Gamma \vdash F : T \quad \Gamma \vdash T <: \langle\rangle}{\Gamma \vdash \overline{V}(F)}\ (Output)$$

Table 5.5: Agent typing rules for the $\pi\mathcal{L}$-calculus.

5.7.1 Properties of well-formed $\pi\mathcal{L}$-terms

If v is not a *free variable* in A, i.e., $v \notin \mathrm{fn}(A)$ and $v \notin \mathrm{fv}(A)$, then we can add a new type binding for v without invalidating the typing of A:

Lemma 5.1 (Weakening)
If $\Gamma \vdash A$ *and* $v \notin \mathrm{fn}(A)$ *and* $v \notin \mathrm{fv}(A)$ *then* $\Gamma, v : T \vdash A$ *for some type* T.

Proof: A simple induction on the structure of A. □

Similarly, if v is not a *free variable* in A, i.e., $v \notin \mathrm{fn}(A)$ and $v \notin \mathrm{fv}(A)$, then we can remove the type binding for x without invalidating the typing of A:

Lemma 5.2 (Strengthening)
If $\Gamma, v : T \vdash A$ *and* $v \notin \mathrm{fn}(A)$ *and* $\notin \mathrm{fv}(A)$ *then* $\Gamma \vdash A$.

Proof: A simple induction on the structure of A. □

If a form variable X has type T in context Γ and a form F has the same type T under context Γ we can substitute F for X while preserving the type of A.

Lemma 5.3 (Substitution)
If $\Gamma \vdash A$, $\Gamma \vdash X : T$, *and* $\Gamma \vdash F : T$ *then* $\Gamma \vdash A\{F/X\}$.

Proof: A simple induction on the structure of A. □

5.7.2 Properties of structural congruence

Following the scheme presented by Turner [112] for the polyadic π-calculus, we can show that types are preserved under structural congruence. Like Turner, we add the following four rules to $\equiv$ given in Table 5.3 to enable us to use induction on the depth of the derivation of $A \equiv B$ in proofs.

$$A \equiv A \qquad \frac{A \equiv B}{B \equiv A}$$

$$\frac{A \equiv B \quad B \equiv C}{A \equiv C} \qquad \frac{A \equiv B}{\mathcal{C}[A] \equiv \mathcal{C}[B]}$$

$\mathcal{C}$ denotes an agent containing a "hole":

$$\mathcal{C} ::= [\cdot] \mid (\nu\ a : T)\mathcal{C} \mid (\mathcal{C} \mid A) \mid (A \mid \mathcal{C}) \mid !V(X).\mathcal{C} \mid V(X).\mathcal{C}$$

Lemma 5.4 (Types are preserved under structural congruence)

1. *If* $\Gamma \vdash A$ *and* $A \equiv B$ *then* $\Gamma \vdash B$.
2. *If* $\Gamma \vdash B$ *and* $A \equiv B$ *then* $\Gamma \vdash A$.

Proof: We proof both parts simultaneously, using induction on the depth of the inference of $A \equiv B$.

- Case $A \mid \mathbf{0} \equiv A$.

  Part 1. If $\Gamma \vdash A \mid \mathbf{0}$ then it must be the case that $\Gamma \vdash A$ as required. Part 2. We have, by assumption, that $\Gamma \vdash A$. Therefore, using rules (*Null*) and (*Parallel composition*), we have $\Gamma \vdash A \mid \mathbf{0}$ as required.

- Case $A \mid B \equiv B \mid A$.

  Part 1. If $\Gamma \vdash A \mid B$ then $\Gamma \vdash A$ and $\Gamma \vdash B$ and the result follows using rule (*Parallel composition*). Part 2 is similar.

- Case $(A \mid B) \mid C \equiv A \mid (B \mid C)$.

  Part 1. If $\Gamma \vdash (A \mid B) \mid C$ then $\Gamma \vdash A$, $\Gamma \vdash B$, and $\Gamma \vdash C$. The result follows after two applications of the rule (*Parallel composition*). Part 2 is similar.

- Case $(\nu\ x : \updownarrow T)\mathbf{0} \equiv \mathbf{0}$.

  Part 1. If $\Gamma \vdash (\nu\ x : \updownarrow T)\mathbf{0}$ then it must be the case that $\Gamma, x : \updownarrow T \vdash \mathbf{0}$ for some (form) type T. We can therefore use strengthening (Lemma 5.2) to prove that $\Gamma \vdash \mathbf{0}$ as required. Part 2. We have, by assumption, that $\Gamma \vdash \mathbf{0}$. Furthermore, we have that $x \notin \mathrm{fn}(\mathbf{0})$ and $x \notin \mathrm{fn}(\mathbf{0})$. We can therefore use weakening (Lemma 5.1) to prove that $\Gamma, x : \updownarrow T \vdash \mathbf{0}$ for some (form) type T as required.

- Case $(\nu\ x : \updownarrow T)(\nu\ y : \updownarrow S)A \equiv (\nu\ y : \updownarrow S)(\nu\ x : \updownarrow T)A$.

  Part 1. If $\Gamma \vdash (\nu\ x : \updownarrow T)(\nu\ y : \updownarrow S)A$ then $\Gamma, x : \updownarrow T, y : \updownarrow S \vdash A$ for some (form) types T and S. However, the order in which variables are declared in Γ is unimportant as long as we have $\Gamma \vdash \diamond$. Therefore, the result follows using rule (*Restriction*) twice. Part 2 is similar.

- Case $(\nu\ x : \updownarrow T)A \mid B \equiv (\nu\ x : \updownarrow T)(A \mid B)$, if x not free in B.

  Part 1. If $\Gamma \vdash (\nu\ x : \updownarrow T)A \mid B$ then it must be the case that $\Gamma, x : \updownarrow T \vdash A$ and $\Gamma \vdash B$ for some (form) type T. We can therefore use weakening (Lemma 5.1) to prove that $\Gamma, x : \updownarrow T \vdash B$, since $x \notin \mathrm{fn}(B)$ and $x \notin \mathrm{fv}(B)$. The result follows using rules (*Parallel composition*) and (*Restriction*).

  Part 2. If $\Gamma \vdash (\nu\ x : \updownarrow T)(A \mid B)$ then it must be the case that $\Gamma, x : \updownarrow T \vdash B$ and $\Gamma, x : \updownarrow T \vdash A$ for some (form) type T. We can therefore use strengthening (Lemma 5.2) to prove that $\Gamma \vdash B$, since $x \notin \mathrm{fn}(B)$ and $x \notin \mathrm{fv}(B)$. The result follows using rules (*Restriction*) and (*Parallel composition*).

- Case $!V(X).A \equiv V(X).A \mid !V(X).A$.

  Part 1. If $\Gamma \vdash !V(X).A$ then it must be the case that $\Gamma \vdash V(X).A$. Therefore, using the rule (*Parallel composition*), we have $\Gamma \vdash V(X).A \mid !V(X).A$ as required. Part 2. If $\Gamma \vdash V(X).A \mid !V(X).A$ then it must be that $\Gamma \vdash V(X).A$. The result follows using rule (*Replication*).

- Case $A \equiv A$.

  Immediate.

- Case $B \equiv A$ where $A \equiv B$.

  Part 1. We have by induction (Part 1) that $\Gamma \vdash B$. Part 2 is similar.

- Case $A \equiv C$ where $A \equiv B$ and $B \equiv C$.

  Part 1. We have, by induction that $\Gamma \vdash B$. Therefore, using induction again, we have $\Gamma \vdash C$ as required. Part 2 is similar.

- Case $\mathcal{C}[A] \equiv \mathcal{C}[B]$ where $A \equiv B$.

  A simple induction on the structure of $\mathcal{C}$ proves the result. □

5.7.3 Untypable faulty terms

Faulty terms denote (sub)-expression in which a runtime error has occurred.

Definition 5.5 (Faulty terms) *The faulty terms are those typed $\pi\mathcal{L}$-terms containing a sub-term of the form:*

$$\begin{array}{rl} X_l & \textit{with } X_l = \textit{wrong;} \\ V(X).A & \textit{where } V \textit{ does not denote a channel;} \\ \overline{V}(F) & \textit{where } V \textit{ does not denote a channel; or} \\ x(X).A \mid \overline{x}(F) & \textit{where } \mathcal{V}(F) \neq \emptyset. \end{array}$$

Lemma 5.5 (Faulty projection is untypable)
If a projection X_l is faulty, i.e. $X_l = \textit{wrong}$, then there are no Γ, T such that $\Gamma \vdash X_l : T$.

Proof: Suppose $\Gamma \vdash X_l : T$ and projection X_l is faulty. We proceed by case analysis, assuming for each case that the projection X_l can be typed, and deriving a contradiction, but there is only one case.

If $\Gamma \vdash X_l : T$, then, by (*Projection*), it must be that $\Gamma \vdash X : \langle l{:}T \rangle$. Therefore, we have $X_l \neq \textit{wrong}$ (label l is valid and denotes a binding), which contradicts the assumption that projection X_l is faulty. □

Theorem 5.1 (Well-formed agents never fail)
If $\Gamma \vdash A$ then A does not fail, i.e., it does not contain any faulty terms.

Proof: Suppose $\Gamma \vdash A$ and A fails, i.e., it contains faulty terms. It suffices to show that the subterms of A that cause A to be faulty are untypable. We proceed by case analysis according to the forms of faulty subterms, assuming for each case that the terms can be typed, and deriving a contradiction.

- Case X_l with $X_l = \textit{wrong}$.

  We have, by assumption, that $\Gamma \vdash A$. Therefore, if X_l is a subterm in A, then it must be the case that $\Gamma \vdash X_l : T$. Using Lemma 5.5, we have a contradiction, as required.

- Case $V(X).A$ where V does not denote a channel.

  We have, by assumption, that $\Gamma \vdash V(X).A$. Therefore, by (*Input*), it must be the case that $\Gamma \vdash V : \downarrow T$, which implies that V denotes a (input) channel, contrary to the assumption.

- Case $\overline{V}(F)$ where V does not denote a channel.

  We have, by assumption, that $\Gamma \vdash \overline{V}(F)$. Therefore, by (*Output*), it must be the case that $\Gamma \vdash V : \uparrow T$, which implies that V denotes a (output) channel, contrary to the assumption.

- Case $x(X).A \mid \overline{x}(F)$ where $\mathcal{V}(F) \neq \emptyset$.

  We have, by assumption, that $\Gamma \vdash x(X).A$, $\Gamma \vdash \overline{x}(F)$ (by (*Parallel composition*)), and there exists a form variable $Y \in \mathcal{V}(F)$ with $Y \notin dom(\Gamma)$ (Y occurs unbound). However, if we have $\Gamma \vdash \overline{x}(F)$, then it must be the case that $\Gamma \vdash F : T$ and $Y \in dom(\Gamma)$, which is contrary to the assumption. □

The definition of faulty terms provides only the possibility to detect the immediate failures of an agent. To complete the soundness proof, we need a subject-reduction theorem that proves that well-formed agents remain well-formed after a successful reduction step.

5.7.4 Subject reduction

Subject reduction expresses the relationship between the operational semantics of a term and a typing of it. Now, a typing environment Γ can be though as agent A's "point of view" on the types of its free variables. Theorem 5.2 shows how this point of view evolves under transitions.

Theorem 5.2 (Subject reduction)
If $\Gamma \vdash A$ *and* $A \longrightarrow B$ *then* $\Gamma \vdash B$.

Proof: We prove the result by induction on the depth of the inference. We consider each reduction rule as the last rule applied in the inference of the antecedent $A \longrightarrow B$.

- Case $A \mid B \longrightarrow A' \mid B$ where $A \longrightarrow A'$.

  Then $\Gamma \vdash A$ and $\Gamma \vdash B$ follow from $\Gamma \vdash A \mid B$ by (*Parallel composition*). We can therefore use induction to prove that $\Gamma \vdash A'$. The result follows using the rule (*Parallel composition*).

- Case $(\nu\ x : \updownarrow T)A \longrightarrow (\nu\ x : \updownarrow T)A'$ where $A \longrightarrow A'$.

  From $\Gamma \vdash (\nu\ x : \updownarrow T)A$ we have $\Gamma, x : \updownarrow T \vdash A$ by (*Restriction*). We can therefore use induction to prove that $\Gamma, x : \updownarrow T \vdash A'$. The result follows using the rule (*Restriction*).

- Case $x(X).A \mid \overline{x}(F) \longrightarrow A\{F/X\}$.

  We have, by assumption, that $\Gamma \vdash x(X).A$ and $\Gamma \vdash \overline{x}(F)$. Therefore, it must be that $\Gamma, X : T \vdash A$ and $\Gamma \vdash x : \updownarrow T$ (since x is used for both input and output

in the same context Γ. Using the rule (*Subsumption*) we get the corresponding direction). However, we also have that $\Gamma \vdash F : T$. We can therefore use the substitution lemma (Lemma 5.3) to prove that $\Gamma, X : T \vdash A\{F/X\}$. The result follows by using the strengthening lemma (Lemma 5.2) since $X \notin \text{fn}(A\{F/X\})$ and $X \notin \text{fv}(A\{F/X\})$.

- Case $A \longrightarrow B$ where $A \equiv A'$, $A' \longrightarrow B'$, and $B' \equiv B$.

  We have, by assumption, that $\Gamma \vdash A$. Therefore, using Lemma 5.4, we have $\Gamma \vdash A'$. Using induction we have $\Gamma \vdash B'$ and the result follows by using Lemma 5.4 again. □

Corollary 5.1 (No runtime errors)
If $\Gamma \vdash A$ *and* $A \Longrightarrow A'$ *then* A' *does not fail.*

Proof: The result follows using Theorem 5.1, Theorem 5.2, and Lemma 5.4. □

Corollary 5.1 completes the soundness prove for our type system.

5.8 Type inference

Type inference is the process of the transforming an untyped or "partially-typed" term into a well-formed term by inferring the missing type annotations, if any such type exists. In the following we present an unification based inference algorithm that takes an untyped $\pi\mathcal{L}$-term A and returns a type substitution σ (if any exists) such that the typed version of A is well-formed with respect to a typing environment $\Gamma\sigma$.

In order to define a relation between typed and untyped $\pi\mathcal{L}$-terms, we define an *erasure function Erase* that removes all type annotations from a given typed $\pi\mathcal{L}$-term.

Definition 5.6 (Erase)

$$\begin{aligned}
Erase(\mathbf{0}) &= \mathbf{0} \\
Erase(A_1 \mid A_2) &= Erase(A_1) \mid Erase(A_2) \\
Erase(!A) &= !Erase(A) \\
Erase((\nu\ a : T)A) &= (\nu\ x)Erase(A) \\
Erase(V(X).A) &= V(X).Erase(A) \\
Erase(\overline{V}(F)) &= \overline{V}(F)
\end{aligned}$$

The type inference problem can now be defined as follows: given an untyped $\pi\mathcal{L}$-term A, find a typed $\pi\mathcal{L}$-term $\Gamma \vdash A'$ with $Erase(A') = A$.

T	::=	K	*Basic types*
	|	CT	*Channel type*
	|	FT	*Form type*
	|	M	*Type variable*
	|	$Wrong$	*Type error*
CT	::=	$\updownarrow FT$	*Bidirectional channel*
	|	$\downarrow FT$	*Input only channel*
	|	$\uparrow FT$	*Output only channel*
FT	::=	$\langle\rangle$	*Empty form type*
	|	$\langle l_1 : T_1 \rangle ... \langle l_n : T_n \rangle$	*Form type*
	|	$FT \backslash l$	*Form type restriction*
	|	$FT\langle M \rangle$	*Form tag*
	|	$FT \backslash \langle M \rangle$	*Lack tag*

Table 5.6: Refined syntax of the types for the $\pi\mathcal{L}$-calculus.

5.8.1 Extended form types

In order to do type inference, we need to refine the syntax of form types. Suppose, we have defined the following agent A:

$$A \equiv a(X).(\overline{b}(X) \mid \overline{X_l}(\langle n = X_m \rangle))$$

Now, to infer the type for channel a in agent A, we know that a form received along a and bound to X must contain bindings for label l and m due to the projections X_l and X_m, respectively. However, this information is not sufficient, since X is also sent along channel b, and the type of channel b is not yet available (it will be inferred from another agent). Therefore, we add a *form tag*, written $\langle M \rangle$, to form types. Such a tag can be thought of as a placeholder for a form type to be determined during the type inference process. Once the type for channel b has been determined, the form tag is substituted by the corresponding form type. For example, if we start the type inference for agent A, we assign X the type $\langle l : M_1 \rangle \langle m : M_2 \rangle \langle M_3 \rangle$. Now, if we can deduce type $\updownarrow \langle m : N_1 \rangle \langle n : N_2 \rangle$ for channel b, then we can substitute $\langle m : N_1 \rangle \langle n : N_2 \rangle$ for M_3 yielding $\langle l : M_1 \rangle \langle m : M_2 \rangle \langle n : N_2 \rangle$ as type for X. Note that multiple bindings are ignored (e.g. $\langle m : N_1 \rangle$). The type inference process will unify M_2 and N_1 in order to denote the same type. If the unification fails, the type inference process fails.

A second problem is introduced by polymorphic form extension. From the typing rule (*Polymorphic extension*) we know that if X has type T and Y has type S, then type S must be a subtype of a restricted form type, i.e., labels that only occur in T

must not occur in any subtypes of S. Using type inference, however, we deduce the set of restricting labels only at the time the type merge(TS) has been inferred, i.e., the types T and S have already been determined. To add the restricting labels to type S, we have to "patch" type S. Therefore, we introduce a *lack tag*, written $\backslash\langle M\rangle$. For example, we have defined the following agent

$$B \equiv a(X).b(Y).\overline{c}(XY)$$

then we start the type inference for agent B assigning X type $\langle M_1\rangle\backslash\langle M_2\rangle$ and Y type $\langle M_3\rangle\backslash\langle M_4\rangle$. Now, while inferring the types for channel a and b, we can substitute T for M_1 and S for M_3 such that the value XY has type merge($T\backslash\langle M_2\rangle, S\backslash\langle M_4\rangle$) yielding merge($TS$) as type for XY and a label restriction $\langle\rangle\backslash l_1...\backslash l_n$ as substitution for M_4 such that the final type for Y is $S\backslash l_1...\backslash l_n$. The label restriction M_2 becomes $\langle\rangle$.

The refined syntax for types used for type inference is given in Table 5.6.

In order to do type inference for *form values*, we need to define the set of restriction labels of a form type.

Definition 5.7 (Label restrictions of a form type) *The set of restriction labels of a form type FT, written $\mathcal{L}_R(FT)$, is defined as:*

$$\begin{aligned}\mathcal{L}_R(\langle\rangle) &= \emptyset \\ \mathcal{L}_R(FT\backslash l) &= \{l\} \cup \mathcal{L}_R(FT)\end{aligned}$$

Now, we can define type substitution.

5.8.2 Type substitution

A *type substitution*, or *substitution* for short, is a finite map from type variables to types. We write $T\{\widetilde{S}/\widetilde{M}\}$ for the substitution of all occurrences of type variables $\widetilde{M}$ with types $\widetilde{S}$ in T. We use σ and ρ to range over type substitutions, $\sigma(M)$ to denote the substitution for type variable M, and we let $dom(\sigma)$ denote the domain of σ. Substitutions naturally extend to both types and contexts.

Definition 5.8 (Type substitution)

$$\sigma, \rho \quad ::= \quad \{M_1 \mapsto S_1, ..., M_n \mapsto S_n\}$$

$$K\sigma \quad = \quad K$$

$$CT\sigma \quad = \quad \begin{cases} \updownarrow (FT\sigma) \\ \downarrow (FT\sigma) \\ \uparrow (FT\sigma) \end{cases}$$

$$FT\sigma \quad = \quad \begin{cases} \langle\rangle \\ \langle l_1 : T_1\sigma\rangle ... \langle l_n : T_n\sigma\rangle \\ FT\sigma \backslash l \end{cases}$$

$$FT\sigma\langle M\sigma\rangle \quad = \quad \begin{cases} FT\sigma\langle M\rangle, \ \textit{if} \ M\sigma = M \\ FT\sigma\langle l_1 : T_1\rangle ... \langle l_n : T_n\rangle, \\ \quad \textit{if} \ M\sigma = \langle l_1 : T_1\rangle ... \langle l_n : T_n\rangle \langle l_{n+1} : T_{n+1}\rangle ... \langle l_{n+m} : T_{n+m}\rangle \\ \quad \textit{and} \ l_1, ..., l_n \notin \mathcal{L}(FT) \end{cases}$$

$$FT\sigma\backslash\langle M\sigma\rangle \quad = \quad \begin{cases} FT\sigma\backslash\langle M\rangle, \ \textit{if} \ M\sigma = M \\ FT\sigma\backslash l_1 ... \backslash l_n, \ \textit{if} \ M\sigma = \langle\rangle\backslash l_1 ... \backslash l_n \end{cases}$$

$$M\sigma \quad = \quad \begin{cases} S, \ \textit{if} \ S \in \textit{dom}(\sigma) \\ M, \ \textit{otherwise} \end{cases}$$

$$\Gamma\sigma \quad = \quad \{ \ x : \text{removetags}(T\sigma) \mid x : T \in \Gamma \ \}$$

In the application of substitution for Γ, we use the function **removetags** to remove *open* form and lack tags from type $T\sigma$. An open tag is a tag with a type variable that is not in $dom(\sigma)$. There are two situations in which open tags can occur:

- Case lack tag $\backslash\langle M\rangle$.

  A lack tag $\backslash\langle M\rangle$ is added to every form type in the type inference process. But the lack tag variable M can only be identified with a type (i.e., a form type that consists solely of label restrictions), when the $\pi\mathcal{L}$-term contains at least one application of a polymorphic form extension, i.e., if no polymorphic form extension is used, then no lack tag will be unified. But open lack tags denote type $\langle\rangle$. We can therefore simply remove open lack tags from a form type without changing the type $T\sigma$.

- Case form tag $\langle M\rangle$.

  A form tag $\langle M\rangle$ is added by the type inference process to the application type of the form variable X of an input-prefixed agent $V(X).A$ if A contains an ap-

plication of X within a polymorphic form extension or if X is used as object of an output-particle. For example, in the agent

$$a(X).(\overline{a}(X) \mid \overline{X_l}(\langle m{=}X_n\rangle))$$

X is assigned the application type $\langle M_1\rangle\langle l : M_2\rangle\langle n : M_3\rangle$. But if the inferred type for X does not contain more bindings than l and n, then the type variable M_1 will never be identified with a type. A open form tag has type $\langle\rangle$. Therefore, it is safe to remove $\langle M_1\rangle$ from $T\sigma$.

Definition 5.9 (Composition of substitution)

$$\sigma \circ \rho = \{M \mapsto (M\sigma)\rho \mid M \in dom(\sigma) \cup dom(\rho)\}$$

If we apply a type substitution σ to a typed $\pi\mathcal{L}$-term A, the $A\sigma$ differs from A only in the types of the names introduced by the restriction operator. An important property of type substitution is that if $\Gamma \vdash A$ is derivable, so is any substitution $\Gamma\sigma \vdash A\sigma$. In other words, types are preserved under substitution.

Lemma 5.6 (Substitution preserves form type)
if $\Gamma \vdash T <: \langle\rangle$ *then* $\Gamma\sigma \vdash T\sigma <: \langle\rangle$.

PROOF: A simple induction on the structure of form type T. □

Lemma 5.7 (Substitution preserves channel type)
if $\Gamma \vdash T <: \updownarrow S$, $\Gamma \vdash T <: \downarrow S$, *or* $\Gamma \vdash T <: \uparrow S$ *then* $\Gamma\sigma \vdash T\sigma <: \updownarrow S\sigma$, $\Gamma\sigma \vdash T\sigma <: \downarrow S\sigma$, *and* $\Gamma\sigma \vdash T\sigma <: \uparrow S\sigma$, *respectively.*

PROOF: A simple induction on the structure of channel type T. □

Lemma 5.8 (Preservation of types under substitution)
If $\Gamma \vdash \Im$ *then* $\Gamma\sigma \vdash \Im\sigma$.

PROOF: The proof is by induction on the derivation of $\Gamma \vdash \Im$.

- Basic rules:
  - Case $\emptyset \vdash \diamond$.
    Immediate.
  - Case $\Gamma, v : T \vdash \diamond$.
    Then it must be the case that $\Gamma \vdash T$ and $v \notin dom(\Gamma)$. We can therefore use induction to prove $\Gamma\sigma \vdash T\sigma$. The result follows using rule (*Environment v*).

- Case $\Gamma \vdash K$.

  We assume that each constant has a variable-free type, and therefore $K\sigma = K$. From $\Gamma \vdash K$ we have $\Gamma \vdash \diamond$. We can therefore use induction to prove $\Gamma\sigma \vdash \diamond$. The result follows using rule (*Constant type*).

- Case $\Gamma \vdash \langle\rangle$.

  Immediate, using Lemma 5.6.

- Case $\Gamma \vdash c : K$.

  Then it must be the case that $\Gamma \vdash K$. We can therefore use induction to prove $\Gamma\sigma \vdash K\sigma$. The result follows using rule (*Constant*).

- Case $\Gamma, x : T \vdash x : T$.

  Then it must be the case that $\Gamma, x : T \vdash \diamond$. We can therefore use induction to prove $\Gamma\sigma, x : T\sigma \vdash \diamond$. The result follows using rule (*Channel*).

- Subtyping rules for types:

  - Case $\Gamma \vdash T <: T$.

    Then it must be the case that $\Gamma \vdash T$. We can therefore use induction to prove $\Gamma\sigma \vdash T\sigma$. The result follows using rule (*Subtyping reflexivity*).

  - Case $\Gamma \vdash T <: U$ where $\Gamma \vdash T <: S$, and $\Gamma \vdash S <: U$.

    We have by induction that $\Gamma\sigma \vdash T\sigma <: S\sigma$. Therefore, using induction again, we have $\Gamma\sigma \vdash S\sigma <: U\sigma$. The result follows using rule (*Subtyping transitivity*).

  - Case $\Gamma \vdash v : S$ where $\Gamma \vdash v : T$, and $\Gamma \vdash T <: S$.

    We have by induction that $\Gamma\sigma \vdash v : T\sigma$ and $\Gamma\sigma \vdash T\sigma <: S\sigma$. The result follows using rule (*Subsumption*).

- Rules for assigning types to forms:

  - Case $\Gamma \vdash \mathcal{E} : \langle\rangle$.

    Immediate, using Lemma 5.6.

  - Case $\Gamma, X : T \vdash X : T$ (form variable).

    Then it must be the case that $\Gamma, X : T \vdash \diamond$ and $\Gamma \vdash T <: \langle\rangle$. We can therefore use induction to prove $\Gamma\sigma, X : T\sigma \vdash \diamond$. The result follows using Lemma 5.6 and rule (*Form variable*).

  - Case $\Gamma \vdash F\langle l{=}V\rangle : \mathrm{merge}(T\langle l{:}S\rangle)$.

    Then it must be the case that $\Gamma \vdash F : T$, $\Gamma \vdash T <: \langle\rangle$, and $\Gamma \vdash V : S$. We can therefore use induction to prove $\Gamma\sigma \vdash F : T\sigma$ and $\Gamma\sigma \vdash V : S\sigma$, respectively. The result follows using Lemma 5.6 and (*Binding extension*).

- Case $\Gamma \vdash FX : \text{merge}(TS)$.

  Then it must be the case that $\Gamma \vdash F : T$, $\Gamma \vdash X : S$, $\Gamma \vdash T <: \langle\rangle$, $\forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle\backslash l_k$, and $\forall\ l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \Gamma \vdash S_{l_i} <: T_{l_i}$. We can therefore use induction to prove $\forall\ l_k \in \mathcal{L}(T\sigma) - \mathcal{L}(S\sigma)\ \Gamma\sigma \vdash S\sigma <: \langle\rangle\backslash l_k$, $\forall\ l_i \in \mathcal{L}(T\sigma) \cap \mathcal{L}(S\sigma)\ \Gamma\sigma \vdash (S\sigma)_{l_i} <: (T\sigma)_{l_i}$, $\Gamma\sigma \vdash F : T\sigma$, and $\Gamma\sigma \vdash X : S\sigma$, respectively. The result follows using Lemma 5.6 and rule (*Polymorphic extension*).

- Case $\Gamma \vdash X_l : T$.

  Then it must be the case that $\Gamma \vdash X : \langle l : T\rangle$. We can therefore use induction to prove $\Gamma\sigma \vdash X : \langle l : T\sigma\rangle$. The result follows using rule (*Projection*).

- Subtyping rules for form types:

  - Case $\Gamma \vdash \langle l_1 : T_1\rangle ... \langle l_n : T_n\rangle <: \langle\rangle$.

    Immediate, using Lemma 5.6.

  - Case $\Gamma \vdash \langle l_1 : T_1\rangle ... \langle l_n + m : T_n + m\rangle <: \langle l_1 : S_1\rangle ... \langle l_n : S_n\rangle$.

    Then it must be the case that $\Gamma \vdash T_i <: S_i$ for all $i \in 1, ..., n$ and $\Gamma \vdash T_j$ for all $j \in n+1, ..., n+m$. We can therefore use induction to prove $\Gamma\sigma \vdash T_i\sigma <: S_i\sigma$ for all $i \in 1, ..., n$ and $\Gamma\sigma \vdash T_j\sigma$ for all $j \in n+1, ..., n+m$. The result follows using rule (*Form subtyping*).

  - Case $\Gamma \vdash T\backslash l <: \langle\rangle$.

    Then it must be the case that $\Gamma \vdash T <: \langle\rangle$. The result follows using Lemma 5.6 and rule (*Subtyping restriction*).

  - Case $\Gamma \vdash S\backslash l <: T\backslash l$.

    Then it must be the case that $\Gamma \vdash T <: \langle\rangle$ and $\Gamma \vdash S <: T$. We can therefore use induction to prove $\Gamma\sigma \vdash S\sigma <: T\sigma$. The result follows using Lemma 5.6 and rule (*Subtyping restricted types*).

  - Case $\Gamma \vdash S\backslash l <: T$.

    Then it must be the case that $\Gamma \vdash T <: \langle\rangle$, $\Gamma \vdash S <: T$, and $l \notin \mathcal{L}(T)$. We can therefore use induction to prove $l \notin \mathcal{L}(T\sigma)$ and $\Gamma\sigma \vdash S\sigma <: T\sigma$. The result follows using Lemma 5.6 and rule (*Subtyping add restriction*).

- Subtyping rules for channels:

  - Case $\Gamma \vdash \updownarrow S <: \updownarrow T$.

    Then it must be the case that $\Gamma \vdash S <: T$ and $\Gamma \vdash T <: S$. We can therefore use induction to prove $\Gamma\sigma \vdash S\sigma <: T\sigma$ and $\Gamma\sigma \vdash T\sigma <: S\sigma$, respectively. The result follows using rule (*Channel subtyping*).

- Case $\Gamma \vdash \uparrow S <: \uparrow T$.

  Then it must be the case that $\Gamma \vdash T <: S$. We can therefore use induction to prove $\Gamma\sigma \vdash T\sigma <: S\sigma$. The result follows using rule (*Output channel subtyping*).

- Case $\Gamma \vdash \downarrow S <: \downarrow T$.

  Similar to previous case using (*Input channel subtyping*).

- Case $\Gamma \vdash \updownarrow T <: \uparrow T$.

  Immediate using Lemma 5.7.

- Case $\Gamma \vdash \updownarrow T <: \downarrow T$.

  Immediate using Lemma 5.7.

- Rules for agents (after application of *Erase*):

  - Case $\Gamma \vdash \mathbf{0}$.

    Immediate, since $\mathbf{0}$ is consistent with any context. Therefore, it must be the case that $\Gamma\sigma \vdash \mathbf{0}$, as required.

  - Case $\Gamma \vdash A \mid B$.

    Then it must be the case that $\Gamma \vdash A$ and $\Gamma \vdash B$. We can therefore use induction to prove $\Gamma\sigma \vdash A$ and $\Gamma\sigma \vdash B$, respectively. The result follows using the rule (*Parallel composition*).

  - Case $\Gamma \vdash !V(X).A$.

    Then it must be the case that $\Gamma \vdash V(X).A$. We can therefore use induction to prove $\Gamma\sigma \vdash V(X).A$. The result follows using the rule (*Replication*).

  - Case $\Gamma \vdash (\nu\ a)A$.

    Then it must be the case that $\Gamma, a : \updownarrow T \vdash A$. We can therefore use induction to prove $\Gamma\sigma, a : \updownarrow T\sigma \vdash A$. The result follows using rule (*Restriction*).

  - Case $\Gamma \vdash V(X).A$.

    Then it must be the case that $\Gamma \vdash V : \downarrow T$, $\Gamma \vdash T <: \langle\rangle$, and $\Gamma, X : T \vdash A$. We can therefore use induction to prove $\Gamma\sigma, X : T\sigma \vdash A$ and $\Gamma\sigma \vdash V : \downarrow T\sigma$, respectively. The result follows using Lemma 5.6, and rule (*Input*).

  - Case $\Gamma \vdash \overline{V}(F)$.

    Then it must be the case that $\Gamma \vdash V : \uparrow T$, $\Gamma \vdash F : T$, and $\Gamma \vdash T <: \langle\rangle$. We can therefore use induction to prove $\Gamma\sigma \vdash F : T\sigma$ and $\Gamma\sigma \vdash V : \uparrow T\sigma$, respectively. The result follows using Lemma 5.6 and rule (*Output*). □

5.8.3 Unification

An important part of the type inference algorithm is the unification process, i.e., the process of finding the most general unifying type substitution. If E is a set of pairs of expressions, then a type substitution σ *unifies* E if $T\sigma \equiv S\sigma$ for every equation $T = S \ \in E$.

The algorithm *Unify* recursively decomposes equations between compound types of the same "shape", substituting types for type variables when necessary. Basically, the algorithm is implemented following the scheme presented by Mitchell [67], Jategaonkar and Mitchell [46], and Thiemann [111] (the swap rule). However, it differs in the fact that we also use *equation predicates* that assemble form types and check the subtype relation, respectively. The following equation predicates are defined:

- $T = \text{PROJ}(l, S)$:

  PROJ performs the type projection yielding S_l when l is defined in S. If S does not have a binding for l, PROJ fails. This predicate is generated for every projection of the form X_l.

- $U = \text{MERGE}(T, S)$:

  MERGE removes all multiple bindings from TS. This predicate can only be applied to valid form types, i.e., they must denote types generated form the syntactic category FT. This predicate does not fail. The result is assigned to U yielding the equation $U = TS$. This predicate is generated for every *binding extension.*

- $U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)$:

  POLYMERGE removes all multiple bindings from TS. This predicate can only be applied to valid form types that must not contain any form tag. This predicate does not fail and yields two results: (i) TS, and (ii) the label restriction set for type S. The former is assigned to U yielding the equation $U = TS$, while the latter is used the generate a substitution $\{M_S \mapsto \langle\rangle\backslash l_1 ... \backslash l_n\}$. For example,

  $$U = \text{POLYMERGE}(\langle l_1 : T_1\rangle\langle l_2 : T_2\rangle\backslash\langle M_T\rangle, \langle l_2 : S_2\rangle\langle l_3 : S_3\rangle\backslash\langle M_S\rangle)$$

  yields

  $$U = \langle l_1 : T_1\rangle\langle l_2 : S_2\rangle\langle l_3 : S_3\rangle$$

  and

  $$\{M_S \mapsto \langle\rangle\backslash l_1\}$$

The condition $S_2 <: T_2$ is checked separately with an ELEMSUB predicate. The predicate POLYMERGE is generated for every application of polymorphic extension.

- SUB(T, S):

  SUB check, using the rules involving a subtype relation, whether T is a subtype of S. The predicate fails if this condition is not satisfied.

- SUBFORM(T, S):

  When T denotes a form type and S denotes a channel type, then SUBFORM checks whether values of type T can be sent along channels of type S. The predicate fails if this condition is not satisfied. This predicate is generated for every output-particle $\overline{V}(F)$ and it guarantees that the minimal type assigned to F respects the all type assignment of V. For example, if we have the two agents $\overline{a}(\langle l = c_1\rangle\langle m = c_2\rangle)$ and $\overline{a}(\langle l = c_1\rangle)$, then the minimal type for both forms $\langle l = c_1\rangle\langle m = c_2\rangle$ and $\langle l = c_1\rangle$ is $\langle l : K_1\rangle$ if $c_1 : K_1$. However, if there is also an input agent $a(X).A$ with $a : \downarrow \langle l : K_1\rangle\langle m : K_2\rangle$, then the minimal form type for values sent along a is $\langle l : K_1\rangle\langle m : K_2\rangle$ and agent $\overline{a}(\langle l = c_1\rangle)$ is therefore not well-formed. The predicate SUBFORM guarantees that this constraint is satisfied.

- ELEMSUB(T, S):

  ELEMSUB checks that for all $l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)$ it holds that S_{l_i} is a subtype of T_{l_i}. The predicate fails if this condition is not satisfied.

The algorithm *Unify* is nondeterministic, in that a set of equations could match more than one clause. However, we need to define the following constraints on the order of the selection of the next equation:

1. $E \cup \{T = S\}$:

   If S is not a type variable or any equation predicate, select $\{T = S\}$ as the next rule.

2. $E \cup \{T = M\}$:

   If E does not contain a rule that matches pattern 1, select $\{T = M\}$ as the next rule.

3. $E \cup \{S = \text{PROJ}(l, T)\}$ or $E \cup \{U = \text{MERGE}(T, S)\}$:

   If E does not contain a rule that matches pattern 1 or 2 and T and S denote types generated form the syntactic category FT, select $\{S = \text{PROJ}(l, T)\}$ and $\{U = \text{MERGE}(T, S)\}$ as the next rule, respectively.

4. $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$:

   If E does not contain a rule matching pattern 1-3 and T and S denote valid form types that must not contain any form tag, then the next rule to be processed is $\{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$.

5. SUBFORM, SUB, and ELEMSUB:

   Rules involving these predicates are processed if no other equations left in the equation set.

The algorithm *Unify* is shown in Appendix C. On equations involving channel types, the algorithm generates different subsequent type equations depending on the occurred channel type constructors. For an equations set of the form $E \cup \{\uparrow T_1 = \uparrow T_2\}$, $E \cup \{\updownarrow T_1 = \downarrow T_2\}$, or $E \cup \{\updownarrow T_1 = \uparrow T_2\}$ *Unify* generates a "contravariant" equation set of the form $E \cup \{T2 = T1\}$. This approach respects the subtyping relation between these channel type constructors. Otherwise, *Unify* identifies the wrong type variables such that *Unify* would fail. For all other channel type combinations *Unify* generates a "covariant" equation set. The reader should note that it is sufficient to generate a covariant subsequent equation set for an equation set $E \cup \{\updownarrow T_1 = \updownarrow T_2\}$. The invariance of T_1 and T_2 is preserved.

On equations involving form types, the algorithm does not need to perform an identity check on the complete types on both sides. Instead, if we have an equation like $FT_1 = FT_2$, then it is sufficient to generate only subequations for all labels $l_i \in \mathcal{L}(FT_1) \cap \mathcal{L}(FT_2)$. However, it always holds that either $\mathcal{L}(FT_1) - \mathcal{L}(FT_2) = \emptyset$ or $\mathcal{L}(FT_2) - \mathcal{L}(FT_1) = \emptyset$ which means that either $FT_2 <: FT_1$ or $FT_1 <: FT_2$ holds.

Finally, if the equation set E does not contain any predicate POLYMERGE, then it is safe to identify all lack tags $\backslash\langle M\rangle$ with $\langle\rangle$ in E.

Lemma 5.9 (Most general unifier[97])
If E is any set of equations between type expressions, then an algorithm Unify which terminates successfully on input E has as output a type substitution σ that is called the most general unifier for E. If E is not unifiable, then Unify(E) terminates yielding $fail$.

Proof: In order to prove Lemma 5.9, it is sufficient to show the following three properties of *Unify* [68]:

1. *Unify* halts for any set E.

2. If *Unify*(E) succeeds, then *Unify*(E) returns a substitution that unifies E.

3. If there is a substitution σ that unifies E, then *Unify*(E) succeeds and produces a substitution σ' such that σ' unifies E. Furthermore, there exists a substitution σ'' such that $\sigma = \sigma' \circ \sigma''$.

In order to prove (1), we define the *degree* of a set E to be the pair $\langle m, n\rangle$, where m is the number of distinct type variables in E, and n is the total number of occurrences of channel type constructors, form type constructors, base types, unbound form tags (i.e., they occur within a form type, and equations predicates. We say that $\langle m, n\rangle$ is smaller than $\langle m', n'\rangle$ if either $m < m'$, $m = m'$ and $n < n'$, or $m < m'$ and $n < n'$. If E is empty, it has degree $\langle 0, 0\rangle$.

The proof proceeds by induction on the degree of E and case analysis on the structure of the type equations of the form $T = S$, or $\{Predicate\}$ – equation predicates SUB, SUBFORM, and ELEMSUB, where $E = E' \cup \{T = S\}$ or $E = E' \cup \{Predicate\}$.

- Case $E \cup \{K_1 = K_2\}$.

  When *Unify* succeeds, we can observe that second component of the degree of E is at most $n - 2$.

- Case $E \cup \{M = T\}$.

  When *Unify* succeeds, we can observe that first component of the degree of $E\{T/M\}$ is at most $m - 1$.

- Case $E \cup \{T = M\}$.

  Here *Unify* succeeds if *Unify* succeeds on $E \cup \{M = T\}$ which corresponds to the previous case.

- Case $E \cup \{CT_1 = CT_2\}$.

  Here, both CT_1 and CT_2 are channel types and if *Unify* succeeds, then the channel type constructors are removed from both types. Therefore, the second component degree of the resulting equation set is at most $n - 2$.

- Case $E \cup \{FT_1 = FT_2\}$.

  - $FT_1 = T\backslash\langle M_{T_1}\rangle\langle M_{T_2}\rangle$ and $FT_2 = S\backslash\langle M_{S_1}\rangle\langle M_{S_2}\rangle$:
    When *Unify* succeeds, then we have four cases:
    - $E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = M_{S_2}\}$, where the second component of the degree is at most $n - 4$.
    - $E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{S_2} = Delta\} \cup \{M_{T_2} = \langle\rangle\}$, where the second component of the degree is at most $n - 3$.
    - $E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = Delta\} \cup \{M_{S_2} = \langle\rangle\}$, where the second component of the degree is at most $n - 3$.
    - $E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = Delta1\} \cup \{M_{S_2} = Delta2\}$, where the second component of the degree is at most $n - 2$.

- $FT_1 = T\backslash\langle M_{T_1}\rangle\langle M_{T_2}\rangle$ and $FT_2 = S\backslash\langle M_S\rangle$:
  When *Unify* succeeds, then we can observe that the second component of the degree of $E \cup \{M_T = M_{S_1}\} \cup \{M_{S_2} = Delta\}$ is at most $n-2$.
- $FT_1 = T\backslash\langle M_T\rangle$ and $FT_2 = S\backslash\langle M_{S_1}\rangle\langle M_{S_2}\rangle$:
  Similar to the previous case.
- $FT_1 = T\backslash\langle M_T\rangle$ and $FT_2 = S\backslash\langle M_S\rangle$:
  When *Unify* succeeds, then we can observe that the second component of the degree of $E \cup \{M_T = M_S\} \cup \{\ (T_{l_i} = S_{l_i}) \mid l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \}$ is at most $n-2$.

- Case $E \cup \{S = \text{PROJ}(l, T)\}$.

  When *Unify* succeeds, we can observe that second component of the degree of $E \cup \{S = T_l\}$ is at most $n-1$.

- Case $E \cup \{U = \text{MERGE}(T, S)\}$.

  When *Unify* succeeds, we can observe that second component of the degree of $E \cup \{U = TS\}$ is at most $n-2$.

- Case $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$.

  When *Unify* succeeds, then the degree of $E\{Lackform/M_S\} \cup \{U = TS\}$ is in the first component at most $m-1$ and in the second component at most $n-2$.

- Case $E \cup \{\text{SUB}(S, T)\}$.

  When *Unify* succeeds, we can observe that second component of the degree of E is at most $n-1$.

- Case $E \cup \{\text{SUBFORM}(FT, CT)\}$.

  Similar to the previous case.

- Case $E \cup \{\text{ELEMSUB}(T, S)\}$.

  Similar to the previous case. □

In order to prove (2) and (3), we use induction on the number of recursive calls in the computation of *Unify*(E). For (2), there are only two non-trivial steps.

- Case $E \cup \{M = T\}$.

  If σ unifies $E\{T/M\}$, then $\{M \mapsto T\} \circ \sigma$ unifies $E \cup \{M = T\}$.

- Case $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$.

  If σ unifies $E\{Lackform/M_S\} \cup \{U = TS\}$, then $\{M_S \mapsto Lackform\} \circ \sigma$ unifies $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$. □

  For (3), the base case $\emptyset$ is straightforward. For the induction step, there are 21 cases, one for each clause in the definition of the algorithm *Unify*. However, there are only two non-trivial clauses.

  - Case $E \cup \{M = T\}$.

    Suppose σ unifies $E \cup \{M = T\}$. Then $\sigma(M) = T\sigma$ and σ must unify $E\{T/M\}$. It is easy to see that M must not occur in T. By the induction hypothesis, $\mathit{Unify}(E\{T/M\}) = \sigma'$ with $\sigma = \sigma' \circ \sigma''$. Then it must be the case that $\mathit{Unify}(E \cup \{M = T\}) = \sigma' \circ \{T/M\}$ succeeds with a substitution unifying $E \cup \{M = T\}$. Now, we argue that $\sigma = (\sigma' \circ \{T/M\}) \circ \sigma''$. For any variable M' different from M, we have $M'((\sigma' \circ \{T/M\}) \circ \sigma'') = M'\sigma$ since $\sigma' \circ \sigma'' = \sigma$. For the variable M, we have $M((\sigma' \circ \{T/M\}) \circ \sigma'') = T(\sigma' \circ \sigma'') = T\sigma$. But since we have already seen that $M\sigma = T\sigma$, we have $M((\sigma' \circ \{T/M\}) \circ \sigma'') = M\sigma$ and therefore $\sigma = (\sigma' \circ \{T/M\})) \circ \sigma''$.
  - Case $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$.

    Suppose σ unifies $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$. Then, we have $\sigma(M_S) = Lackform\sigma$ and σ must unify $E\{Lackform/M_S\} \cup \{U = TS\}$. By the induction hypothesis, $\mathit{Unify}(E\{Lackform/M_S\} \cup \{U = TS\}) = \sigma'$ with $\sigma = \sigma' \circ \sigma''$. Therefore, $\mathit{Unify}(E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}) = \sigma' \circ \{Lackform/M_S\}$ succeeds with a substitution unifying the equation set $E \cup \{U = \text{POLYMERGE}(T\backslash\langle M_T\rangle, S\backslash\langle M_S\rangle)\}$. Now, we argue that $\sigma = (\sigma' \circ \{Lackform/M_S\}) \circ \sigma''$. For any variable M different from M_S, we have $M((\sigma' \circ \{Lackform/M_S\}) \circ \sigma'') = M\sigma$ since $\sigma' \circ \sigma'' = \sigma$. For variable M_S, we have $M_S((\sigma' \circ \{Lackform/M_S\}) \circ \sigma'') = Lackform(\sigma' \circ \sigma'') = Lackform\sigma$. But since we have already seen that $M_S\sigma = Lackform\sigma$, we have $M_S((\sigma' \circ \{Lackform/M_S\}) \circ \sigma'') = M_S\sigma$ and therefore it must hold that $\sigma = (\sigma' \circ \{Lackform/M_S\}) \circ \sigma''$. □

5.8.4 Inference algorithm

In the following we present an inference algorithm which takes a type context Γ and an untyped agent A as arguments, and either fails (if no valid typing exists), or returns the minimal substitution σ such that $\Gamma\sigma \vdash A$.

The algorithm is defined using similar approach as presented by Wand [121], i.e., the algorithm generates a type equation set E by processing a set of *subgoal rules* starting with $(\Gamma_0; A_0)$, where Γ_0 maps all free variables of A_0 and A_0 is the top-level agent after applying *Erase*. In fact, the algorithm mimics the construction of the type

derivation for a given agent A. If all subgoals are processed, then the algorithm halts and *Unify* is called to produce a most general unifier for E if such exists. The type inference algorithm fails if and only if *Unify fails*. We now give the structure of the algorithm:

Input:

A term A_0, where $A_0 = Erase(A)$ and A is a typed $\pi\mathcal{L}$-agent.

Initialization:

Set $E = \emptyset$ and $G = \{(\Gamma_0; A_0)\}$, where Γ_0 maps all free variables of A_0 to distinct type variables.

Loop Step:

If $G = \emptyset$, then halt and return E. Otherwise, choose a subgoal from G, delete if form G, and add to E and G new verification conditions and subgoals, as specified in the action table.

One important difference of this algorithm compared with the one presented by Turner [112] for the polyadic π-calculus is that unification is done at the end of the type inference process. In Turner's approach, *Unify* is called directly for every input and output process. This approach fails, however, in the presence of *polymorphic form extension*. For example, if we have defined the following $\pi\mathcal{L}$-system

$$S_1 \equiv (a(X).b(Y)\overline{c}(XY) \mid \overline{a}(F_1) \mid \overline{b}(F_2))$$

the unification of the equation set solely generated for $a(X).b(Y)\overline{c}(XY)$ would yield a substitution such that both X and Y are assigned type $\langle\rangle$. The reason is that the available information is not sufficient to correctly determine the types for X and Y. Therefore, unlike Turner's approach, we need to consider the complete agent system in order the infer the correct types. When we consider the whole system (i.e., we have generated the complete equation set E) of

$$S_2 \equiv (a(X).b(Y)\overline{c}(XY) \mid \overline{a}(\langle l{=}1\rangle\langle m{=}"String"\rangle) \mid \overline{b}(\langle m{=}"OtherString"\rangle\langle n{=}1\rangle))$$

then we can correctly infer $X : \langle l{:}Int\rangle\langle m{:}String\rangle$ and $Y : \langle m{:}String\rangle\langle n{:}Int\rangle\backslash l$ with $Int,\ String\ \in K$.

We do not formalise how the algorithm picks "fresh" type variables while processing the given subgoals. We will implicitly assume that whenever a type variable is declared to be "fresh" it is distinct from any type variables mentioned either in the current context or in rules which have already been processed. In practice this condition is satisfied by using a global counter to number new type variables.

Before we can present the subgoal rules, we need to define the format of subgoals, the notion of a *solution* of (E, G) and the *application type* of a form variable.

Definition 5.10 (Subgoals)
A subgoal for agents is a tuple $(\Gamma; A)$*, where* Γ *is a type context with* $dom(\Gamma) = \mathrm{fn}(A) \cap \mathrm{fv}(A)$ *and* A *is an agent, where all type annotations have been erased.*

A subgoal for forms and variables is a triple $(\Gamma; X; T)$*, where* Γ *is a type context that records all variables of* X*,* X *is a subterm generated from the syntactic domain* F *and* V*, respectively, and* T *is a type.*

Definition 5.11 (Solution of (E, G)**)**
Let σ *be a substitution. We say* σ *solves an equation* e*, written* $\sigma \models e$*, if* σ *unifies* e*. If* E *is a set of type equations, we write* $\sigma \models E$ *iff* $\sigma \models e$ *for each* $e \in E$*. If* $(\Gamma; A)$ *or* $(\Gamma; X; T)$ *are subgoals, we write* $\sigma \models (\Gamma; A)$ *and* $\sigma \models (\Gamma; X; T)$ *iff* $\Gamma\sigma \vdash A$ *and* $\Gamma\sigma \vdash X : T\sigma$*. If* G *is a set of subgoals, we write* $\sigma \models G$ *iff* $\sigma \models g$ *for each* $g \in G$*. Finally, we say* σ *solves* (E, G)*, written* $\sigma \models (E, G)$*, iff* $\sigma \models E$ *and* $\sigma \models G$*.*

Definition 5.12 (Application type of a form variable)
The application type of a form variable X *with respect to an agent* A*, written* $\mathcal{T}(X, A)$*, collects all distinct applications of* X *in* A *by assigning* X *a form type* T *that contains one entry for every distinct application using* $\langle l : M_i \rangle$ *with* M_i *fresh for occurrences of projections* X_l *and* $\langle M_j \rangle$ *with* M_j *fresh for occurrences of* X *without projection.*

$\mathcal{T}(X, A)$ is defined by the algorithm *Collect* which takes a form variable X, an agent A and a type T, and returns a new type T' for all distinct applications of X in A. For example, given an agent

$$!Update(X).cell(Y).(\overline{cell}(X) \mid \overline{X_{result}}(\langle val = X_{val} \rangle))$$

then $Collect(X;\ cell(Y).(\overline{cell}(X) \mid \overline{X_{result}}(\langle val = X_{val} \rangle)))$ yields

$$\langle\rangle\langle M_1 \rangle\langle result : M_2 \rangle\langle val : M_3 \rangle$$

as the application type for form variable X. The algorithm is shown in Appendix D.

Like Wand [121], we can now state the invariant for our algorithm.

Proposition 5.1 (Soundness and Completeness)
A solution for $(E, (\Gamma; A))$ *or* $(E; (\Gamma; X; T))$ *generates exactly the typings of* $(\Gamma; A)$ *and* $(\Gamma; X; T)$*, respectively.*

Soundness:

$(\forall\sigma)(\sigma \models (E, (\Gamma; A)) \Longrightarrow \Gamma\sigma \vdash A)$

$(\forall\sigma)(\sigma \models (E, (\Gamma; X; T)) \Longrightarrow \Gamma\sigma \vdash X : T\sigma)$

Completeness:

$\Gamma\sigma \vdash A \Longrightarrow (\exists\rho)(\rho \models (E, (\Gamma; A)) \wedge \Gamma\sigma = \Gamma\rho)$

$\Gamma\sigma \vdash X : T\sigma \Longrightarrow (\exists\rho)(\rho \models (E, (\Gamma; X; T)) \wedge \Gamma\sigma = \Gamma\rho \wedge T\sigma = T\rho)$

The invariant is clearly established by the initialization step. At termination, when $G = \emptyset$, we have

$$\begin{array}{ll} (1) & (\forall\sigma)(\sigma \models E \Longrightarrow \Gamma\sigma \vdash A) \\ & (\forall\sigma)(\sigma \models E \Longrightarrow \Gamma\sigma \vdash X : T\sigma) \\ (2) & \Gamma\sigma \vdash A \Longrightarrow (\exists\rho)(\rho \models E \wedge \Gamma\sigma = \Gamma\rho) \\ & \Gamma\sigma \vdash X : T\sigma \Longrightarrow (\exists\rho)(\rho \models E \wedge \Gamma\sigma = \Gamma\rho \wedge T\sigma = T\rho) \end{array}$$

so that the solutions of E (*Unify*(E)) give the substitutions σ such that $\Gamma\sigma \vdash A$ and $\Gamma\sigma \vdash X : T\sigma$ are provable, where σ is the most general unifier for E.

For the typed $\pi\mathcal{L}$-calculus, with typing rules summarised in Appendix B, the following actions (subgoals) are defined. We present the actions with the corresponding typing rules. The reader should note that we only need subgoals for rules involving agents, forms, channels, and constant values. A possible subtyping relation is checked using the predicates SUB, SUBFORM, and ELEMSUB.

- Case $(\Gamma; \mathbf{0})$ – (*Null*).
- Case $(\Gamma; A \mid B)$ – (*Parallel composition*).

  Generate the subgoals $(\Gamma; A)$ and $(\Gamma; B)$.
- Case $(\Gamma; !A)$ – (*Replication*).

  Generate the subgoal $(\Gamma; A)$.
- Case $(\Gamma; (\nu\ a)A)$ – (*Restriction*).

  Let M be a fresh type variable. Generate the subgoal $(\Gamma, a : \updownarrow M; A)$.
- Case $(\Gamma; V(X).A)$ – (*Input*).

  Let $T = \mathcal{T}(X; A)$ and M_1, M_2 be fresh type variables. Set $E = E \cup \{\mathrm{SUB}(T\backslash\langle M_1\rangle, \langle\rangle)\} \cup \{M_2 = \downarrow T\backslash\langle M_1\rangle\}$. Generate the subgoals $(\Gamma, X : T\backslash\langle M_1\rangle; A)$ and $(\Gamma; V; M_2)$.
- Case $(\Gamma; \overline{V}(F))$ – (*Output*).

  Let M_1, M_2, M_3, M_4 be fresh type variables. Set $E = E \cup \{\mathrm{SUB}(M_1, \langle\rangle)\} \cup \{M_2 = \uparrow M_1\} \cup \{\mathrm{SUBFORM}(M_1, M_2)\} \cup \{M_1 = \mathrm{MERGE}(M_4, \backslash\langle M_3\rangle)\}$. Generate the subgoals $(\Gamma; V; M_2)$ and $(\Gamma; F; M_4)$.

- Case $(\Gamma; c; M)$ – (*Constant*).

  Set $E = E \cup \{M = K\}$ if $\Gamma(c) = K$.

- Case $(\Gamma; x; M)$ – (*Channel*).

  Set $E = E \cup \{M = T\}$ if $\Gamma(x) = T$.

- Case $(\Gamma; \mathcal{E}; M)$ – (*Empty form*).

  Set $E = E \cup \{M = \langle\rangle\}$.

- Case $(\Gamma; X; M)$ – (*Form variable*).

  Set $E = E \cup \{M = T\} \cup \{\mathrm{SUB}(T, \langle\rangle)\}$ if $\Gamma, X : T \vdash X : T$.

- Case $(\Gamma; F\langle l{=}V\rangle; M)$ – (*Binding extension*).

  Let M_1, M_2, and M_3 be fresh type variables. Set $E = E \cup \{M = \mathrm{MERGE}(M_1, M_2)\}$ $\cup\{(M_2 = \langle l : M_3\rangle)\}$. Generate the subgoals $(\Gamma; F; M_1)$ and $(\Gamma; V; M_3)$.

- Case $(\Gamma; FX; \tau)$ – (*Polymorphic extension*).

  Let M_1, M_2 be fresh type variables. Set $E = E \cup \{M = \mathrm{POLYMERGE}(M_1, M_2)\}$ $\cup\{\mathrm{ELEMSUB}(M_2, M_1)\}$. Generate the subgoals $(\Gamma; F; M_1)$ and $(\Gamma; X; M_2)$.

- Case $(\Gamma; X_l; M)$ – (*Projection*).

  Let M_1 be a fresh type variable. Set $E = E \cup \{M = \mathrm{PROJ}(l, M_1)\}$ and generate the subgoal $(\Gamma; X; M_1)$.

Proposition 5.2 *Check*$(\Gamma; A)$ *always terminates.*

PROOF: Each action generates subgoals involving terms smaller than the original. □

Proposition 5.3 *Each action preserves the invariant of the algorithm.*

PROOF: For (1), we proceed by induction on the structure of actions (subgoals).

- Case $(\Gamma; \mathbf{0})$.

  Immediate, since $\mathbf{0}$ is consistent with any context. Therefore, we have $\Gamma\sigma \vdash \mathbf{0}$ as required.

- Case $(\Gamma; A \mid B)$.

  Then, it must be the case that $\sigma \models (\Gamma; A)$ and $\sigma \models (\Gamma; B)$. We can therefore use induction to prove $\Gamma\sigma \vdash A$ and $\Gamma\sigma \vdash B$. The result follows using rule (*Parallel composition*).

- Case $(\Gamma; !A)$.

  Then, it must be the case that $\sigma \models (\Gamma; A)$. We can therefore use induction to prove $\Gamma\sigma \vdash A$. The result follows using rule (*Replication*).

- Case $(\Gamma; (\nu\ a)A)$.

  Let M be a fresh type variable. Then, it must be the case that $\sigma \models (\Gamma, a : \updownarrow M; A)$. We can therefore use induction to prove $\Gamma\sigma, a : \updownarrow T\sigma \vdash A$. The result follows using rule (*Restriction*).

- Case $(\Gamma; V(X).A)$.

  Let $T' = \mathcal{T}(X, A)$ and M_1, M_2 be fresh type variables. Then, it must be the case that $\sigma \models \mathrm{SUB}(T'\backslash\langle M_1\rangle, \langle\rangle)$, $\sigma \models M_2 = \downarrow T'\backslash\langle M_1\rangle$, $\sigma \models (\Gamma, X : T'\backslash\langle M_1\rangle; A)$, and $\sigma \models (\Gamma; V; M_2$. We can therefore use induction to prove $\Gamma\sigma \vdash V : \downarrow T\sigma$, $\Gamma\sigma \vdash T\sigma <: \langle\rangle$, and $\Gamma\sigma, X : T\sigma \vdash A$. The result follows using rule (*Input*).

- Case $(\Gamma; \overline{V}(F))$.

  Let M_1, M_2, M_3, M_4 be fresh type variables. Then it must be the case that $\sigma \models \mathrm{SUBFORM}(M_1, M_2)$, $\sigma \models M_1 = \mathrm{MERGE}(M_4, \backslash\langle M_3\rangle)$, $\sigma \models \mathrm{SUB}(M_1, \langle\rangle)$, $\sigma \models M_2 = \uparrow M_1$, $\sigma \models (\Gamma; V; M_2)$, and $\sigma \models (\Gamma; F; M_4)$. We can therefore use induction to prove $\Gamma\sigma \vdash V : \uparrow T\sigma$, $\Gamma\sigma \vdash F : T\sigma$, and $\Gamma\sigma \vdash T\sigma <: \langle\rangle$. The result follows using rule (*Output*).

- Case $(\Gamma; c; M)$.

  Then, it must be the case that $\sigma \models M = K$. We can therefore use induction to prove $\Gamma\sigma \vdash K\sigma$. The result follows using rule (*Constant*).

- Case $(\Gamma; x; M)$.

  Then, it must be the case that $\sigma \models M = T$ with $T = \Gamma(x)$. We can therefore use induction to prove that $\Gamma\sigma, x : T\sigma \vdash \diamond$. The result follows using rule (*Channel*).

- Case $(\Gamma; \mathcal{E}; M)$.

  Then, it must be the case that $\sigma \models M = \langle\rangle$. The result is immediate, since $\langle\rangle$ is consistent with any context. Therefore, we have $\Gamma\sigma \vdash \mathcal{E}$ as required.

- Case $(\Gamma; X; M)$.

  Then, it must be the case that $\sigma \models M = T$ and $\sigma \models \text{SUB}(T, \langle\rangle)$ with $T = \Gamma(X)$. We can therefore use induction to prove $\Gamma\sigma, X : T\sigma \vdash \diamond$ and $\Gamma\sigma \vdash T\sigma <: \langle\rangle$. The result follows using rule (*Form variable*).

- Case $(\Gamma; F\langle l{=}V\rangle; M)$.

  Let M_1, M_2, M_3 be fresh type variables. Then, it must be the case that $\sigma \models M_2 = \langle l : M_3\rangle$, $\sigma \models M = \text{MERGE}(M_1, M_2)$, $\sigma \models (\Gamma; F; M_2)$, and $\sigma \models (\Gamma; V; M_3)$. We can therefore use induction to prove that $\Gamma\sigma \vdash F : T\sigma$, $\Gamma\sigma \vdash T\sigma <: \langle\rangle$, and $\Gamma\sigma \vdash V : S\sigma$. The result follows using rule (*Binding extension*).

- Case $(\Gamma; FX; M)$.

  Let M_1, M_2 be fresh type variables. Then, it must be the case that $\sigma \models M = \text{POLYMERGE}(M_1, M_2)$, $\sigma \models \text{ELEMSUB}(M_2, M_1)$, $\sigma \models (\Gamma; F; M_1)$, and $\sigma \models (\Gamma; X; M_2)$. We can therefore use induction to prove that $\Gamma\sigma \vdash T\sigma <: \langle\rangle$, $\forall l_k \in \mathcal{T}\sigma - \mathcal{L}(S\sigma)\ \Gamma\sigma S\sigma <: \langle\rangle \backslash l_k$, $\forall l_i \in \mathcal{T}\sigma \cap \mathcal{L}(S\sigma)\ \Gamma\sigma(S\sigma)_{l_i} <: (T\sigma)l_i$, $\Gamma\sigma \vdash F : T\sigma$, and $\Gamma\sigma \vdash X : S\sigma$. The result follows using rule (*Polymorphic extension*).

- Case $(\Gamma; X_l; M)$.

  Let M_1 be a fresh type variable. Then, it must be the case that $\sigma \models M = \text{PROJ}(l, M_1)$ and $\sigma \models (\Gamma; X; M_1)$. We can therefore use induction to prove that $\Gamma\sigma \vdash X : \langle l : T\sigma\rangle$. The result follows using rule (*Projection*). □

For (2), we assume that the invariant holds before the action is taken, and we need to show that it holds afterwards. Therefore, we assume that $\Gamma\sigma \vdash A$ and $\Gamma\sigma \vdash X : T\sigma$. By the induction hypothesis, we know that

$$(\exists\rho)(\rho \models (E, (\Gamma; A)) \wedge \Gamma\sigma = \Gamma\rho)$$
$$(\exists\rho)(\rho \models (E, (\Gamma; X; T)) \wedge \Gamma\sigma = \Gamma\rho \wedge T\sigma = T\rho)$$

and we need to show that

$$(\exists\rho')(\rho' \models (E', G') \wedge \Gamma\sigma = \Gamma\rho')$$
$$(\exists\rho')(\rho' \models (E, G') \wedge \Gamma\sigma = \Gamma\rho' \wedge T\sigma = T\rho')$$

where (E', G') is the state after the action step. In each case, let G' denote G after the selected goal has been deleted. Then we know that $\rho \models E$, $\rho \models G'$, and $\rho \models g$, where g is the selected subgoal. We consider each action in turn.

- Case $(\Gamma; \mathbf{0})$.

  Then $\Gamma\rho \vdash \mathbf{0}$. The result is immediate, since $\mathbf{0}$ is consistent with any context.

- Case $(\Gamma; A \mid B)$.

  Then $\Gamma\rho \vdash A \mid B$. By the typing rules, it must be true that $\Gamma\rho \vdash A$ and $\Gamma\rho \vdash B$, hence $\rho \models (\Gamma; A)$ and $\rho \models (\Gamma; B)$ as required.

- Case $(\Gamma; !A)$.

  Then $\Gamma\rho \vdash !A$. By the typing rules, it must be true that $\Gamma\rho \vdash A$, hence $\rho \models (\Gamma; A)$ as required.

- Case $(\Gamma; (\nu\ a)A)$.

  Then $\Gamma\rho \vdash (\nu\ a)A$. By the typing rules, there must be some type T such that $\Gamma\rho, a : \updownarrow T \vdash A$. So let M be a fresh type variable, and let ρ' be defined by $\rho' = \rho \circ \{M \mapsto \updownarrow T\}$. Then $\rho' \models (\Gamma, a : M; A)$ as required.

- Case $(\Gamma; V(X).A)$.

  Then $\Gamma\rho \vdash V(X).A$. By the typing rules, there must be some type T such that $\Gamma\rho \vdash V : \downarrow T$, $\Gamma\rho \vdash T <: \langle\rangle$, and $\Gamma\rho, X : T \vdash A$. So let $T' = \mathcal{T}(X, A)$ and M_1, M_2 be fresh type variables, and let $\rho' = \rho \circ \{M_2 \mapsto \downarrow T\}$. Then $\rho' \models \mathrm{SUB}(T'\backslash\langle M_1\rangle, \langle\rangle)$, $\rho' \models M_2 = \downarrow T'\backslash\langle M_1\rangle$, $\rho' \models (\Gamma; X : T'\backslash\langle M_1\rangle; A)$, and $\rho' \models (\Gamma; V; M_2)$ as required.

- Case $(\Gamma; \overline{V}(F))$.

  Then $\Gamma\rho \vdash \overline{V}(F)$. By the typing rules, there must be some type T such that $\Gamma\rho \vdash V : \uparrow T$, $\Gamma\rho \vdash T <: \langle\rangle$, and $\Gamma\rho \vdash F : T$. So let M_1, M_2, M_3, M_4 be fresh type variables, and let $\rho' = \rho \circ \{M_1 \mapsto T\} \circ \{M_2 \mapsto \uparrow T\} \circ \{M_4 \mapsto T\}$. Then $\rho' \models \mathrm{SUB}(M_1, \langle\rangle)$, $\rho' \models \mathrm{SUBFORM}(M_1, M_2)$, $\rho' \models \mathrm{MERGE}(M_4, \backslash\langle M_3\rangle)$, $\rho' \models M_2 = \uparrow M_1$, $\rho' \models (\Gamma; V; M_2)$, and $\rho' \models (\Gamma; F; M_4)$ as required.

- Case $(\Gamma; c; M)$.

  Then $\Gamma\rho \vdash c : M\rho$. By the typing rules, it must be true that $\Gamma\rho \vdash K$, hence $\rho \models M = K$. So $\rho \models (E', G')$ as required.

- Case $(\Gamma; x; M)$.

  Then $\Gamma\rho, x : M\rho \vdash x : M\rho$. By the typing rules, there must be some type T such that $\Gamma\rho, x : T \vdash \diamond$, hence $\rho \models M = T$. So $\rho \models (E', G')$ as required.

- Case $(\Gamma; \mathcal{E}; M)$.

  Then $\Gamma\rho \vdash \mathcal{E} : \langle\rangle$. The result is immediate, since $\langle\rangle$ is consistent with any context. Therefore, $\rho \models M = \langle\rangle$ and $\rho \models (E', G')$ as required.

- Case $(\Gamma; X; M)$.

  Then $\Gamma\rho, X : M\rho \vdash X : M\rho$. By the typing rules, there must be some type T such that $\Gamma\rho, X : T \vdash \diamond$ and $\Gamma\rho \vdash T <: \langle\rangle$. Then $\rho \models M = T$, $\rho \models \mathrm{SUB}(T, \langle\rangle)$, and $\rho \models (E', G')$ as required.

- Case $(\Gamma; F\langle l{=}V\rangle; M)$.

  Then $\Gamma\rho \vdash F\langle l{=}V\rangle : M\rho$. By the typing rules, there must be some types T and S such that $\Gamma\rho \vdash F : T$, $\Gamma\rho \vdash T <: \langle\rangle$, and $\Gamma\rho \vdash V : S$. So let M_1, M_2, M_3 be fresh type variables, and let $\rho' = \rho \circ \{M_1 \mapsto T\} \circ \{M_3 \mapsto S\}$. Then $\rho' \models M_2 = \langle l : M_3\rangle$, $\rho' \models M = \mathrm{MERGE}(M_1, M_2)$, $\rho' \models (\Gamma; F; M_1)$, and $\rho' \models (\Gamma; V; M_3)$ as required.

- Case $(\Gamma; FX; M)$.

  Then $\Gamma\rho \models FX : M\rho$. By the typing rules, there must be some types T and S such that $\Gamma\rho \vdash F : T$, $\Gamma\rho \vdash X : S$, $\Gamma\rho \vdash T <: \langle\rangle$, $\forall l_k \in \mathcal{L}(T) - \mathcal{L}(S)$ $\Gamma\rho S <: \langle\rangle\backslash l_k$, and $\forall l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)$ $\Gamma\rho S_{l_i} <: T_{l_i}$. So let M_1, M_2 be fresh type variables, and let $\rho' = \rho \circ \{M_1 \mapsto T\} \circ \{M_2 \mapsto S\}$. Then $\rho' \models M = \mathrm{POLYMERGE}(M_1, M_2)$, $\rho' \models \mathrm{ELEMSUB}(M_2, M_1)$, $\rho' \models (\Gamma; F; M_1)$, and $\rho' \models (\Gamma; X; M_2)$ as required.

- Case $(\Gamma; X_l; M)$.

  Then $\Gamma\rho \vdash X_l : M\rho$. By the typing rules, there must be some type T such that $\Gamma\rho \vdash X : \langle l : T\rangle$. So let M_1 be a fresh type variable, and $\rho' = \rho \circ \{M \mapsto T\}$. Then $\rho' \models M = \mathrm{PROJ}(l, M_1)$ and $\rho' \models (\Gamma; X; M_1)$ as required. □

Chapter 6

A composition system

As shown in Section 2.6, a general-purpose composition language should support the following features:

- **Active Objects:** Objects are computational entities that provide services based on an encapsulated state. Objects may be active (concurrent), distributed, mobile, and may live in different environments. In any case, objects can be viewed as a kind of server, or *process*. The process view of objects provides a way to formalize the notion of objects. A composition language must be able to instantiate and communicate with active objects.

- **Components:** Components are abstractions over the object space [74]. Components may be fine-grained, when used to build individual objects, or coarse-grained, when used to build compositions of objects. They may be also runtime entities, but, more generally, components must be constructed (composed) and instantiated before they are part in an application.

- **Glue:** Glue mechanisms and operators (connectors) define how associated components interact with each other [107]. A composition language must support the specification of new kinds of connectors.

- **Object Models:** A composition language must be able to bridge the gap between different object models. Objects and components that cannot be separated from their individual runtime environments must still be able to communicate. Connectors must achieve the mappings between these object models. Using a formal semantics allow us to reason about properties of the model more easily.

- **Reflection:** Glue is often realized by intercepting messages between objects, and performing some (reflective) transformations on these messages [10, 32]. Reflection is also important for exercising run-time control on configurations. Metaobjects are active objects that control the creation, instantiation, and composition

of other objects [52], and can be used to realize various forms of reflective behaviour.

- **High-level syntax:** The specification of an application as a composition of components must be highly readable and compact. It is therefore important to be able to assign a high-level syntax when defining components (as is possible in languages like Smalltalk).

Furthermore, we have identified the key properties of software components and component frameworks:

- A software component is an element of a component framework.
- A software component is a "static abstraction with plugs".
- A component framework is a collection of software components with a software architecture that determines the interfaces that components may have and the rules governing their composition.

In the previous chapters, we have developed the $\pi\mathcal{L}$-calculus and a corresponding first-order type system as a formal basis for software composition. In this chapter we present a Java-based composition system and the composition language Piccola. Both, the system and the language are based on the $\pi\mathcal{L}$-calculus, i.e., components and their composition have a process semantics defined by the $\pi\mathcal{L}$-calculus.

6.1 The architecture

An overview of the composition system Piccola is given in Figure 6.1. On top of the system, we have the composition language Piccola. With Piccola we can define *composition scripts* which are again components. Complied scripts are stored in the Piccola *component library*. We have two kinds of information in the component library: *binaries* and *component interface definitions*. The component binaries are in fact a kind of object files that need to be bound to an executable agent image by a component linker. The interface definitions are used by the Piccola compiler to perform static checks when a corresponding component is used within a script. Interface specifications are very similar to type libraries used in COM [60].

The kernel of the composition system is built using Java [7]. More precisely, on top of the Java virtual machine (JVM) runs a virtual $\pi\mathcal{L}$-machine [2] that implements the $\pi\mathcal{L}$-calculus. In the $\pi\mathcal{L}$-machine each entity of the calculus is implemented by a corresponding Java-class, i.e., we have classes for agents, channels, and forms.

The base class for agents is `Agent` that provides the methods `run` to start an agent and `iter` to perform one evaluation step. The method `run` internally creates a new

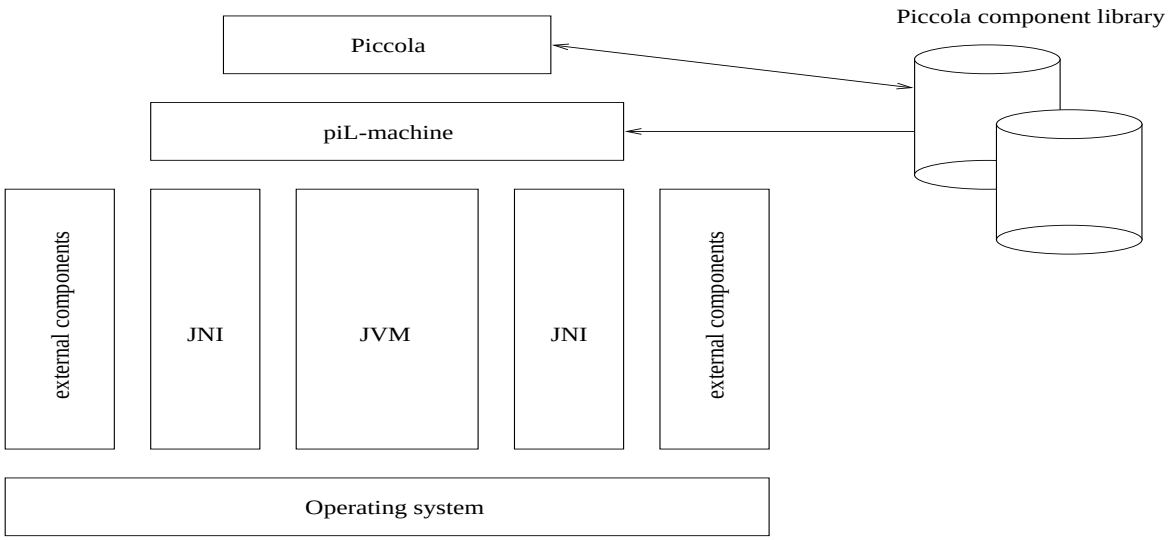

Figure 6.1: The architecture of the PICCOLA system.

Java-thread and associates the agent with the thread. The method `iter` is responsible for the thread control. An agent thread is executed until the next continuation returned by `iter` is `null` – the inactive agent. On receiving `null`, the current thread is terminated. The different forms of agents are implemented by corresponding subclasses like `ParAgent` for the parallel composition of two agents.

Channels in the $\pi\mathcal{L}$-machine are represented by the classes that implement the interface `Channel`. The interface `Channel` defines two methods: `get` and `put` which are used to receive or the send a form along a channel, respectively.

In general, the programmer does not get directly in contact with channel objects. However, there is one exception. The $\pi\mathcal{L}$-machines provides the class `ExtChannel` that implements a gateway to the Java system. Forms sent along such channels are forwarded to an Java object that implements a so-called *external service* (e.g., `print`).

Finally, forms are implemented using the interface `Form` which defines the methods: `project` and `extend`. The method `project` is used to perform a projection for a given label, while method `extend` is responsible for creating a new forms either by binding extension or polymorphic extension.

If we want to use objects or components that have not been written in PICCOLA, we have to define an external service using class `ExtChannel`. However, the external service must not necessarily be implemented solely in Java. Using the *Java Native Interface* (JNI), we can integrate almost any object or component model into the composition system using the wrapping technology of JNI. For example, we can define a JNI-wrapper that allows us to call system commands directly from a PICCOLA script. The corresponding input and output of the command are encoded into a form that has

Declarations	::=	*Declaration* [*Declarations*]
Declaration	::=	'**new**' *NameList* '**run**' *Agent*
Agent	::=	*PrimaryAgent* ['|' *Agent*]
PrimaryAgent	::=	'**null**' *Location* '!' *Form* *Location* '?' '(' *Variable* ')' '**do**' *Agent* *Location* '?∗' '(' *Variable* ')' '**do**' *Agent* '**let**' *Declarations* '**in**' *Agent* '**end**' '**if**' *BoolExpression* '**then**' *Agent* ['**else**' *Agent*] '**end**' '(' *Agent* ')'
NameList	::=	*Name* [, *NameList*]
BoolExpression	::=	*Value Built-in-BoolOperator Value*
Location	::=	*Name* *Variable* '.' *Label*
Form	::=	'<' [*FormElementList*] '>' *Variable*
FormElementList	::=	*FormElement* [',' *FormElementList*]
FormElement	::=	*Variable* *Label* '=' *Value*
Value	::=	*Location* *String* *Number*

Table 6.1: The core of Piccola.

the label `stdin`, `stdout`, and `stderr`.

Using Java as implementation language, we have an additional advantage. Due to the platform independency of the Java byte code, the composition system can run on an arbitrary architecture (if an implementation for this architecture exists). Furthermore, using Java's RMI facility, it will be even possible to execute remote services. This is, however, part of future development.

6.2 Towards a composition language

In this section we develop a first version of the composition language PICCOLA. In order to define PICCOLA, we use the scheme that has been successfully used for the definition of PICT [89], i.e., we define a language core and extend it step by step with higher-level syntactic forms that are translated into the core language.

6.2.1 The core language

The core is presented in Table 6.1. For describing the syntax, we rely on a meta notation similar to the Backus-Naur Form that is commonly employed for language definitions. Keywords and symbolic constants appear inside quotes. Optional expressions are enclosed in square brackets.

The reader should note that a parallel composition of agents extends to the right as far as possible. Therefore, if an input-prefixed agent is built using parallel composition of subagents, these subagents have to be written in parentheses (e.g., the agent (`a?(X) do b! <Y>`) | `c! <Z>` is different from the agent `a?(X) do b! <Y> | c! <Z>`).

In contrast to the $\pi\mathcal{L}$-calculus, the syntax for forms has been simplified. A form `X` extended with a sequence of bindings `<label1 = value1><label2 = value2>` is now written as a list of form elements: `<X, label1 = value1, label2 = value2>`.

In the following, we iteratively extend the syntax by defining higher-level abstractions. These higher-level abstractions come with a set of small examples that will motivate the newly introduced concepts or aspects of the higher-level language. Throughout this section, we show only informally the enhancements. The complete syntax definition of PICCOLA is given in Appendix E.

6.2.2 Procedures

Assume we have implemented a simple person database service that is located at a global channel `lookup`. In order to query information about a person, we have to send a form containing the labels `name` and `result`. The label `name` binds a string denoting the name of the person while the label `result` maps a channel along which the query result is returned. In the following example, we query information about a person "Smith" and display the received information on the screen using a built-in display agent located at channel `print`.

```
new result                                          // reply channel
run lookup! <name="Smith",result=result>            // invoke query
    | result?(Info) do                              // wait for information
            print!Info                              // print information
```

This definition, however, can immediately be simplified, because form variables can host arbitrary forms. In fact, instead of using a newly created reply channel `result`,

we can directly bind the channel `print` to label `result`, such that the query result is directly passed to the display agent. The simplified definition is shown in the following:

```
run lookup! <name="Smith",reply=print>
```

Now, we would like to provide this behaviour as a single service that is parameterized with the person string to be queried. Therefore, we define a replicated input agent that listens at channel `printPerson` waiting for a form containing at least a label `name` and displays the corresponding information on the screen.

```
new printPerson                                    // service channel
run printPerson?*(ArgForm) do                      // wait for a form
     lookup! <ArgForm,result=print>
```

In order to invoke this service, we send a form containing the label `name` to the channel `printPerson`:

```
printPerson! <name="Smith">
```

In fact, the above service definition is a parameterized agent abstraction. The structure of the definition is so common that we provide a new element for the syntactic domain *Declaration*; we extend the language with a *procedure declaration*:

$$\textit{Declaration} \quad ::= \quad \text{'}\textbf{procedure}\text{'}\ \textit{Name}\ \text{'('}\ [\ \textit{Variable}\ ]\ \text{')'}\ \text{'}\textbf{do}\text{'}\ \textit{Agent}$$

Having the procedure declaration available, we also want to have a convenient way to invoke procedures. Therefore, we add an *application*[1]The reader should note that we allow arbitrary locations to denote procedure names. For example, an agent can get access to a procedure by receiving a form that contains a binding which maps to the procedure's name. as additional syntactic form to primary agents:

$$\textit{Application} \quad ::= \quad \textit{Location}\ \text{'('}\ \textit{Form}\ \text{')'}$$

Now, assuming that the service `lookup` was also defined as procedure, the above `printPerson` service can be rewritten using the new syntax:

```
procedure printPerson(ArgForm) do
     lookup(<ArgForm,result=print>)
```

6.2.3 Value declaration

So far, new form variables can only be introduced by input-prefixed agents or procedures. However, it is often convenient to make some variables globally accessible, such that they appear "free" somewhere in the program text and provide a value throughout the rest of that program. To define such variables and to assign them values, we introduce a *value declaration* which is defined as follows:

$$\textit{Declaration} \quad ::= \quad \text{'}\textbf{value}\text{'}\ \textit{Variable}\ \text{'='}\ \textit{Form}$$

[1](

6.2.4 Complex forms

All form expressions that we have encountered so far have been built up in a simple way, using just variables and bindings from labels to core values. If we, however, want to define updatable data structures, we need a packing technique that allows us to define data and operations over them in one syntactic construct. Therefore, we extend the syntax of forms and allow local declarations and call these constructs *complex forms*:

$$\textit{Form} \quad ::= \quad \text{'}\textbf{let}\text{'}\ \textit{Declaration}\ \text{'}\textbf{in}\text{'}\ \textit{Form}\ \text{'}\textbf{end}\text{'}$$

For example, this syntactic construct together with the value declaration allows us to define a storage cell as follows:

```
value AStorageCell =
      let
         new cell
         run cell!<>
         procedure Read(Args) do cell?(Val) do (Args.result!Val | cell!Val)
         procedure Update(Val) do cell?(OldVal) do cell!Val
      in
         <Read=Read,Update=Update>
      end
```

The contents of `AStorageCell` can be read by `Read` and updated by `Update`. This cell can store arbitrary form values. Later, we will define a storage cell generator that implements `Read` and `Update` as functions which will allow us to apply the operations synchronously.

The reader should note that if a complex form appears in an output, we only send the corresponding form value and keep the declarations private. For example, if we have the following agent

```
AChannel!let new x in <val=x> end
```

then this agent is translated into the core construct

```
let new x in AChannel!<val=x> end
```

6.2.5 Nested forms

Up to now, forms are flat values, i.e., there is no support to add an additional structure to forms. However, it is often necessary to keep things separated. For example, if a service expects both a channel and a value, we have to send both in one form which is typically written like

```
<channel=cname,AFormValue>
```

Unfortunately, this definition merges the channel and the form value, such that the original structure information is lost. Therefore, we introduce so-called *nested forms*, i.e., we extend *Value* by forms

$$\textit{Value} \quad ::= \quad \textit{Form}$$

whereby a form containing such a value is translated into a complex form.

The above example can now be rewritten as

```
<channel=cname,val=AFormValue>
```

This definition preserves the original structure and has to be read as

```
let
   new r
   run r?*(X) do X.result!AFormValue
in
   <channel=cname,val=r>
end
```

6.2.6 Functions

Each time we have applied a form in an output agent or a call expression, we have used "static" forms. To get a greater flexibility and even a more compact specification, we add abstractions for "dynamic" forms. The term "dynamic" means that forms are either generated by an agents that act as functions or that have themselves dynamic elements. For functions, we extend *Declarations* as follows:

$$\textit{Declaration} \quad ::= \quad \text{'}\textbf{function}\text{'}\ \textit{Name}\ \text{'('}\ [\ \textit{Variable}\ ]\ \text{')'}\ \text{'='}\ \textit{Form}$$

In order to call a function, we add also the syntactic domain *Application* to forms.

$$\textit{Form} \quad ::= \quad \textit{Application}$$

However, in contrast to the procedures, the translation of functions uses `result` as default label to return the function result. Therefore, a function call

```
AFun( AForm )
```

is transformed into

```
AFun( <AForm,result=AResultChannel> )
```

The result of a function may be available before the complete function body has been processed. To stress this fact, we extend the syntax domain *Form* with a *return expression*:

$$\textit{Form} \quad ::= \quad \text{'}\textbf{return}\text{'}\ \textit{Form}$$

This expression should only be used within a function declaration, because the return expression is translated to an output process that sends the form expression along a channel mapped by label `result`. For example, if the formal parameter of a function is named `Args` a return expression

```
return AForm
```

is translated into

```
Args.result!AForm
```

The reader should note that if one specifies a return expression within a procedure this expression will evaluate to the `null` agent, because the necessary label `result` is not defined.

6.2.7 Active forms

Active forms are forms that contain active elements, i.e., they have functions or procedure specifications or function calls in place of bindings. This concept allows us to define form expressions that can act as objects. Therefore, we add the following form elements:

FormElement ::= '**procedure**' *Name* '(' [*Variable*] ')' '**do**' *Agent*
'**function**' *Name* '(' [*Variable*] ')' '=' *Form*
Application

The functions and procedures are translated into private agents. The names of the functions and procedures are also used as label names.

An application, however, is transformed into a value declaration, i.e., the active form is translated into a complex form that contains a value declaration and the value replaces the application within the form. For example,

```
<l=AFun()>
```

is transformed into

```
let value v=AFun() in <l=v> end
```

However, we allow not only form elements to be active, but also forms. Therefore, we extend forms with *conditionals*.

Form ::= '**if**' *BoolExpression* '**then**' *Form* ['**else**' *Form*] '**end**'

Conditional forms are transformed into a function that implements the conditional form and a function call that triggers the evaluation of the conditional. The function call itself replaces the conditional form (see Section 6.3 for further details).

6.2.8 Sequencing

In fact, the form <> can be considered as a *continuation signal*, i.e., it carries no information but tells the calling agent that its request has been satisfied. Using this signal, we can synchronize parallel running agents. Therefore, we provide a convenient syntax to specify "invoke operation, wait for a signal as a result, and continue":

$$\begin{aligned} \textit{PrimaryAgent} &::= \textit{Form}\ \text{';'}\ \textit{PrimaryAgent} \\ \textit{Form} &::= \textit{Form}\ \text{';'}\ \textit{Form} \end{aligned}$$

In both, the lefthand-side form is evaluated before the righthand-side agent or form becomes active. This means, however, that the value of the lefthand-side form is lost; we only interested in the fact that this form has been evaluated, so that it is now safe to proceed. For example, a form sequence $\mathbf{f_1}\,;\mathbf{f_2}$ has to be read as:

```
let value _ = f1 in f2 end
```

In some cases, however, one needs the value that has been produced within a form sequence. In order to specify this request, we can use the *return expression*. For example, a form sequence in the function

```
function AFun(X) = f1;return f2;f3
```

is transformed into a complex form:

```
function AFun(X)
   let
      value _ = f1
      value v = f2
      run let value _ = f3 in null end
   in
      v
   end
```

The reader should note that the function `AFun` may return the value `v` to the caller before the evaluation of form $\mathbf{f_3}$ has been completed. In fact, $\mathbf{f_3}$ does not belong to the result value, but it represents some cleanup or post operation that guarantees some invariants to be hold for further calls of `AFun`.

6.2.9 External services

In order to incorporate components that have not been developed in PICCOLA, we provide an *external declaration*:

$$\textit{Declaration} ::= \text{'}\mathbf{extern}\text{'}\ \textit{String}\ \textit{Name}$$

In this declaration *String* denotes a Java class that provides the interface implementation to the external service while *Name* can be an arbitrarily chosen name that maps the Java class to a channel name. For example, a print service located at `print` is declared as follows:

```
extern "Piccola.builtin.print" print
```

The PICCOLA system provides several Java classes to map external components. However, a detailed description of them is beyond the scope of this paper.

6.2.10 Composition scripts

Finally, we add the facility to define separate modules or composition scripts. A script is itself a component. i.e, it can be composed with other composition scripts.

Composition scripts can be separately compiled. The result is stored in a composition library. Composition scripts can load other scripts that have been previously compiled. In fact, circular dependencies of composition scripts are not allowed.

A composition script is defined as follows:

Script ::= '**module**' *ModuleName* [*Imports*] *Declarations* [*Main*]

Imports ::= '**load**' *NameList* ';'

Main ::= '**main**' *Agent*

Inspired by Python [53], the main declaration specifies the agent that has to be started in the PICCOLA environment when the script is executed at the top level. Main declarations of imported scripts are ignored.

The reader should note that compositon scripts defined using the above syntax allow only *static composition*. It is not possible to change such a system at runtime.

6.2.11 An example

Agents can assume different roles. In general, agents either act as servers or as clients. Therefore, a server, like in Linda [25, 82], can be thought of as a *blackboard* where client agents can write, read, and remove data. If the data is not present on the blackboard, reading and removing suspend the the agent until they are available.

We can implement a blackboard in PICCOLA as follows:

```
module Blackboard

function newBlackboard() =
  let
    new chn
  in
    <
      procedure write( Args ) do (chn!<Args.val()> | Args.result!<>),
      procedure remove( Args ) do chn?(X) do Args.result!X,
      procedure read( Args ) do chn?(X) do (Args.result!X | chn!X)
    >
  end
```

Note, although the predicates are implemented as procedures, they act as functions. The reason is that a nondestructive read as needed in `read` is not part of the calculus, but this can be modelled by an input-prefixed agent that reads a form from a given channel and sends it back immediately on the same channel.

Now, using the blackboard module the storage cell can be written as follows:

```
module StorageCell

load Blackboard;

value StorageCell =
  let
    value Blackboard = newBlackboard()
  in
    Blackboard.write( <val=<val= 1>> );
    <
      function Read() = Blackboard.read(),
      function Update(Val) = Blackboard.remove(); Blackboard.write( Val )
    >
  end

main StorageCell.Update( <val=<val="string value">> ); print!StorageCell.Read()
```

6.3 Interpretation of higher-level constructs

In the following we present an interpretation of the higher-level syntax constructs in the core language. We use the meta variables $\mathbf{a_1}, ..., \mathbf{a_n}$ to range over agents, $\mathbf{f_1}, ..., \mathbf{f_n}$ to range over forms, and $\mathbf{v_1}, ..., \mathbf{v_n}$ to range over values.

6.3.1 Procedures and procedure calls

A procedure declaration is translated into the core language as follows:

```
procedure AProc( AVar ) do a   ⟹   new AProc
                                   run AProc?*(AVar) do a
```

A procedure call is translated simply into a output agent:

```
AProc( f )   ⟹   AProc!f
```

6.3.2 Values declarations

A **value** declaration can be used to assign an arbitrary form value to a name.

```
let value v = f in a end   ⟹   let new r in r!f | r?v do a end
```

6.3.3 Functions and function calls

A function declaration is translated into the core language as follows:

```
function AFun( AVar ) = f   ⟹   new AFun
                                run AFun?*(AVar) do AVar.result!f
```

A function call is translated simply into a output agent. However, we need to provide a result binding that is be used for the function result. The result channel is denoted by `ResultChannel`.

```
AFun( f )   ⟹   AFun!<f,result=ResultChannel>
```

6.3.4 Complex forms

The private declarations of a complex form are added to the private declarations of the agent in which the complex form appears. For example, an output agent that emits a complex form is translated as follows:

```
AChannel!let new x in <l=x> end   ⟹   let new x in AChannel!<l=x> end
```

6.3.5 Nested forms

A nested form is translated as follows:

$$<l_1=<f_1>,\ldots,l_n=<f_n>> \implies$$

```
let
     new r1,...,rn
     run r1?*(X) do X.result!f1
     ⋮
     run rn?*(X) do X.result!fn
in
     <l1=r1,...,ln=rn>
end
```

6.3.6 Active forms

Active forms are translated into complex forms:

```
    <function AFun(X) = f>   ⟹   let function AFun(X) = f in <AFun=AFun> end

<procedure AProc(X) do a>   ⟹   let procedure AProc(X) do a in <AProc=AProc> end

              <l=AFun(f)>   ⟹   let value v = AFun(f) in <l=v> end

           <AFun(f),l=x>   ⟹   let value v = AFun(f) in <v,l=x> end
```

```
                                        let
                                             new chn
                                             run chn?(Args) do
                                                    if Args.left = Args.right
if v1 = v2 then f1 else f2 end   ⟹                  then Args.result!f1
                                                    else Args.result!f2 end
                                        in
                                             <chn( <left=v1,right=v2> )>
                                        end
```

6.3.7 Sequencing

Sequence expressions are transformed as follows:

```
  f;a   ⟹  let value _ = f in a end
   f;   ⟹  let value _ = f in null end
f1;f2   ⟹  let value _ = f1 in f2 end
```

6.4 Results and shortcomings

We have presented a first design of the (untyped) composition language PICCOLA, based on the $\pi\mathcal{L}$-calculus. This development is driven by a set of key requirements for a composition language [76]. Furthermore, PICCOLA provides support such that (i) an application may run on a variety of hardware and software platforms (the runtime system in built in Java), and (ii) open applications may be inherently concurrent and distributed (PICCOLA's formal semantics is based on a process calculus).

Although the set of basic language primitives of PICCOLA is relatively small, component scripts already facilitate the specification and modeling of generic abstractions for adaptation (gluing) and composition of software components. For example, we can define the abstraction *future* that is used in concurrent object-based programming allowing a client to continue execution after invocation of a service. In fact, futures are *Proxies* that mimic the "wait-by-necessity" construct found in concurrent languages. With normal invocation the client blocks until the result is delivered along the reply channel. With futures, the client only blocks if the result is needed before it is available.

In PICCOLA, we can implement a future as follows. A future wraps a service, returning a new form with a function `val` that provides access to the wrapped service. The function `val` maintains a private blackboard `slot` that is used to store the result of original service. Now, if the wrapped service is invoked we return immediately a function that returns the service value stored in `slot`. In parallel, we start an agent (`slot.write( <val = Service.val( X )> )`) that invokes the original service and writes the result to `slot`. If the user tries to access the service result the call `slot.read()` will block until `slot` has been filled.

```
function future( Service ) =
   <
      function val( X ) =
         let
            value slot = newBlackBoard()
         in
            return <function val () = slot.read()>;
            slot.write( <val = Service.val( X )> )
         end
      >
```

Assume a service is denoted by form `S`. We can now use the future abstraction as follows:

```
value r = future( <val = S> ).val( X )
```
– r is a future containing the result of S(X)

Using `r.val()` we get the value of the future.

Similarly, it is easily possible to implement *generic synchronization policies* [55, 105], *mixins* [15, 114]. Moreover, the object encodings presented in Chapter 3 can easily be modelled in PICCOLA by replacing records with forms.

For the moment PICCOLA is untyped. However, most component approaches (e.g., COM [98], CORBA [79], or Darwin [54]) equip partly or fully the interface specifications with type annotations. One argument for this decision is that only fully and explicitly typed interfaces can benefit from type checking. Furthermore, an independent development of both the client and the provider side may be more or less impossible without appropriate type information (e.g., the current version of PICCOLA already records which names have been used to denote forms). Therefore, a next extension of PICCOLA will support type annotations based on the type system presented in Chapter 5.

Chapter 7

Conclusions and future work

We have presented the $\pi\mathcal{L}$-calculus, an offspring of the asynchronous π-calculus, and a first design of the composition language PICCOLA, based on the $\pi\mathcal{L}$-calculus. The development of both is driven by a set of key requirements for a composition language shown in Section 2.6. Moreover, the composition language PICCOLA provides support such that (i) an application may run on a variety of hardware and software platforms (the runtime system in built in Java), and (ii) open applications may be inherently concurrent and distributed (PICCOLA's formal semantics is based on a process calculus).

The $\pi\mathcal{L}$-calculus, and hence PICCOLA, facilitates the specification and modeling of both (concurrent) compositional abstractions and generic glue code for adaptation and composition of software components. Although our examples are neither exhaustive nor canonical, we think they represent to some degree the essence component-oriented software development.

Ultimately we are targeting the development of open, hence distributed systems [78]. Given the ad hoc way in which the development of open systems is supported in existing languages, we have identified the need for composing software from predefined, plug-compatible software components. The overall goal of our work, and hence the development of PICCOLA, is the development of a formal model for software composition, integrating a black-box framework for modelling objects and components, and an executable composition language for specifying components and applications as compositions of software components.

We are planning a further development of the $\pi\mathcal{L}$-calculus to address various other practical problems. For example: Should labels be first-class values? Generic glue code may need to learn about new labels representing extended interfaces of components. Do first-class forms suffice to model a general reflective behaviour? Glue code is often reflective in nature. Is it enough to reflect over messages or do we need more?

In the field of concurrent and distributed systems, various process calculi have recently been proposed [20, 34, 119] that incorporate other aspects of distributed com-

putation, such as communication failure, distributed scopes, and security. For the moment, we focus on composed systems within one administrative domain and do not take into account other distribution aspects. But a proper solution that addresses these aspects will play a crucial role when we want to model composition between distributed components.

To our knowledge, it is the first time that asymmetric record concatenation (polymorphic form extension) has been fully incorporated in a programming language. Moreover, it is possible to do automatic type reconstruction starting form a totally untyped program. Polymorphic form extension becomes a very useful feature when modelling higher-level compositional abstractions like classes and class inheritance. In fact, with polymorphic form extension we expect to get a well-defined behaviour of multiple inheritance. When combining two or more features then, by the definition of the calculus, it is always clear which services are available after applying polymorphic form extension. Furthermore, polymorphic extension supports the view that subclassing is subtyping like in C++. However, we are not restricted to that view. We simply get this view for free while still being free to chose another.

The current component model of Piccola supports only *static composition*, i.e., all elements of a composed system must be known (available) at compile- and link-time. This model does therefore not support a replacement of components at runtime. However, we already have a language feature that is prepared to support a *dynamic loading* of components.

A composition script can load other scripts using the declaration **load**. In order the support dynamic loading of components, we can change the semantics of **load**, so that it behaves as a built-in service. As result of loading a service dynamically at runtime the actual system is updated by the newly loaded component. By update, we mean that the load command supports both the replacement of an existing component or the extension of the system with a new component.

A dynamic loading of components is very similar to the *QueryInterface* method of the *IUnknown* interface of COM [60, 98]. This, however, requires a dynamic component model, i.e., components are itself agents that provide a query interface for component services. The composition system will then register all loaded components in a dictionary and provides the application programmer a handle that gives access to the component. This handle can then be used to actually get access to a component service. Due to the fact that the composition system maintains all components and provides only handles for components to the application programmer, a dynamic loading or replacement of components can be done transparently.

The actual type system for the $\pi\mathcal{L}$-calculus does not support parametric polymorphism, i.e., we cannot define agents that operate uniformly on arbitrary values. For example, if we have a generic implementation of a service `Add` that, when invoked with two integer arguments, returns the sum of the integer, or, when invoked with string arguments, returns the concatenation of the strings, then we need to define two $\pi\mathcal{L}$-

agent. In this case, however, we know that the different types will not break the system. Therefore, we may add a type paramenter to the agent `Add` that is instantiated with the actual types of the input argument. For example, instead of defining two agents

$$Add : \downarrow \langle l : Int \rangle \langle r : Int \rangle \langle result : \uparrow \langle val : Int \rangle \rangle$$

and

$$Concat : \downarrow \langle l : String \rangle \langle r : String \rangle \langle result : \uparrow \langle val : String \rangle \rangle$$

we may specify simply

$$Add\ [: T :] : \downarrow \langle l : T \rangle \langle r : T \rangle \langle result : \uparrow \langle val : T \rangle \rangle$$

where T is a type parameter. However, there is in general no decision procedure for subtyping for such a type system, which means that such a type systems are undecidable [22, 88]. The simplest solution at the moment consists in requiring equal universal quantifier bounds.

Appendix A

Pict

We now present a rigorous definition of the syntax and semantics of core elements of PICT. This should help the reader to understand the subsequent encodings of our component object model. The full definition of PICT can be found in [89].

A.1 Simple processes

The simplest process that can be written in PICT is **skip**. This process has no observable bahaviour and is equivalent with the inactive process **0** of the π-calculus. To make this process into a PICT program we prefix it with the keywork **run**:

run skip

Since the process **skip** does nothing, its execution immediately finishes. The keyword **run** can be used to prefix abitrary processes and therefore force their execution.

RunDeclaration ::= **run** *Process*

A.2 Channels and types

Besides processes, the most primitive entities in the π-calculus are names that represent communication channels. In fact, each channel is a port over which one process may communicate with another. Every channel must be created before it can be used.

ChannelDeclaration ::= **new** *Name* [**:** *ChannelType*]

The declaration **new x :** ^**T** creates a fresh channel, different from any other channel in the system, and makes the name **x** refer to this channel within the scope of its declaration. Values sent and received along **x** will have the type **T**.

PICT defines the following basic types: **Bool**, **Char**, **Int**, and **String**. Additionally, we have the following composite types: *channel types*, *tuple types*, *record types*, and *recursive types*.

CompositeTypes	::=	^*Type*	Input/output channel
		!*Type*	Output channel
		?*Type*	Input channel
		[$Type_1$,..., $Type_n$]	Tuple type
		Record [l_1 :$Type_1$, ..., l_n :$Type_n$] **end**	Record type
		Rec(*Name*) *Type*	Recursive type

Channel types are enriched by distinguishing between input and output capabilities of channels. This is driven by the observation that in practice, it is relatively rare that a channel is used both for input and output in the same region of the program. This fact is captured by two refinements of the channel type ^**T**: a type **!T** giving only the capability to write values of type **T** and, symmetrically, a type **?T** giving only the capability to read values of type **T**.

Recursive types allow one to build and manipulate recursive data structures like lists and trees, or allow one to define *binary methods* [18]. For example, suppose we want to write a program in which a channel **x** is used to send itself. A valid type for **x** would be something like:

$$\mathbf{x}: \hat{}(\hat{}(\hat{}(\hat{}(\hat{}(\hat{}(\ldots))))))$$

Since **x** is a channel, the type of **x** must be prefixed by ^. But the type of the values carried along **x** is the same as the type of **x** itself. Therefore, the initial ^ has to be applied to some type beginning with another ^, which has to be applied to a type beginning with a ^, and so on. A finite abbreviation for this inifinitely long type expression is

$$\mathbf{Rec(X)}\ \hat{}\mathbf{X}$$

which has to read as a type of the form ^**X**, where **X** stands for the type itself. However, a recursive type cannot be used in positions, where a channel type is required. In order to use recursive types, PICT provides both explicit unfolding and folding that transforms a value with a recursive type into one with a nonrecursive type and one with a recursive type, respectively.

A.3 Values

The entities that can be communicated along channels are called *values*. They include channels, tuples, and records.

Value	::=	*Name*	*variable*
		$[Value, ..., Value]$	*tuple*
		record end	*empty record*
		Value **with** *Label* = *Value* **end**	*record extension*

As in the π-calculus, there are no channel constants, only variables ranging over channels. Record values are constructed by extending the empty record with a new single field. To define a multi-field record we use the usual form **record** $l_1 = v_1, ..., l_n = v_n$ **end** as an abbreviation for **record end with** $l_1 = v_1$ **end**...**with** $l_n = v_n$ **end**.

If **rval** is a record that contains already a field with the label **l**, then the record extension **rval with l = newval** creates a copy of **rval** where the field labelled with **l** is updated with the value **newval**. If label **l** is not part of **rval**, then the record extension creates a copy of **rval** extended with a new field labelled with **l**.

A.4 Processes

The basic forms of processes are input prefixes, output atoms, parallel composition, and process prefixed by local declarations[1].

Process	::=	*Value* **!** *Value*	Output atom
		Value **?** *Pat* > *Process*	Input prefix
		Value **?*** *Pat* > *Process*	Replicated input
		Process | *Process*	Parallel composition
		let *Declaration* **in** *Process*	Declaration

The general form of a sender process is **x!v**, where **x** is a channel and **v** is a value. Symmetrically, a receiver process has the form **x?p > e**, where **x** is a channel, **p** is a pattern. For example, given the following two processes

x!3 | (x?p > printi!p)

If we prefix them with **run**, then the sender process **x!3** will send the integer value **3** along channel **x**. The receiver process will match received value against pattern **p** to yield a set of bindings for the variables in **p**, i.e. matches value **3** with variable pattern **p**. The receiver process will evolve into the process

printi!3

This sender process will in turn communicate with the PICT environment, where the channel **printi** is defined. Actually, the complete PICT program is

```
new x
run x!3 | ( x?p > printi!p)
```

[1] At this stage *Declaration* is a list of **new** and **run** declarations.

Pattern in input prefixes are defined as follows:

Pat	::=	*Name*	Variable pattern
		[Pat_1, ..., Pat_n]	Tuple pattern
		record $Label_1$ = Pat_1, ..., $Label_n$ = Pat_n **end**	Record pattern
		_	Wildcard pattern

The substitution of pattern is a finite map from variables to values. A substitution mapping of the variable **x** to the value **v** is written $\{\mathbf{x} \mapsto \mathbf{v}\}$. The empty substitution is written $\{\}$. If σ_1 and σ_2 are substitutions with disjoint domains, then $\sigma_1 \cup \sigma_2$ is a substitution that combines the effects of σ_1 and σ_2.

When a value **v** is sucessfuly matched by a pattern **p**, the result is a substitution *match*(**p**, **v**), defined as follows:

$$
\begin{array}{lcl}
\mathit{match}(\mathrm{x},\ \mathrm{v}) & = & \{\mathrm{x} \mapsto \mathrm{v}\} \\
\mathit{match}([\mathrm{p}_1,...,\mathrm{p}_n],[\mathrm{v}_1,...\mathrm{v}_n]) & = & \mathit{match}(\mathrm{p}_1,\ \mathrm{v}_1) \cup ... \cup \mathit{match}(\mathrm{p}_n,\ \mathrm{v}_n) \\
\mathit{match}(\_,\ \mathrm{v}) & = & \{\} \\
\mathit{match}(\textbf{record end},\ \textbf{record end}) & = & \{\} \\
\mathit{match}(\textbf{record } \mathrm{j}_1{=}\mathrm{p}_1,...,\mathrm{j}_n{=}\mathrm{p}_n,\mathrm{l}{=}\mathrm{p} \textbf{ end},\ \mathrm{v}_0 \textbf{ with } \mathrm{l}{=}\mathrm{v} \textbf{ end}) & & \\
\qquad = \mathit{match}(\mathrm{p},\ \mathrm{v}) \cup \mathit{match}(\textbf{record } \mathrm{j}_1{=}\mathrm{p}_1,...,\mathrm{j}_n{=}\mathrm{p}_n \textbf{ end},\ \mathrm{v}_0) & & \\
\mathit{match}(\textbf{record } \mathrm{j}_1{=}\mathrm{p}_1,...,\mathrm{j}_n{=}\mathrm{p}_n \textbf{ end},\ \mathrm{v}_0 \textbf{ with } \mathrm{l}{=}\mathrm{v} \textbf{ end}) & & \\
\qquad = \mathit{match}(\textbf{record } \mathrm{j}_1{=}\mathrm{p}_1,...,\mathrm{j}_n{=}\mathrm{p}_n \textbf{ end},\ \mathrm{v}_0), \text{ if } \mathrm{l} \notin \{\mathrm{j}_1,...,\mathrm{j}_n\} & &
\end{array}
$$

The reduction relation **e** → **e'** defines what can happen as the evalution of a program proceeds. In fact, the relation may be read as "process **e** can evolve to the process **e'**".

The most basic rule of reduction is the one specifying what happens when an input prefix meets an output atom:

$$\frac{\mathit{match}(\mathbf{p}, \mathbf{v}) \text{ defined}}{\mathbf{x!v} \mid (\mathbf{x?p} > \mathbf{e}) \rightarrow \mathit{match}(\mathbf{p}, \mathbf{v})(\mathbf{e})}$$

Similarly, when a replicated input prefix meets an output atom, the result is a running instance of the body of the input prefix plus a fresh copy of the replicated input itself:

$$\frac{\mathit{match}(\mathbf{p}, \mathbf{v}) \text{ defined}}{\mathbf{x!v} \mid (\mathbf{x?*p} > \mathbf{e}) \rightarrow \mathit{match}(\mathbf{p}, \mathbf{v})(\mathbf{e}) \mid (\mathbf{x?*p} > \mathbf{e})}$$

The next two rules allow reduction to proceed under parallel composition and declaration:

$$\frac{\mathbf{e_1} \rightarrow \mathbf{e'_1}}{\mathbf{e_1} \mid \mathbf{e_2} \rightarrow \mathbf{e'_1} \mid \mathbf{e_2}}$$

$$\frac{\mathbf{e} \rightarrow \mathbf{e'}}{\mathbf{let\ d\ in\ e\ end} \rightarrow \mathbf{let\ d\ in\ e'\ end}}$$

Finally, we have structural congruence rules. The first two state that parallel composition is commutative and associative, while the third, often called the rule of *scope extrusion*, states that a new channel **x** can be communicated outside its original scope:

$$\mathbf{e_1} \mid \mathbf{e_2} \equiv \mathbf{e_2} \mid \mathbf{e_1}$$

$$(\mathbf{e_1} \mid \mathbf{e_2}) \mid \mathbf{e_3} \equiv \mathbf{e_1} \mid (\mathbf{e_2} \mid \mathbf{e_3})$$

$$\frac{\mathbf{x} \notin \mathrm{fn}(\mathbf{e_2})}{\textbf{let new x in } \mathbf{e_1} \textbf{ end} \mid \mathbf{e_2} \equiv \textbf{let new x in } \mathbf{e_1} \mid \mathbf{e_2} \textbf{ end}}$$

A.5 Derived forms

The previous definitions form the very basic core of PICT. However, it is not very convienient to program using only the core constructs. PICT defines a set of higher-level abstractions promoting PICT to a real programming language. In the subsequent presentation we will present some of these higher-level abstractions. In particular, we present the process abstraction **def** and **abs**, the function abstraction of processes, the value declaration **val**, and sequencing.

The abstraction **def** assigns a process expression a name. This name can be used in the rest of the program as abbreviation for the process expression. Moreover, the abstraction **def** allows one to define mutual recursive process expressions. The abstraction **def** has the form

ProcessDeclaration ::= **def** $Name_1$ Pat_1 > $Process_1$
[**and** ... **and** $Name_n$ Pat_n > $Process_n$]

and is translated to

new Name$_1$**,..., Name**$_n$
run Name$_1$**?*Pat**$_1$ > **Process**$_1$
...
run Name$_n$**?*Pat**$_n$ > **Process**$_n$

For example, we can define two processes **tt** and **ff** taking as parameter a boolean an signal at the correspoding channel.

def tt[b:^[^[],^[]]] > b?[t,f] > t![]
and ff[b:^[^[],^[]]] > b?[t,f] > f![]

Then **tt![b]** and **ff![b]** can be used in the rest of the program as abbreviation for the processes **b?[t,f] > t![]** and **b?[t,f] > f![]**, respectively.

Anonymous process declarations like **let def x[] > e in x end** are often useful. Therefore, PICT provides a special form allowing the useless **x** to be omitted:

AnonymousProcessDeclaration ::= **abs** *Pat* > *Process*

which is translated to

let def x *Pat* > *Process* **in x end**

In PICT a function is implemented by a process that expects input values plus a reply channel along which the function result is sent. Assume we want to define a function *plusone* that increments the value of an integer by one. We can define this function as follows[2]:

```
def plusone[v,r] > r!(n+1)

new r
run plusone![3,r] | (r?x > printi!x)
```

Here, the channel **r** takes the role of the reply channel along the result can be read. The program fragment just "calls" the function *plusone* and, in parallel, waits for the result to be sent along channel **r**. This kind of process definition is so common that PICT provides a higher-level abstraction for it:

FunctionDeclaration ::= **def** *Name*[$\mathbf{p}_1$,...,$\mathbf{p}_n$] = *Value*

which is translated to

def *Name*[$\mathbf{p}_1$,...,$\mathbf{p}_n$,**r**] > r! *Value*

For example, the function *plusone* can now be rewritten

```
def plusone[v] = n+1

printi!(plusone[3])
```

Usually, while excecuting a program we need to store some temporary results. The value declaration **val** in PICT serves exactly this purpose. The **val** declaration can be used to assign a name to an abitrary value.

ValueDeclaration ::= **let val** *Pat* = *Value* **in** *Process* **end**

is translated to

let new r in r! *Value*| **r?** *Pat* > *Process* **end**

The expression on the left of the = can be an arbitrary pattern, a **val** declaration can be used to bind several variables at once. For example,

[2]The operator + is a builtin channel in PICT.

val [x,y] = [[],[a]]

binds **x** to [] and **y** to [**a**]. The reader should note that the translation implies that the body of the **val** declaration cannot proceeed until all bindings introduced be **val** have been established.

Finally, a very common result is a *continuation signal*, which carries no information but tells the calling process that its request has been satisfied and it is now safe to continue. The *continuation signal* is encoded as empty tuple []. PICT provides a "statement" operator **;** to build sequences of values. A sequence

v;e

is translated to

let val [] = v in e end end

For example, given th builtin function **prInt** which sends a continuation signal upon finishing, we can define a process that will always print the number 3 followed by the number 4.

run prInt[3];prInt[4];skip

The final **skip** is needed in order to finish to calculation. Fortunately, there exists an abbreviation in PICT that allows one to omit **skip** such that the process can simply be rewritten as

run prInt[3];prInt[4];

Appendix B

Typing rules for $\pi\mathcal{L}$

B.1 Judgements

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash K$	K is a well-formed constant type in Γ
$\Gamma \vdash T$	T is a well-formed type in Γ
$\Gamma \vdash A$	A is a well-formed agent in Γ
$\Gamma \vdash F$	F is a well-formed form in Γ
$\Gamma \vdash v : T$	value v has type T in Γ
$\Gamma \vdash F : T$	form F has type T in Γ
$\Gamma \vdash T <: S$	T is a subtype of S in Γ

B.2 Basic rules

(*Empty environment*)
$$\emptyset \vdash \diamond$$

(*Environment v*)
$$\frac{\Gamma \vdash T \qquad v \notin dom(\Gamma)}{\Gamma, v : T \vdash \diamond}$$

(*Constant type*)
$$\frac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K}$$

(*Empty form type*)
$$\Gamma \vdash \langle\rangle$$

$$\frac{(Constant)\quad \Gamma \vdash K}{\Gamma \vdash c : K}$$

$$\frac{(Channel)\quad \Gamma, x : T \vdash \diamond}{\Gamma, x : T \vdash x : T}$$

B.3 Subtyping rules for types

$$\frac{(Subtyping\ reflexivity)\quad \Gamma \vdash T}{\Gamma \vdash T <: T}$$

$$\frac{(Subtyping\ transitivity)\quad \Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

$$\frac{(Subsumption)\quad \Gamma \vdash v : T \qquad \Gamma \vdash T <: S}{\Gamma \vdash v : S}$$

B.4 Rules for assigning types to forms

$$\frac{(Empty\ form)}{\Gamma \vdash \mathcal{E} : \langle\rangle}$$

$$\frac{(Form\ variable)\quad \Gamma, X : T \vdash \diamond \qquad \Gamma \vdash T <: \langle\rangle}{\Gamma, X : T \vdash X : T}$$

$$\frac{(Binding\ extension)\quad \Gamma \vdash F : T \qquad \Gamma \vdash T <: \langle\rangle \qquad \Gamma \vdash V : S}{\Gamma \vdash F\langle l{=}V\rangle : \mathrm{merge}(T\langle l{:}S\rangle)}$$

$$\frac{\begin{array}{l}(Polymorphic\ extension)\\ \Gamma \vdash F : T \qquad \Gamma \vdash X : S \qquad \Gamma \vdash T <: \langle\rangle \qquad \forall\ l_k \in \mathcal{L}(T) - \mathcal{L}(S)\ \Gamma \vdash S <: \langle\rangle \backslash l_k \\ \forall\ l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \Gamma \vdash S_{l_i} <: T_{l_i}\end{array}}{\Gamma \vdash FX : \mathrm{merge}(TS)}$$

$$\frac{(Projection)\quad \Gamma \vdash X : \langle l{:}T \rangle}{\Gamma \vdash X_l : T}$$

B.5 Subtyping rules for form types

$$\frac{(Form\ subtyping\ empty\ form)\quad \Gamma \vdash T_1 \quad ... \quad \Gamma \vdash T_n \quad l_i\ distinct}{\Gamma \vdash \langle l_1{:}T_1 \rangle ... \langle l_n{:}T_n \rangle <: \langle\rangle}$$

$$\frac{(Form\ subtyping)\quad \Gamma \vdash T_1 <: S_1 \quad ... \quad \Gamma \vdash T_n <: S_n \quad \Gamma \vdash T_{n+1} \quad ... \quad \Gamma \vdash T_{n+m} \quad l_i\ distinct}{\Gamma \vdash \langle l_1{:}T_1 \rangle ... \langle l_{n+m}{:}T_{n+m} \rangle <: \langle l_1{:}S_1 \rangle ... \langle l_n{:}S_n \rangle}$$

$$\frac{(Subtyping\ restriction)\quad \Gamma \vdash T <: \langle\rangle}{\Gamma \vdash T \backslash l <: \langle\rangle}$$

$$\frac{(Subtyping\ restricted\ types)\quad \Gamma \vdash T <: \langle\rangle \quad \Gamma \vdash S <: T}{\Gamma \vdash S \backslash l <: T \backslash l}$$

$$\frac{(Subtyping\ add\ restriction)\quad \Gamma \vdash T <: \langle\rangle \quad \Gamma \vdash S <: T \quad l \notin \mathcal{L}(T)}{\Gamma \vdash S \backslash l <: T}$$

B.6 Subtyping rules for channels

$$\frac{(Channel\ subtyping)\quad \Gamma \vdash S <: T \quad \Gamma \vdash T <: S}{\Gamma \vdash \updownarrow S <: \updownarrow T}$$

$$\frac{(Output\ channel\ subtyping)\quad \Gamma \vdash T <: S}{\Gamma \vdash \uparrow S <: \uparrow T}$$

$$\frac{(Input\ channel\ subtyping)\quad \Gamma \vdash S <: T}{\Gamma \vdash \downarrow S <: \downarrow T}$$

(*Output channel channel subtyping*)
$\Gamma \vdash \updownarrow T <: \uparrow T$

(*Input channel channel subtyping*)
$\Gamma \vdash \updownarrow T <: \downarrow T$

B.7 Rules for agents

(*Null*)
$\Gamma \vdash \mathbf{0}$

(*Parallel composition*)
$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \mid B}$$

(*Replication*)
$$\frac{\Gamma \vdash V(X).A}{\Gamma \vdash !V(X).A}$$

(*Restriction*)
$$\frac{\Gamma, a : \updownarrow T \vdash A}{\Gamma \vdash (\nu\ a : \updownarrow T)A}$$

(*Input*)
$$\frac{\Gamma \vdash V : \downarrow T \qquad \Gamma \vdash T <: \langle\rangle \qquad \Gamma, X : T \vdash A}{\Gamma \vdash V(X).A}$$

(*Output*)
$$\frac{\Gamma \vdash V : \uparrow T \qquad \Gamma \vdash F : T \qquad \Gamma \vdash T <: \langle\rangle}{\Gamma \vdash \overline{V}(F)}$$

Appendix C

The algorithm *Unify*

$$\mathit{Unify}(\emptyset) = \{\}$$

$$\mathit{Unify}(E \cup \{K_1 = K_2\}) =$$
$$\quad \text{if } K_1 \not\equiv K_2 \text{ then } \mathit{fail}$$
$$\qquad \text{else } \mathit{Unify}(E)$$

$$\mathit{Unify}(E \cup \{M = T\}) =$$
$$\quad \text{if } M \equiv T \text{ then } \mathit{Unify}(E)$$
$$\qquad \text{else if } M \text{ occurs in } T \text{ then } \mathit{fail}$$
$$\qquad\quad \text{else } \{M \mapsto T\} \circ \mathit{Unify}(E\{T/M\})$$

$$\mathit{Unify}(E \cup \{T = M\}) = \mathit{Unify}(E \cup \{M = T\})$$

$$\mathit{Unify}(E \cup \{\uparrow T_1 = \uparrow T_2\}) = \mathit{Unify}(E \cup \{T_2 = T_1\})$$

$$\mathit{Unify}(E \cup \{\updownarrow T_1 = \downarrow T_2\}) = \mathit{Unify}(E \cup \{T_2 = T_1\})$$

$$\mathit{Unify}(E \cup \{\updownarrow T_1 = \uparrow T_2\}) = \mathit{Unify}(E \cup \{T_2 = T_1\})$$

$$\mathit{Unify}(E \cup \{\updownarrow T_1 = \updownarrow T_2\}) = \mathit{Unify}(E \cup \{T_1 = T_2\})$$

$$\mathit{Unify}(E \cup \{\downarrow T_1 = \downarrow T_2\}) = \mathit{Unify}(E \cup \{T_1 = T_2\})$$

$$\mathit{Unify}(E \cup \{\uparrow T_1 = \downarrow T_2\}) = \mathit{Unify}(E \cup \{T_1 = T_2\})$$

$$\mathit{Unify}(E \cup \{\downarrow T_1 = \uparrow T_2\}) = \mathit{Unify}(E \cup \{T_1 = T_2\})$$

$Unify(E \cup \{\uparrow T_1 = \updownarrow T_2\}) = Unify(E \cup \{T_1 = T_2\})$

$Unify(E \cup \{\downarrow T_1 = \updownarrow T_2\}) = Unify(E \cup \{T_1 = T_2\})$

$Unify(E \cup \{FT_1 = FT_2\}) =$
  case $FT_1 = T\backslash\langle M_{T_1}\rangle\langle M_{T_2}\rangle$ and $FT_2 = S\backslash\langle M_{S_1}\rangle\langle M_{S_2}\rangle$:
    if $\mathcal{L}(S) - \mathcal{L}(T) = \emptyset$ and $\mathcal{L}(T) - \mathcal{L}(S) = \emptyset$
      then $Unify(E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = M_{S_2}\})$
    if $\mathcal{L}(S) - \mathcal{L}(T) = \emptyset$
      then let $Delta = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(T) - \mathcal{L}(S)$ in
        $Unify(E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{S_2} = Delta\} \cup \{M_{T_2} = \langle\rangle\})$
    if $\mathcal{L}(T) - \mathcal{L}(S) = \emptyset$
      then let $Delta = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(S) - \mathcal{L}(T)$ in
        $Unify(E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = Delta\} \cup \{M_{S_2} = \langle\rangle\})$
      else
        let $Delta1 = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(S) - \mathcal{L}(T)$ and
          $Delta2 = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(T) - \mathcal{L}(S)$ in
            $Unify(E \cup \{M_{T_1} = M_{S_1}\} \cup \{M_{T_2} = Delta1\} \cup \{M_{S_2} = Delta2\})$

  case $FT_1 = T\backslash\langle M_{T_1}\rangle\langle M_{T_2}\rangle$ and $FT_2 = S\backslash\langle M_S\rangle$:
    if $\mathcal{L}(S) - \mathcal{L}(T) = \emptyset$ then $Unify(E \cup \{M_{T_1} = M_S\} \cup \{T = S\})$
      else let $Delta = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(S) - \mathcal{L}(T)$ in
        $Unify(E \cup \{M_{T_1} = M_S\} \cup \{M_{T_2} = Delta\})$

  case $FT_1 = T\backslash\langle M_T\rangle$ and $FT_2 = S\backslash\langle M_{S_1}\rangle\langle M_{S_2}\rangle$:
    if $\mathcal{L}(T) - \mathcal{L}(S) = \emptyset$ then $Unify(E \cup \{M_T = M_{S_1}\} \cup \{T = S\})$
      else let $Delta = \langle l_1 : U_1\rangle ... \langle l_n : U_n\rangle$ with $l_1, ..., l_n \in \mathcal{L}(T) - \mathcal{L}(S)$ in
        $Unify(E \cup \{M_T = M_{S_1}\} \cup \{M_{S_2} = Delta\})$

  case $FT_1 = T\backslash\langle M_T\rangle$ and $FT_2 = S\backslash\langle M_S\rangle$:
    $Unify(E \cup \{M_T = M_S)\} \cup \{\ (T_{l_i} = S_{l_i}) \mid l_i \in \mathcal{L}(T) \cap \mathcal{L}(S)\ \})$

$Unify(E \cup \{S = \mathrm{PROJ}(l, T)\}) =$
  if $T_l = PROJ(l, T)$ then $Unify(E \cup \{S = T_l\})$
    else *fail*

$Unify(E \cup \{\mathrm{SUB}(T, S)\}) =$
  if $SUB(T, S)$ then $Unify(E)$
    else *fail*

$\mathit{Unify}(E \cup \{\text{SUBFORM}(FT, CT)\}) =$
  if $\mathit{SUBFORM}(FT, CT)$ then $\mathit{Unify}(E)$
    else *fail*

$\mathit{Unify}(E \cup \{U = \text{MERGE}(T, S)\}) =$
  let $TS = \mathit{MERGE}(T, S)$ in
    $\mathit{Unify}(E \cup \{U = TS\})$

$\mathit{Unify}(E \cup \{U = \text{POLYMERGE}(T \backslash \langle M_T \rangle, S \backslash \langle M_S \rangle)\}) =$
  let $(TS, M_S, Lackform) = POLYMERGE(T \backslash \langle M_T \rangle, S \backslash \langle M_S \rangle)$ in
    $\{M_S \mapsto Lackform\} \circ \mathit{Unify}(E\{Lackform/M_S\} \cup \{U = TS\})$

$\mathit{Unify}(E \cup \{\text{ELEMSUB}(S, T)\}) =$
  if $\mathit{ELEMSUB}(S, T)$ then $\mathit{Unify}(E)$
    else *fail*

Appendix D

Algorithm *Collect*

$Collect(X; A) =$
  let $ApplicationFound = false$ in
    let $ApplicationType = CollectAgentApplication(X; A; \langle\rangle)$ in
      if $ApplicationFound = true$ then return $ApplicationType$
      else return M, M *fresh*

$CollectAgentApplication(X; \mathbf{0}; T) = T$

$CollectAgentApplication(X; A_1 \mid A_2; T) =$
  return $CollectAgentApplication(X; A_2; CollectAgentApplication(X; A_1; T))$

$CollectAgentApplication(X; !A; T) =$
  return $CollectAgentApplication(X; A; T)$

$CollectAgentApplication(X; (\nu\ a)A; T) =$
  return $CollectAgentApplication(X; A; T)$

$CollectAgentApplication(X; V(Y).A; T) =$
  if $X \neq Y$
  then return $CollectAgentApplication(X; A;\ CollectVApplication(X; V; T))$
  else return $CollectVApplication(X; V; T)$

$CollectAgentApplication(X; \overline{V}(F); T) =$
  return $CollectFApplication(X; F;\ CollectVApplication(X; V; T))$

CollectVApplication$(X; c; T) = T$

CollectVApplication$(X; x; T) = T$

CollectVApplication$(X; Y_l; T) =$
  if $X = Y$ then *ApplicationFound* := *true*
    if $l \notin \mathcal{L}(T)$ then return $T\langle l : M\rangle$, M *fresh*
    else return T
  else return T

CollectFApplication$(X; \mathcal{E}; T) = T$

CollectFApplication$(X; Y; T) =$
  if $X = Y$ then *ApplicationFound* := *true*
    if T *does not contain a form tag* then return $T\langle M\rangle$, M *fresh*
    else return T
  else return T

CollectFApplication$(X; FY; T) =$
  return *CollectFApplication*$(X; F;$ *CollectFApplication*$(X; Y; T))$

CollectFApplication$(X; F\langle l = V\rangle; T) =$
  return *CollectFApplication*$(X; F;$ *CollectVApplication*$(X; V; T))$

Appendix E

Piccola language definition

Script ::= '**module**' *Name* [*Imports*] *Declarations* [*Main*]

Imports ::= '**load**' *NameList* ';'

NameList ::= *Name* [',' *NameList*]

Declarations ::= *Declaration* [*Declarations*]

Declaration ::= '**extern**' *String Name*
'**new**' *NameList*
'**run**' *Agent*
'**procedure**' *Name* '(' [*Name*] ')' '**do**' *Agent*
'**value**' *Name* '=' *Form*
'**function**' *Name* '(' [*Name*] ')' '=' *Form*

Main ::= '**main**' *Agent*

Agent ::= *PrimaryAgent* ['|' *Agent*]

PrimaryAgent ::= '**null**'
Location '!' *Form*
Location '?' '(' *Name* ')' '**do**' *Agent*
Location '?∗' '(' *Name* ')' '**do**' *Agent*
'**let**' *Declarations* '**in**' *Agent* '**end**'
'**if**' *BoolExpression* '**then**' *Agent* ['**else**' *Agent*] '**end**'
Application
'(' *Agent* ')'
PrimaryForm ';' [*Agent*]

BoolExpression ::= *Value Built-in-BoolOperator Value*

Location ::= *Name*
Variable '.' *Label*
PrimaryForm '.' *Label*

Application ::= *Location* '(' [*Form*] ')'

Form ::= *SeqForm*

SeqForm ::= *PrimaryForm* [';' *SeqForm*]

PrimaryForm ::= '<' [*FormElementList*] '>'
'**let**' *Declarations* '**in**' *Form* '**end**'
Application
'**return**' *PrimaryForm*
'**if**' *BoolExpression* '**then**' *Form* ['**else**' *Form*] '**end**'
'(' *Form* ')'
Variable

FormElementList ::= *FormElement* [',' *FormElementList*]

FormElement ::= *Variable*
Application
Label '=' *Value*
'**procedure**' *Name* '(' [*Name*] ')' '**do**' *Agent*
'**function**' *Name* '(' [*Name*] ')' '=' *Form*

Value ::= *Location*
Number
String
Form

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Franz Achermann. JPict – a framework for π-Agents. unpublished manuscript, 1998.

[3] Mehmet Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, NL, 1989.

[4] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On Bisimulations for the Asynchronous π-calculus. Technical Report RR-2913, INRIA Sophia-Antipolis, June 1996.

[5] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Operational Semantics of a Parallel Object-Oriented Language. In *Proceedings of Principles of Programming Languages (POPL'86)*, pages 194–208, January 1986.

[6] American National Standards Institute. *The Programming Language Ada Reference Manual*, LNCS 155, Springer, 1983.

[7] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, May 1996.

[8] John Barnes. *Programming in Ada'95*. Addison-Wesley, 1995.

[9] Manuel Barrio Solorzano. *Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto*. PhD thesis, Universidad de Valladolid, September 1995.

[10] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, NL, June 1994.

[11] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Proceedings POPL '90*, pages 81–94, San Francisco, January 1990.

[12] Borland International. *Delphi Benutzerhandbuch*, 1995.

[13] Gérard Boudol. Asynchony and the π-calculus. *Notes*, 1992.

[14] Niels Boyen, Carine Lucas, and Patrick Steyaert. Generalised mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, Programming Technology Lab, Vrije Universiteit Brussel, 1994.

[15] Gilad Bracha and William Cook. Mixin-based Inheritance. In Norman Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, October 1990.

[16] K. Brockschmidt. *Inside OLE 2: the Fast Track to Building Powerful Object-Oriented Applications*. Microsoft Press, 1993.

[17] Nat Brown and Charlie Kindel. *Distributed Component Object Model Protocol – DCOM/1.0*. Microsoft Corporation, January 1998.

[18] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. To appear in Theory and Practice of Object Systems, 1996.

[19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, May 1996.

[20] L. Cardelli and A. D. Gordon. Mobile ambients. *Lecture Notes in Computer Science*, 1378, 1998.

[21] Luca Cardelli. Typeful Programming. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[22] Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[23] Luca Cardelli and John C. Mitchell. Operations on Records. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.

[24] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[25] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[26] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[27] Brad Cox. *Superdistribution – Objects as Property on the Electronic Frontier.* Addison-Wesley, 1996.

[28] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[29] Laurent Dami. Labelled Reductions, Runtime Erros, and Operational Subsumption. In *Proc. 24th International Colloquim on Automata, Languages, and Programming*, LNCS1256, pages 782–793. Springer, 1997.

[30] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, February 1998.

[31] Nikolas G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[32] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A Reflective Model for First Class Dependencies. In *Proceedings OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 265–280, October 1995.

[33] Jesse Feiler and Anthony Meadow. *Essential OpenDoc.* Addison-Wesley, 1996.

[34] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[35] Daniel P. Friedman, Mitchell Wand, and Christopher T. Hayens. *Essentials of Programming Languages.* McGraw-Hill, 1992.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[37] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, September 1989.

[38] Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management.* Addison Wesley, 1995.

[39] J. Gosling and H. McGilton. *The Java Language Environment.* Sun Microsystems Computer Company, May 1995.

[40] Martin Hansen, Hans Hüttel, and Josva Kleist. Bisimulations for asynchronous mobile processes. In *Proceedings of the Tbilisi Symposium on Language, Logic, and Computation*, 1995.

[41] Robert W. Harper and Benjamin C. Pierce. A Record Calculus Based on Symmetric Concatenation. Technical Report CMU-CS-90-157R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1991.

[42] Kohei Honda. Two Bisimilarities in ν-Calculus. Technical Report CS report 92-002, Keio University, March 1993.

[43] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 133–147. Springer, July 1991.

[44] Kohei Honda and Mario Tokoro. On asynchronous Communication Semantics. In *ECOOP'91*, LNCS 612. Springer, June 1992.

[45] Kohei Honda and Nobuko Yoshida. On Reduction-Based Process Semantics. In *Proc. of 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, pages 371–387. Springer, 1993.

[46] Lalita A. Jategaonkar and John C. Mitchell. Type inference with extended pattern matching and subtypes. *Fund. Informaticae*, 19:127–166, 1993. Preliminary version appeared in *Proc. ACM Symp. Lisp and Functional Programming Languages,* 1988, 198-212.

[47] Cliff B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In E. Best, editor, *Proceedings CONCUR '93*, LNCS 715, pages 158–172. Springer, 1993.

[48] Grégor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[49] Grégor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241, pages 220–242. Springer, June 1997.

[50] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[51] Ray Lischner. *Secrets of Delphi 2*. Waite Group Press, 1996.

[52] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[53] Mark Lutz. *Programming Python: Object-Oriented Scripting*. O'Reilly & Associates, October 1996.

[54] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings ESEC '95*, LNCS 989, pages 137–153. Springer, September 1995.

[55] Ciaran McHale. *Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity Collegue, Dublin, Ireland, October 1994.

[56] M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.

[57] Vicki de Mey. Visual Composition of Software Applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 275–303. Prentice Hall, 1995.

[58] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[59] Microsoft Corporation. *Visual Basic Programmierhandbuch*, 1997.

[60] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, October 1995. Draft Version 0.9.

[61] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[62] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[63] Robin Milner. Functions as Processes. In *Proceedings ICALP '90*, LNCS 443, pages 167–180. Springer, July 1990.

[64] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[65] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[66] Robin Milner and Davide Sangiorgi. Barbed Bisimulation. In Werner Kuich, editor, *Proceedings of ICALP '92*, LNCS 623, pages 685–695. Springer, July 1992.

[67] John C. Mitchell. Type inference with simple subtypes. *Journal Functional Programming*, 1(3):245–286, 1991.

[68] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[69] Michael Morrison. *Presenting JavaBeans*. Sams.net Publishing, March 1997.

[70] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[71] Uwe Nestmann. What Is a 'Good' Encoding of Guarded Choice? Technical Report RS-97-45, BRICS Aalborg University, October 1997.

[72] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. Submitted to CONCUR'96, January 1996.

[73] Oscar Nierstrasz. Composing active objects. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[74] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[75] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.

[76] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[77] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[78] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[79] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.

[80] Gerald K. Ostheimer and Antony J. T. Davie. π-Calculus Characterizations of some Practical λ-Calculus Reduction Strategies. Technical Report CS/93/14, Department of Mathematical and Computing Science, University of St. Andrews, October 1993.

[81] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In *Proceedings of POPL'97*, ACM, pages 256–265, January 1997.

[82] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998.

[83] Michael Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79. Springer, 1992.

[84] Michael Papathomas. Behaviour Compatibility and Specification for Active Objects. In Dennis Tsichritzis, editor, *Object Frameworks*, pages 31–40. Centre Universitaire d'Informatique, University of Geneva, July 1992.

[85] David Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *5th GI Conference on Theoretical Computer Science*, volume LNCS 104, pages 167–183. Springer, 1981.

[86] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM Components in Haskell. In *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.

[87] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[88] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[89] Benjamin C. Pierce. Programming in the Pi-Calculus: An experiment in concurrent language design. Technical report, Computer Laboratory, University of Cambridge, UK, May 1995. Tutorial Notes for Pict Version 3.6k.

[90] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[91] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[92] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, March 1997.

[93] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1998.

[94] C. V. Ramamoorthy, Vijay Garg, and Atul Prakash. Support for Reusability in Genesis. *IEEE Transaction on Software Engineering*, 14(8):1145–1154, August 1988.

[95] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, 1992.

[96] Didier Rémy. Typing Record Concatenation for Free. Technical Report RR-1739, INRIA Rocquencourt, August 1992.

[97] J. A. Robinson. A Machine-Oriented Logic Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[98] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[99] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[100] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Computer Science Deptartment, University of Edinburgh, UK, May 1993.

[101] Davide Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA Sophia-Antipolis, April 1995.

[102] Davide Sangiorgi. A Theory of Bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. An extract appeared in *Proceedings of CONCUR '93*, LNCS 715, Springer.

[103] Davide Sangiorgi. An interpretation of Typed Objects into Typed Pi-calculus. Technical Report RR-3000, INRIA Sophia-Antipolis, September 1996.

[104] Jean-Guy Schneider and Markus Lumpe. Modelling Objects in PICT. Technical Report IAM-96-004, University of Bern, Institute of Computer Science and Applied Mathematics, January 1996.

[105] Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, October 1997. Hermes.

[106] Mary Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In Mansur Samadzadeh and Mansour Zand, editors, *Proceedings of SIGSOFT Symposium on Software Reusability*, pages 3–6, Seattle, April 1995. ACM Press.

[107] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[108] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[109] Sun Microsystems. *JavaBeans*, July 1997. Version 1.01.

[110] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[111] Peter Thiemann. *Grundlagen der funktionalen Programmierung*. B.G. Teubner, 1994.

[112] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[113] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA '87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, December 1987.

[114] Marc Van Limberghen and Tom Mens. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1):1–30, March 1996.

[115] Guido van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.

[116] Patrick Varone. Implementation of "Generic Synchronization Policies" in PICT. Technical Report IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, April 1996.

[117] Vasco T. Vasconcelos. Typed Concurrent Objects. In Mario Tokoro and Remo Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 100–117. Springer, July 1994.

[118] Pierre Viret. Viewing C++ Objects as Communicating Processes. Master's thesis, Laboratoire de Téléinformatique, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH, March 1996.

[119] Jan Vitek and Giuseppe Castagna. Towards a calculus of secure mobile computations. Electronic commerce objects, Centre Universitaire d'Informatique, University of Geneva, July 1998.

[120] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

[121] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[122] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93:1–15, 1991. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), 92–97.

[123] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, second edition, June 1997.