Typeful Compositional Styles

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät

der Universtität Bern

vorgelegt von

Stefan A. Kneubuehl

2003

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Dr. Franz Achermann

Institut für Informatik und angewandte Mathematik

Abstract

In component-based software development, a software application is composed of components that are *plugged* together. While components represent the stable parts, the changing or evolving *configuration* of a system is defined in *scripts*. The separation of changing from stable parts promises flexible software systems.

Compositional styles that define component interfaces, higher-level connectors and composition rules describe an architectural framework. Styles expose the large-scale architecture of a software system explicitly in scripts. But they lack a formal foundation allowing one to reason about styles.

We propose to use *Contractual types*, an experimental type system that can express both services provided and required by a component, as a formal basis to define compositional styles. We argue that this approach permits us (1) to verify the consistency of a style, (2) to check the implementation of a style for correctness, and (3) to detect compositional mismatches in component configurations.

We exemplify our claims by giving type-based definitions of some compositional styles. Implementing a contractual type checker for the composition language Piccola allows us to verify existing style implementations in Piccola against the definitions given. A flexible implementation of the Piccola language in Java provides a basis for the experiments with the type system.

Contents

List of Tables

Chapter 1

Introduction

1.1 Software Composition

We live in a changing world. Real world software systems evolve as the requirements and environment are changing.

The component-oriented software engineering approach tries to minimize the effort needed to change software by separating the changing from the stable parts of a system [NA00]. *Components* are the stable parts that are combined using *connectors*. The software system is thought of as a *configuration* of components.

A *connector* represents an interaction protocol of components. It defines the set of services the components use to communicate with each other. Services *provided* by one component may be *required* by the other component an vice versa. Furthermore, the protocol lays down rules for the components' interaction.

Thus, a configuration of components can be thought of as a graph, the nodes representing the components, the edges the connections between them. In order to constitute a software system, such a configuration must be *sound*, it's components must adhere to the protocol defined by their connectors.

Thinking of a software system as a configuration of components, changing the system comes down to applying the following operations to the configuration:

- Add, replace, or remove components.
- Add, alter, or remove connectors.

An altered component configuration must be sound in order to represent a (new) software system. To guarantee this, the components must be checked against connector specifications.

1.1.1 Current Problems

Current real-world applications have characteristics that hinder their evolution or make it more difficult:

- *Rich interfaces* lead to complex interaction protocols.
- *Frameworks* enforce fixed interaction mechanisms that cannot be altered or extracted from the framework. Furthermore, the object interactions imposed by a framework are often hard to understand, as they are spread throughout the framework.
- *Object-oriented languages* are designed to expose the object interfaces, not their interactions [Nie02]. It's therefore hard to understand and alter connectors.

Short, the architecture of current software systems is not explicit in the software's source code. This makes it hard to understand the architecture, and to extract the connector specifications. But having a clear picture of the architecture, especially of the connectors is essential for both integrating new components in a system and altering existing ones.

Architectural description languages such as Wright [All97] have been designed to describe the connectors and configuration of a software system. But similar to design description languages like UML, these description languages have the drawback that they are not directly connected with the source code. Therefore, they tend to be out of sync with the system, similar to comments in the source code [DDN02].

1.1.2 The Goal

An ideal environment for software composition that enables low-cost evolution should meet the following criteria:

- *Explicit architecture*: The architecture of a system should be encoded explicitly in its source code. Changing the architecture should be as easy as altering its graph representation.
- *Encodable rules*: Connectors describing the interaction protocols between components including rules should be encodable in the system's source code.
- *Architectural styles*: The possibility to formally define different architectural styles [AAG95] shoud be given.
- *Verification*: Automated verification techniques should be able to check component and connector implementations against formally defined styles.

Such properties of a software composition environment enable fast, low-cost software evolution, because

- The architecture is easy to understand and alter.

- Automated verification detects architectural mismatches.
- Interaction protocols are not spread throughout the system and thus easy to change and maintain.

1.1.3 The Pieces

This thesis proposes to use several technologies that work well together to provide a composition environment that tries to meet the above criteria.

Component algebras have been proposed as a means to describe architectural styles [AN01]. They represent components as elements of sorts and connectors as operators. Component algebras can express what components can be composed with each other, and what the result of such a composition is, but they don't allow to specify the interaction protocol.

Piccola is a composition language based on a formal, π-like calculus [Ach02]. Piccola uses *forms*, immutable and extensible records as values. Forms can be used to represent external components providing and requiring services. Providing customizable operator syntax, piccola has been used to implement architectural styles that have been defined as component algebras. The interaction protocol has been specified as Piccola source code. But Piccola can detect compositional mismatches only at runtime.

Contractual Types are an experimental type system for forms that allow not only to express what a component *provides*, but also what it *requires*.

Existing OO libraries such as the Java API provide a lot of functionality that could be wrapped as components and adapted to different architectural styles.

1.1.4 Putting the Pieces Together

The contribution of this thesis is to enhance the above mentioned pieces in such a way that they fit neatly together. It proposes to:

- use contractual types to formalize architectural styles expressed as component algebras.
- use a form-based composition language such as Piccola to implement component algebras.
- represent external components as forms, and adapt them to fit the roles defined by the component algebras.
- use contractual types as a type system for Piccola to verify implemented component configurations.

In particular, this thesis contributes the following:

- A Piccola runtime environment is implemented in Java in order to experiment with a form-based composition language. The core of the environment is a virtual machine that implements the Form Calculus and thus

can be used to implement other form-based languages. Key aspects of the implementation and the formal specification of the virtual machine are presented in chapter 2.

- In chapter 3, an enhanced Form Calculus is presented that allows us to represent external components as forms. The Contractual Types are enhanced accordingly in order to be able to type external components. The enhancement is justified by using it to formally specify the not yet formalized language bridge between Java and Piccola.
- In chapter 4, the enhanced Contractual Types are presented. It is shown how this type system is used to formally specify architectural styles and to verify an implementation of such a style in Piccola.
- Chapter 5 concludes the thesis by summarizing what has been learned pursuing this approach, and comparisons with related work are made.

In the rest of this chapter, the notion of component algebras is presented, also giving an example architectural style. Furthermore, the Form Calculus is introduced.

1.2 Architectural Styles

An architectural style [AAG95] formally describes a set of component *roles* and connectors. It specifies what interface a component must have in order to play a certain role. The connectors define how the components are plugged together.

1.2.1 Component Algebras

It has been proposed that architectural styles be expressed as *component algebras* [AKN00]: Components of different sorts can be combined using operators that plug the components together, yielding composed components. A component algebra defines

- *Component sorts*, a role description for components. A component sort defines the services a component must *provide* as well as the services it may *require*. A component belongs to a sort, if it provides all services needed and requires no more than the allowed services.
- *Operators*, describing how two or more components of given sorts are *composed*. It specifies how the provided and required services are connected and it defines the resulting component yielded by the operation.

Component algebras provide a *higher level of abstraction* than scripts that just wire components together. The sorts define compact, generic interfaces for components that take into account not only services provided by a component, but also its *required* services. The operators are general component *connectors* that encode composition rules not only for specific components, but for all components of the corresponding sorts.

Source | *Sink*	→	*Pipe*	close and start flow
Source | *Filter*	→	*Source*	appending a filter a source
Filter | *Filter*	→	*Filter*	composition of filters
Filter | *Sink*	→	*Sink*	prepending a filter to a sink
Source & *Source*	→	*Source*	sequential composition of sources
Source + *Source*	→	*Source*	parallel composition of sources
Sink + *Sink*	→	*Sink*	multiplex flow to two sinks

Table 1.1: The Pipes and Filters Operators

1.2.2 Pipes and Filters - A Sample Component Algebra

In this section, a sample component algebra based on the well-known architectural style of *Pipes and Filters* is presented. Three sorts of components can be identified:

- *Sources* produce data and push it downstream.
- *Sinks* consume data from upstream.
- *Filters* accept data from upstream, process the data and push it further downstream.

The inter-component communication is based on two services: The *put* service is used to push data to a downstream component, the *close* service is called to signal the end of the data flow. The two services are provided by components that accept data (*Filters* and *Sinks*) and required by data-producing components (*Sources* and *Filters*).

Various operators are defined that allow us to compose Sources, Filters, and Sinks to linear, merging or forking pipes (see table 1.1):

The *Pipe* operator | is used to compose a linear pipe. The services provided by the right-hand component are wired to the corresponding required services of the left-hand component. Depending on the composed sorts, the resulting component is a Source (when piping a Source to a Filter), a Filter (when piping two Filters), or a Sink (when piping a Filter to a Sink).

If a Source is piped to a Sink, the result is the empty component *Pipe* that neither provides nor requires services. This operation has the additional effect of starting a process that has the Source producing data and pushing it towards the sink.

The Pipe operator allows us to define data streams much like in unix shells:

$$aSource \mid aFilter \mid aSink$$

The *Sequentional Composition* operator & is used to compose multiple Sources. The resulting component is a Source that first pushes the data produced by

the left-hand component downstream. When the left-hand Source signals the end of data, the right-hand Source is allowed to produce its data and push it downstream.

The *Parallel Composition* operator + additionally allow us to merge or split streams. The composition of two Sources again is a Source that pushes the data produced by the two Sources downstream in no particular order. Composing two Sinks with this operator yields a Sink that forwards the received data to both original components, forking the stream.

Using a component algebra, the architecture of a software system can be expressed very *explicitly*. For example, the expression below can be easily interpreted:

$$(Source_1 + Source_2) \mid ((Filter \mid Sink_1) + Sink_2)$$

Five components are composed in the following way: Two Sources procuce data in parallel. The data is pushed to two sinks, one of which will received the data filtered by the Filter component.

1.3 Forms

Forms are a unifying concept for representing components, objects, services, and environments in a composition language. A form may represent

- a *service*.
- a immutable, extensible *record* that binds labels to forms. Thus, forms are *first-class values*.
- an *environment*, in which an expression is evaluated.

The Form Calculus presented here is based on the *Pure Form Calculus* introduced in [Nie03]. Additionally, context lookup is defined.

The Form Calculus is intended to be a subset of the Piccola Calculus [Ach02]. Yet to formally embed the Form Calculus in the Piccola Calculus is beyond the scope of this work.

1.3.1 Syntax

The syntax of the Form Calculus is given in table 1.2. The meta-variables E, F range over reduceable form *expressions*, and U, V, W range over non-reduceable, yet open form *pre-values*. Form *values* are closed pre-values, that is pre-values containing no unbound variables. x ranges over *labels*, the form calculus' identifiers. Form expressions have the following meaning:

- The *empty form* is the form that is not a service and that has no bindings.

E, F	$::=$	ε	Empty form	U, V, W	$::=$	ε	
	$\mid$	x	Label lookup		$\mid$	x	
	$\mid$	$\mathbf{R}$	Context lookup		$\mid$	$\mathbf{R}$	
	$\mid$	$x \mapsto F$	Label binding		$\mid$	$x \mapsto Y$	
	$\mid$	$\lambda x.F$	Service definition		$\mid$	$\lambda x.F$	
	$\mid$	$F \cdot E$	Form extension		$\mid$	$Y \cdot Z$	
	$\mid$	$F\ E$	Service application				
	$\mid$	$F;E$	Sandbox				

Table 1.2: Syntax of the Form Calculus

$\varepsilon \cdot U$	$\equiv$	U	$x \mapsto V \cdot x \mapsto W$	$\equiv$	$x \mapsto W$	
$U \cdot \varepsilon$	$\equiv$	U	$x \mapsto V \cdot y \mapsto W$	$\equiv$	$y \mapsto W \cdot x \mapsto V$	$(x \neq y)$
$U \cdot (V \cdot W)$	$\equiv$	$(U \cdot V) \cdot W$	$\lambda x.E \cdot \lambda y.F$	$\equiv$	$\lambda y.F$	
$x \mapsto V \cdot \lambda y.F$	$\equiv$	$\lambda y.F \cdot x \mapsto V$				

Table 1.3: Structural equivalence of form values

- A *label* represents the form bound to that label in the environment.
- The *context* represents the environment.
- A *label binding* is a form that represents a record with one binding. It binds a label x to a form F.
- A *service definition* represents a service that can be invoked.
- *Form extension* is used to build composed forms containing multiple labels and possibly service.

Form extension is used to compose a form out of the primitives. For example, the following form contains two labels x and y, both bound to the empty form. Additionally, the form provides the identity service.

$$x \mapsto \varepsilon \cdot y \mapsto \varepsilon \cdot \lambda x.x$$

The extension is a asymmetric operation, the right-hand form overriding labels bound in the left-hand form as well as a service provided by the left form. In the following example, the right-hand form overrides both the x binding and the service of the left-hand form:

$$(x \mapsto \varepsilon \cdot y \mapsto \varepsilon \cdot \lambda y.y) \cdot (\lambda z.\varepsilon \cdot x \mapsto z) \equiv \lambda z.\varepsilon \cdot x \mapsto z \cdot y \mapsto \varepsilon$$

1.3.2 Semantics

The semantics of the form extension is expressed as structural equivalence of forms (see table 1.3). The equivalences show further that the empty form ε is neutral with respect to form extension, and that form extension is assoziative.

The *service lookup* U_λ of a form U is either the service represented by the form, or an error ($\perp$) if U does not provide a service. Similar, the *label lookup* U_x of U is the form bound to x in the form, or an error if U does not bind the label:

$$U_\lambda = \begin{cases} \lambda x.F & (U \equiv V \cdot \lambda x.F) \\ \perp & (\text{otherwise}) \end{cases} \qquad U_x = \begin{cases} W & (U \equiv V \cdot x \mapsto W) \\ \perp & (\text{otherwise}) \end{cases}$$

The semantics of the service application and sandbox are defined as reduction rules. In a service application, the left-hand form must represent a service. In that case, the application is reduced to a sandbox, meaning that the service is evaluated in the current environment extended with the argument bound to the service's parameter. If the left-hand form does not represent a service, the application yields an error.

$$\begin{array}{ll} U\ V \rightarrow \mathbf{R} \cdot x \mapsto V;F & (U_\lambda = \lambda x.F) \\ U\ V \rightarrow \perp & (U_\lambda = \perp) \end{array}$$

The sandbox models the evaluation of the right-hand form in an environment represented by the left-hand form value. Sandbox evaluation looks up labels in the provided environment.

$$U;F \rightarrow U[\![F]\!]$$

The reduction is therefore explained in terms of *form substitution* (see table 1.4): A label x is substituted by the form bound to x in the environment. If the label is not bound in the environment, the substitution yields an error. The context lookup $\mathbf{R}$ is substituted by the environment itself. Other constructs are handled transparently, excepting the label restriction and service definition. Here, the environment is extended with the label x to be hidden (the parameter respectively) bound to a label lookup x. This provides a identity substitution for the concerned label.

The nested reduction rule ensures that nested form expressions are reduced properly:

$$E[F] \rightarrow E[F'] \qquad (F \rightarrow F', \quad E[\cdot] \neq \lambda x.[\cdot])$$

The second condition in the above rule ensures that services are not evaluated by mistake.

1.3.3 Piccola Calculus

The Piccola calculus [Ach02] can be seen as a superset of the Form calculus. Additionally to forms, the Piccola calculus uses the concept of *agents* and *channels* known from the asynchronous π-calculus to model parallel computing.

$U[\![\varepsilon]\!]$	=	ε	Empty
$U[\![x]\!]$	=	U_x	Lookup label
$U[\![\mathbf{R}]\!]$	=	U	Lookup context
$U[\![x \mapsto F]\!]$	=	$x \mapsto U[\![F]\!]$	Close binding
$U[\![\lambda x.F]\!]$	=	$\lambda x.((U \cdot x \mapsto x)[\![F]\!])$	Close service
$U[\![F \cdot E]\!]$	=	$U[\![F]\!] \cdot U[\![E]\!]$	Close extension
$U[\![F\ E]\!]$	=	$U[\![F]\!]\ U[\![E]\!]$	Close application
$U[\![F;E]\!]$	=	$U[\![F]\!];E$	Close sandbox

Table 1.4: Form Substitution

The Piccola calculus also provides a mechanism for label *hiding* to hide labels bound in a form. To hide a label x in a form F, we can write $F \backslash x$.

Furthermore, a special operator **L** is provided by the Piccola calculus to support form introspection.

Functional Piccola agents are an intermediate representation used to define the semantics of the Piccola language. The functional agents can be embedded in the Piccola calculus. They replace the syntactical elements of the Piccola calculus used for parallel computing with the abstractions **run** and **new** for creating new agents, or channels respectively.

Thus, we can say that the Functional Piccola Agents, the basis of the Piccola language, consists of the Form calculus plus label hiding, introspection **L** and the primitive services **run** and **new** for parallel computing.

Chapter 2

JPiccola - A Flexible Language Implementation

2.1 Piccola Engine Architecture

The semantics of Piccola are given by a mapping of the terms of the Piccola syntax to agents of the Piccola calculus [Ach02][Nie03]. The mapping is explained by the following steps:

- Piccola terms are reduced to the simplified Piccola language, a subset of the Piccola syntax, eliminating the syntactical sugar features.
- The simplified Piccola terms are mapped to functional Piccola agents.
- Functional Piccola agents are embedded in the Piccola calculus.

To convert Piccola source code to Piccola machine instructions, multiple transformations have to be applied sequentially. This is a well-known application of the Pipe architectural style. The Piccola engine is divided into the following components:

- *Parser*: parses the Piccola source code and builds the corresponding syntax tree.
- *Simplifier*: transforms the syntax tree by eliminating the syntactical sugar features.
- *Compiler*: converts the syntax tree to Piccola machine instructions.
- *State Machine*: executes the instructions, and builds up form values.

The components use various data representations. *Plain text* is used as input for the parser. The parser's output is a *syntax tree*. The simplifier has a syntax tree both for input and output. The compiler transforms the simplified syntax tree to linear *instructions*. The Piccola machine reads instructions producing *forms*.

2.1.1 Parsing

Piccola provides some language abbreviations, also known as syntactical sugar [Ach02]. These abbreviations allow us to write Piccola code in a more intuitive way. For example, it is possible to define a curried abstraction:

```
add a b: a + b
```

This code is syntactically equivalent and can thus be rewritten or *simplified* to the following code, which corresponds directly to an expression in the calculus:

```
add = \a: (\b: a + b)
```

Piccola semantics are only provided for fully simplified forms. Thus, the syntax tree has to be simplified before any further processing.

In order to allow an easy simplification process, the Piccola syntax is redefined in way that clearly seperates expressions that can be simplified (*sugar form expressions*) and constructs that can not (*simple form expressions*). Simple form expressions are further separated into *terminal* and *non-terminal* expressions.

The abstract Piccola syntax is given in table 2.1. The lower-case names (literal, identifier, op, $\text{op}^{\{}$, $\text{op}^{\}}$) represent tokens. When both a simple and a sugar form match a token sequence, the simple form is preferred. For example, the token sequence

"def" "x" "=" "(" ")"

produces a FixedPoint syntax node, not a CompoundBinding.

A parser generation tool (JavaCC) [Jav] is used to generate a parser for the syntax given in table 2.1. The syntax tree produced by the parser has a different node type for every form expression (e.g. Empty or PrefixApplication).

2.1.2 Simplifying

Before the syntax tree can be processed further, the sugar form expressions need to be eliminated. This is done by a tree transformation: the syntax tree is traversed in post-order (children are visited first), replacing every node of the kind SugarForm by a sub-tree of SimpleForm nodes that represent the simplified form of the original node.

The simpified form of a SugarNode is given by the Piccola language abbreviations in [Ach02].

For example, the simplifications for a Collection expression are defined as follows:

$$\begin{array}{lll} \text{op}^{\{}\ \text{op}^{\}} & = \text{"DefaultOp"}\ \text{"."}\ \text{op}^{\{\}}\ \text{"("}\ \text{")"} & \text{(coll-empty)} \\ \text{op}^{\{}\ [T_1\ \text{","}]\ T_2\ \text{op}^{\}} & = \text{op}^{\{}\ [T_1]\ \text{op}^{\}}\ \text{"."}\ \text{"add"}\ \text{"("}\ T_2\ \text{")"} & \text{(coll-add)} \end{array}$$

Note that the rules only produce Application, Empty, Label, and Projection nodes, all being SimpleForm syntax nodes, provided the rule (coll-add) is applied multiple times.

Form	::=	SimpleForm | SugarForm
SimpleForm	::=	TSimpleForm | NTSimpleForm
TSimpleForm	::=	Empty | Constant | Label | Root
Empty	::=	"(" ")"
Constant	::=	literal
Label	::=	identifier
Root	::=	"root"
NTSimpleForm	::=	Abstraction | Application | Binding | Extension | FixedPoint | Projection | Sandbox
Abstraction	::=	"\" [identifier] ":" Form
Application	::=	Form Form
Binding	::=	identifier "=" Form
Extension	::=	Form "," Form
FixedPoint	::=	"def" identifier "=" Form
Projection	::=	Form "." identifier
Sandbox	::=	"root" "=" Form ["," Form]
SugarForm	::=	Collection | CompoundBinding | CurriedAbstraction | InfixApplication | NamedAbstraction | PrefixApplication | Quote
Collection	::=	op{[FormList] op}
CompoundBinding	::=	["def"] NestedLabel "=" Form ["," Form]
CurriedAbstraction	::=	"\" Params ":" Form
InfixApplication	::=	Form op Form
NamedAbstraction	::=	["def"] NestedLabel Params ":" Form ["," Form]
PrefixApplication	::=	op Form
Quote	::=	"'" Form ["," Form]
FormList	::=	[FormList ","] Form
NestedLabel	::=	("root" | identifier) { "." identifier }
Params	::=	{ identifier | "(" ")" }+

Table 2.1: Piccola syntax

Using the two rules above, the following Piccola source code is simplified in four steps:

```
[1, 2, 3]                                   (coll-add)
[1, 2].add(3)                               (coll-add)
[1].add(2).add(3)                           (coll-add)
[].add(1).add(2).add(3)                     (coll-empty)
DefaultOp.[_]().add(1).add(2).add(3)
```

2.1.3 Fixed Point Processing

Special attention needs the FixedPoint expression. The translation of a fixed point expression to Functional Piccola Agents is quite complicated since the concept of fixed points is not known in the calculus.

Therefore, channels are used to model a *fixed point cell*, that is able to store forms. The Functional Piccola expression creating such a cell is named *Fix*. The result of *Fix* is a service returning the current value of the cell. Additionally, the result provides a *set* label bound to a service used to assign a new value to the cell.

As an example, consider the following expression that creates a fixed point cell, binds it to x and assigns the form F to it:

$$x{\mapsto}Fix;(x;set\ F)$$

This expression creates a fixed point cell, binds it to x and reads the current value of the cell:

$$x{\mapsto}Fix;(x\ \varepsilon)$$

Using the *Fix* construct, a fixed point of the Piccola language can be translated to the the functional agents as follows:

$$[\![\text{"def"}\ \ x\ \text{"="}\ \ T]\!] = \mathbf{R}{\cdot}x{\mapsto}Fix;(x;set)\ \ (x{\mapsto}\text{lookup}[x].[\![T]\!])$$

First, a new fixed point cell is created and bound to x. Then, the functional agent $[\![T]\!]$ representing the implementation of the fixed point is translated using the lookup[·] transformation. This transformation ensures that every access to x inside the fixed point is redirected to the cell by replacing x with $x\ \varepsilon$. Finally, the value of the fixed point is assigned to the cell using the *set* service.

The lookup[·] transformation is implemented as a post-order traversion of the syntax tree.

2.2 Piccola State Machine

2.2.1 Machine State

The Piccola virtual machine models the Functional Piccola Agents which are embedded in the Piccola calculus. In a Piccola calculus term $C; A$ the form C denotes the context in which the agent A is evaluated. The evaluation is done by step-wise reductions of the term. Similar, the Piccola virtual machine evaluates a linearized agent (Piccola code) in a given context. During evaluation, both the current context and value change. Therefore, the Piccola machine needs two stacks to keep track of the current *context* and *value*.

When an application of an abstraction to a form is encountered in an agent, the evaluation of the current Piccola code must be suspended, and the code representing the abstraction has to be evaluated before resuming the evaluation of the original code. Thus, a third stack that stores the pieces of *code* being evaluated, is needed.

Summarizing, the *state* of the Piccola virtual machine is given by three stacks:

- The *value stack* contains forms and keeps track of the current value.
- The *context stack* containts forms representing the evaluation context.
- The *code stack* or execution stack contains Piccola code.

2.2.2 Instructions / State Transitions

The instructions or state transitions of the Piccola machine a shown in table 2.2. An instruction may have some requirements for the current state noted in the columns *state before*. The *state after* columns show the changes made to the different stacks.

For example, the execution of the **pushContext** instruction requires at least one form F on the value stacks. After the instruction has been executed, F will be removed from the value stack, and pushed on top of the context stack.

Following is a short description of the instructions. Most instructions are closely related to agents.

- **apply**: Applies the closure form $\lambda(p, C)$ to the argument F. The new context $\mathrm{ctx}(F, C)$ is the closure's context extended with the argument bound to the *dynamic* label. The closure's code p is pushed onto the code stack and will be executed next.
- **bind**(x): Binds the current value F the the label x.
- **cell**: Pushes a memory cell form $\mathrm{cel}(F)$ as current value. A memory cell can be modelled using a channel. The memory cell can be used both to introduce states in a Piccola program at a higher level than channels, and to represent the *Fix* construct needed to represent Piccola fixed points in functional Piccola agents as $Fix = \mathrm{cel}(\varepsilon)$.

instruction	state before			state after		
	value	context	code	value	context	code
apply	$F, \lambda(p, C), \ldots$	$C, \ldots$	$\ldots$	$\ldots$	$\text{ctx}(C, F), C, \ldots$	$p, \ldots$
bind(x)	F	$\ldots$	$\ldots$	$x \mapsto F, \ldots$	$\ldots$	$\ldots$
cell	$F, \ldots$	$\ldots$	$\ldots$	$\text{cel}(F)$	$\ldots$	$\ldots$
empty	$\ldots$	$\ldots$	$\ldots$	$\varepsilon, \ldots$	$\ldots$	$\ldots$
exit	$\ldots$	$C, \ldots$	$p, \ldots$	$\ldots$	$\ldots$	$\ldots$
extend	$F_1, F_2, \ldots$	$\ldots$	$\ldots$	$F_2 \cdot F_1, \ldots$	$\ldots$	$\ldots$
hide(x)	$F, \ldots$	$\ldots$	$\ldots$	$hide_x F, \ldots$	$\ldots$	$\ldots$
popContext	$\ldots$	$C, \ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
project(x)	$F, \ldots$	$\ldots$	$\ldots$	$F; x, \ldots$	$\ldots$	$\ldots$
pushContext	$F, \ldots$	$\ldots$	$\ldots$	$\ldots$	$F, \ldots$	$\ldots$
root	$\ldots$	$C, \ldots$	$\ldots$	$C, \ldots$	$C, \ldots$	$\ldots$
send(c)	$F, \ldots$	$\ldots$	$\ldots$	$\varepsilon, \ldots$	$\ldots$	$\ldots$
service(p)	$\ldots$	$C, \ldots$	$\ldots$	$\lambda(p, C), \ldots$	$C, \ldots$	$\ldots$
receive(c)	$\ldots$	$\ldots$	$\ldots$	$c?, \ldots$	$\ldots$	$\ldots$
variable(x)	$\ldots$	$C, \ldots$	$\ldots$	$C; x, \ldots$	$C, \ldots$	$\ldots$
where $\text{ctx}(F, C)$ = $C \cdot \mathit{dynamic} \mapsto F$						
$\text{cel}(F)$ = **new** ε; *send* $F \cdot \mathit{set} \mapsto \lambda x.(\mathit{receive}\ \varepsilon; \mathit{send}\ x \cdot x) \cdot$ $\lambda x.(\mathbf{R} \cdot d \mapsto \mathit{receive}\ \varepsilon; \mathit{send}\ d \cdot d))$						

Table 2.2: Piccola machine instructions

- **empty**: Pushes the empty form as current value on the stack.
- **exit**: Exits from a closure application by removing the top elements from the context and code stacks. Every Piccola code sequence representing a closure must end with this instruction.
- **extend**: Extends the second form F_2 on the value stack with the topmost form F_1. The two forms are replaced by the resulting form.
- **hide**(x): Hides the label x in the top form F on the value stack.
- **popContext**: Removes the top context from the stack. Together with **pushContext**, this instruction is used to linearize the sandbox agent.
- **project**(x): Projects the top form F on the value stack to the label x. F is replaced by the value bound to x in F.
- **pushContext**: Pushes the current value on the context stack.
- **receive**(c): Receives a form from channel c. This instruction blocks the execution until a form is available at c. The receive form is pushed on the value stack.
- **root**: Pushes the current context on the value stack.
- **send**(c): Sends the top form F of the value stack along the channel c. F is removed from the stack.

$[\mathbf{run}]$	=	variable(run)	run
$[hide_x]$	=	hide(x)	hide
$[\varepsilon]$	=	empty	empty form
$[\mathbf{R}]$	=	root	current root
$[A; B]$	=	$[A]$, pushContext, $[B]$, popContext	sandbox
$[A; x]$	=	$[A]$, project(x)	projection
$[A \cdot B]$	=	$[A]$, $[B]$, extend	extension
$[x]$	=	variable(x)	variable
$[x \mapsto A]$	=	$[A]$, bind(x)	binding
$[\lambda x.A]$	=	service($[A]$, exit)	abstraction
$[AB]$	=	$[A]$, $[B]$, apply	application
$[Fix]$	=	empty, cell	fixed point cell

Table 2.3: Transformation of functional Piccola agents to Piccola code

- **service**(p): Pushes a new closure form $\lambda(p, C)$ on the value stack. The closure's context is the current context, the closure's Piccola code is specified as parameter of the instruction.
- **variable**(x): Projects the current context C to the label x. The value bound to x in C is pushed on the value stack.

2.2.3 Translating Calculus Terms to Machine Instructions

Finally, the link between the Piccola calculus and the Piccola state machine is made by providing a translation from calculus terms to machine instructions. The machine architecture and the instructions have been chosen in a way that simplifies this translation.

Table 2.3 shows the translation of the Piccola calculus constructs.

There is no translation provided for the inspect expression **L**, as no Piccola source code produces that expression. Rather, the standard Piccola context provides a *inspect* service with the same behaviour.

Also no translation is available for the **new** expression. When translating source code to functional Piccola agents, **new** can only occur in the *Fix* construct when translating a fixed point. Similar to inspect, the standard Piccola context provides a *newChannel* service with the same semantics as **new**.

A special translation is provided for the projection $A; x$. As the projection is an often used language feature, the **project** instruction has been provided to replace the **pushContext**, **variable**, **popContext** sequence.

The *Fix* construct is directly translated to the **empty**, **cell** instruction sequence.

Chapter 3

Accessing External Components

A composition language typically composes components written in other languages [NM95]. Thus, the composition language must provide means to access such components: Entities of the composition language itself are used to represent, or *wrap*, external components. Threrefore, such entities may be called *wrappers*.

The mechanism of wrapping components is the basis for the interoperability between the two languages. It allows one language to manipulate runtime entities of the other language, thus building a *bridge* between the languages.

There are some basic requirements for a *bridging* mechanism.

- When using components in the composition language, it should be of no concern to the programmer in what language the component has been written. Thus, the bridging should be *transparent*.
- The *calling mechanism* of the components must be mapped to the composition language's calling mechanism.
- To allow full interoperability, every value that can be communicated to or from the component must be wrapped, too. More generally, the bridge must map *every possible value* of the component's language to the composition language.
- To ensure consistency, the bridging mechanism should be bijective. That means that a value that is passed forth and back through the bridge should still be the same value.

3.1 Piccola - Java Bridging

An approach to bridge Piccola to Java has been described by [Sch01] and [Ach02].

When bridging Piccola to Java, $\mathcal{O}$ becomes the set of Java objects, and $\mathcal{F}$ the set of Piccola forms. `null` $\in \mathcal{O}$ is the Java `null` value, and $\varepsilon \in \mathcal{F}$ is the Piccola empty form value. As Piccola is implemented in Java, forms are represented by Java objects, formally $\mathcal{F} \subset \mathcal{O}$.

3.1.1 Representing Java Objects as Forms

Forms representing Java objects provide bindings that allows Piccola to access the public fields and methods of the object.

The form representing a Java object provides a binding

- for each non-static public field of the object,
- for each set of non-static public methods with the same name (overloaded methods) of the objects.

A special case are Java objects of the `java.lang.Class` class. In Piccola, these objects are wrapped differently in order to represent a Java class. A form representing a Java object of the `java.lang.Class` class provides a binding

- for each static public field of the class represented by the object,
- for each set of static public methods with the same name of the class represented by the object,
- for the constructors of the class represented by the object.

As an example, consider the following Java class `MyClass`:

```
public class MyClass {
    public static int aStaticField;
    public static void staticFunc() { }
    public int aField;
    public void foo(int i) { }
    public void foo() { }
    public void func() { }
}
```

A form representing an object of that class provides the bindings `aField`, `foo`, and `func` in addition to the bindings representing the methods inherited from `java.lang.Object`.

The wrapper form of the `java.lang.Class` object representing `MyClass` provides the bindings `aStaticField`, `staticFunc`, and `new`. The `new` binding is a dedicated binding representing the constructors of `MyClass`.

The form representing a Java object O is denoted by wrap(O).

3.1.2 Up and down

When bridging between two languages, one language is typically embedded in the other. In context of the Agora programming language, de Meuter has called the embedded language the *up* level and the host language the *down* level [Meu98]. In our context, Piccola is the up level and Java the down level.

The entities of the up level are therefore Piccola forms, the ones of the down level are Java objects. In order to explain the briding, we will define the two functions *up* and *down* translating an entity from the down to the up level and vice versa.

The first condition for the up function is that the Java `null` value should be mapped to the empty form:

$$\mathrm{up}_1(\texttt{null}) \stackrel{\text{def}}{=} \varepsilon$$

Java objects that already are forms ($O \in \mathcal{F}$) need not be wrapped. Other Java objects $O \in \mathcal{O} \setminus \mathcal{F}$ are wrapped by a form object $\mathrm{wrap}(O) \in \mathcal{F}$, leading to the following approach for the up function:

$$\mathrm{up}_2(O) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & (O = \texttt{null}) \\ O & (O \in \mathcal{F}) \\ \mathrm{wrap}(O) & (\text{otherwise}) \end{cases}$$

Given that the equality of wrapped objects is based on the equality of the objects themselves, formally

$$\mathrm{wrap}(O) = \mathrm{wrap}(P) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad O = P.$$

The up function obviously is bijective.

In Piccola, wrapped objects often are extended by Piccola-specific bindings, such as a binding for the + operator for wrapped string objects. These additional extensions made in Piccola can be thought of as Piccola-level wrappers. The extended form should still be converted back to the same Java object:

$$\mathrm{down}(F \cdot \mathrm{up}(O)) = \mathrm{down}(O) \tag{3.1}$$

This desired behaviour can be achieved by modifying the up and down functions. The key idea is to extend the wrapped object $\mathrm{wrap}(O)$ by a binding of $\mathrm{wrap}(O)$ to the dedicated label *peer*. This ensures that the original wrapped object is still accessible though the *peer* binding even when the form returned by up is extended by other forms:

$$\mathrm{up}(O) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & (O = \texttt{null}) \\ O & (O \in \mathcal{F}) \\ (\mathrm{wrap}(O) \cdot \mathit{peer} \mapsto \mathrm{wrap}(O)) & (\text{otherwise}) \end{cases}$$

The down function is defined in such a way that it converts forms of the form $\mathrm{wrap}(O)$ and $(F \cdot \mathit{peer} = \mathrm{wrap}(O))$ to the object O:

$$\mathrm{down}(F) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \texttt{null} & (F = \varepsilon) \\ F & (\forall O \in \mathcal{O} : G \neq \mathrm{wrap}(O)) \\ O & (\exists O \in \mathcal{O} : G = \mathrm{wrap}(O)) \end{array} \right\}, \quad G = (\mathit{peer} \mapsto F \cdot F).\mathit{peer}$$

These definitions of up and down the first three requirements for a bridging mechanism. The remaining issue is that the above definitions of up and down are not bijective. This can be easily seen in equation 3.1. This can be resolved by introducing a new form equivalence.

3.1.3 Peer Equivalence

As two forms representing the same object should be considered equal, the following *peer equivalence* for forms is introduced:

$$F \doteq G \quad \overset{\text{def}}{\Leftrightarrow} \quad \text{down}(F) = \text{down}(G)$$

The peer equivalence is the same as the equality notion for *native* forms which do not represent an object. For such forms, it holds that

$$F = \text{down}(F),$$

as can be seen in the definition of down. Let F and G be native forms:

$$F \doteq G \quad \Leftrightarrow \quad \text{down}(F) = \text{down}(G) \quad \Leftrightarrow \quad F = G$$

For wrapped objects, it holds that:

$$\text{wrap}(O) \quad \doteq \quad \mathit{peer} \mapsto \text{wrap}(O)$$

Furthermore, the peer equality enhances the semantics of form extension in a consistent way for wrapped objects $\text{wrap}(O)$. A form extended with a form representing an object also represents that object. And a form representing an object extended by a 'normal' form still represents that object:

$$\begin{array}{lcl} \text{wrap}(O) & \doteq & F{\cdot}\mathit{peer} \mapsto \text{wrap}(O) \\ \text{wrap}(O) & \doteq & \mathit{peer} \mapsto \text{wrap}(O){\cdot}F \quad (\text{down}(F) = F) \end{array}$$

The above definition of the down function differs slightly from the former definitions down_f in [Sch01] and [Ach02]:

$$\text{down}_f(F) = \left\{ \begin{array}{ll} \texttt{null} & (F = \varepsilon) \\ G & (\forall O \in \mathcal{O} : G \neq \text{wrap}(O)) \\ O & (\exists O \in \mathcal{O} : G = \text{wrap}(O)) \end{array} \right\}, \quad G := (\mathit{peer} \mapsto F{\cdot}F).\mathit{peer}$$

Essentially, the difference concerns so called *fake peer forms*, i.e. of forms that have a *peer* label bound to a form that is not a wrapped object:

$$F \cdot \mathit{peer} \mapsto G, \quad G \neq \text{wrap}(G)$$

The former definitions of down would handle such forms by projecting them to the *peer* label, resulting in a violation of the bijectivity of down:

$$\text{down}_f \circ \text{protect}_f(\varepsilon) \quad = \quad \text{down}_f(\mathit{peer} \mapsto \varepsilon) \quad = \quad \text{down}_f(\varepsilon)$$

Such fake peer forms have been used to *protect* forms: Sometimes it is not desirable to down form arguments for an external function call. For example when storing a form in a external container using an add function, the desired effect is to pass the form as it is to the container, not to down it first.

In the following Piccola code, for example, the form $(\text{up}(1)\cdot \mathit{info} \mapsto \varepsilon)$ will be downed to the object 1 when passed as argument for add. The Java object representing 1 is stored in the list. The upped result of the get function is the form up(1), and the *info* binding has been lost.

```
list = Host.class("java.util.ArrayList").new()
list.add(1, info = ())
list.get(1).info  # Oops, info is not bound in form
```

This problem has been caused by the following protect function:

$$\text{protect}_f(F) \stackrel{\text{def}}{=} \mathit{peer} \mapsto F$$

However, a new definition of protect resolves this issue:

$$\text{protect}(F) \stackrel{\text{def}}{=} \text{wrap}(F) \cdot \mathit{peer} \mapsto \text{wrap}(F)$$

Now, the combination of the down and protect functions is the identity for all forms, especially for fake peer forms such as $\mathit{peer} \mapsto \varepsilon$:

$$\text{down} \circ \text{protect}(F) = \text{down}(\text{wrap}(F) \cdot \mathit{peer} \mapsto \text{wrap}(F)) = F$$

In the Piccola example, the protected form is downed resulting in the original form, which is stored in the list.

```
list = Host.class("java.util.ArrayList").new()
list.add protect(1, info = "Hello")
list.get(1).info  # Ok, info is bound in form
```

3.1.4 Argument passing

Some consideration is needed for the argument passing mechanism, i.e. for the definition of the downargs function. Java allows multiple arguments for function call, whereas Piccola services just take one form as argument. The first idea might be to model Java function calls with multiple arguments by curried Piccola services. But this approach fails due to Java's capability of function overloading, as is illustrated in the following Piccola pseudo code:

```
getProperty key def: # Call to System.getProperty

getProperty "key"    # Oops: Call System.getProperty(String)
                     # or return curried service for
                     # System.getProperty(String, String)?
```

The approach of Achermann [Ach02] is to map certain bindings of the argument form to the arguments. The value bound to the label val_i maps to the i-th argument for the corresponding Java function. The number of arguments is determined by the highest consecutive val-binding:

$$\text{numargs}(F) = \begin{cases} n & (\text{exists}(F, val_i), i = 1 \ldots n \wedge \neg\text{exists}(F, val_{n+1})) \\ 0 & (\neg\text{exists}(F, val_1)) \end{cases}$$

$$\begin{array}{ll} \text{downargs}: & \mathcal{F} \mapsto \mathcal{O}^n \\ \text{downargs}(F) & := (\text{down}(F.val_1), \ldots, \text{down}(F.val_n)), \quad n := \text{numargs}(F) \end{array}$$

This leads to the following syntax for Java calls in Piccola:

```
System = Host.class "java.lang.System"
System.setProperty(val1 = "key", val2 = "value")
System.out.println(val1 = 42)
```

As many Java methods have just one parameter, it's convenient to use the argument form itself as first (and only) argument. Thus, the conversion function is enhanced the the following form:

$$\text{downargs}(F) = \left\{ \begin{array}{ll} () & (F = \varepsilon) \\ (\text{down}(F.val_1), \ldots, \text{down}(F.val_n)) & (n > 0) \\ (\text{down}(F)), & (\text{otherwise}) \end{array} \right\}, \quad n := \text{numargs}(F)$$

Using this definition, the Java calls with one argument can be written more intuitively:

```
System = Host.class "java.lang.System"
System.out.println 42
```

This is the straightforward approach of mapping names to numbers (positions). In practice, val-bindings are not often used directly. More often, Java methods requiring more than one argument are wrapped. The wrapping service can be a curried function taking one argument form for each val-binding needed. Alternatively, it is a service taking a single argument form with labels describing rather the arguments' meaning than their position.

```
System = Host.class "java.lang.System"

# curried service allows partial argument binding
setProperty1 key value: System.setProperty(val1 = key, val2 = value)
setProperty1 "key" "value"
setValueForKey = setProperty1 "key"
setValueForKey "value"
```

```
# more intuitive labels in argument form
setProperty2 args: System.setProperty(val1 = args.key, val2 = args.value)
setProperty2(key = "key", value = "value")
```

We propose an other, more intuitive approach for argument passing. Instead of specifying the arugment positions using dedicated bindings, the arguments are specified using a native Piccola list. Such a list provides both the services *size* for looking up the number of arguments and *at* for looking up the argumetns themselves. This leads to the following definition of the *downargs* function:

$$\text{downargs}(F) = (F.\text{at}(1), \ldots, F.\text{at}(n)), \quad n = F.\text{size}()$$

The corresponding Piccola syntax is shown on the following example:

```
System = Host.class "java.lang.System"
System.setProperty["key", "value"]
System.out.println[42]
```

Currently, both argument passing methods are implemented in JPiccola.

3.1.5 Piccola-level wrapping

In practice, it is desirable to extend upped objects with specific Piccola bindings. This has been the motivation for the above definitions of the up and down functions.

Until now, there have been two different methods for Piccola-level wrapping:

- *Automatic* wrapping for Piccola literal constants (Strings and Numbers).
- *Manual* wrapping by essentially applying Piccola wrapper services to all forms returned from an external (Java) function call.

This has lead to rather cumbersome wrapper services. First, such services typically were recursive, since methods of many objects return values of the same type as the object. Second, all methods returning values had to be redefined in Piccola. Below is a simple example for a string wrapper:

```
Hook.wrapper =
  def wrapString X:
    X
    concat Y: wrapString X.concat(Y)
    _+_ Y: wrapString X.concat(Y)

"Hello" + "World"
```

In the above code, `wrapString` is called three times: twice implicitly for the literals `"Hello"` and `"World"` and once explicitly by the wrapper's + service.

For the new implementation of JPiccola, a new wrapping approach has been chosen:

- Every Java object introduced into Piccola is a candidate for automatic wrapping. These are objects representing Piccola literals, return values of Java methods called from Piccola, and field values accessed from Piccola.
- Piccola services can be registered as wrappers using the built-in `registerWrapper` service. This service takes a Java class name and a Piccola service as arguments. The service is registered as a wrapper for the specified Java class.
- For every candidate object, a suitable wrapper service is sought. When a wrapper for the object's class is registered, the wrapper service is applied to the upped object.
- The automatic wrapping is not recursive, i.e. no automatic wrapping is done while a wrapper service is executed. Otherwise, wrapper implementations could easily lead to endless loops (Consider a string wrapper implementation containing a string literal).

This wrapping strategy leads to much simpler wrapper code. The example code shows a string wrapper definition: The only thing required to do is to define the + operator, and to register the wrapper service.

```
wrapString X:
  'PEER = X.peer
  X
  _+_ = PEER.concat

registerWrapper "java.lang.String" wrapString

"Hello" + "World"
```

In the code above, `wrapString` is called three times automatically: twice for the literals `"Hello"` and `"World"`, once for the result of the `concat` method.

There are two reasons to prefer a built-in registration service to the old `Hook` lookup in the current Piccola context:

- The built-in registration service can store the services in a collection that provides faster lookup than forms (e.g. a `SortedMap` object).
- When registered internally, the wrapper services are still available when the `Hook` binding is missing in the current context (e.g. in a sandbox, see example below).

```
root = (println = println)
  println "Hello" + "World"  # Doesn't work with Hook-based wrapping
```

The wrapper lookup mechanism deserves some more consideration. The wrapper service for a Java object is looked up as follows:

- If the object is an array, the special wrapper registered under the class name 'array' is applied.
- Otherwise, the wrapper registered under the object's class name is applied. If no wrapper has been registered under that name, the wrapper is recursively looked up for the object's superclasses.

3.2 Implementation Considerations

Having formally defined the bridging mechanism, the implementation should be straightforward. However, some issues arise from some special properties of the Java language.

- Java generally distinguishes between two kinds of values: primitive values and object values. [Java Language Specification]. The Java Reflection API only supports the introspection of objects, not of primitive values. This must be taken care of by the bridge implementation.
- Java arrays are objects, but they do not provide any methods to access their elements directly. If an array is wrapped 'normally', there is no easy way to access the array. Thus, a special treatment of array objects by the bridge is required.
- Java, like other OO languages, allows method overloading. This means that methods of a class are not identified by their name only, but by their signature, consisting of name, number of parameters and parameter types.

3.2.1 Wrapping Primitive Values

Java has primitive types that are not objects. These are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. Fortunately, Java provides wrapper classes for these types, and the Java reflection API automatically wraps values of the primitive types with the corresponding wrappers:

Primitive type	Wrapper class
`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

However, the Java wrapper classes do not provide methods to access operations available for the primitive types, such as the + operator. The reflection API doesn't have that capability. Thus, methods that implement those primitive operations must be provided by the Piccola implementation.

The operations for the number types require a non-trivial implementation. A numerical operation can have two values of different types as arguments. The resulting value is of the greater of the two types.

The following piece of Java codes shows the implementation of the plus wrapper method for the primitive + operation:

```
public class NumberOp {
  private static final int UNKOWN  = 0;
  private static final int BYTE    = 1;
  private static final int SHORT   = 2;
  private static final int INTEGER = 3;
  private static final int LONG    = 4;
  private static final int FLOAT   = 5;
  private static final int DOUBLE  = 6;

  private static final int getType(Number number) {
    if (number.getClass() == Double.class) return DOUBLE;
    else if (number.getClass() == Float.class) return FLOAT;
    else if (number.getClass() == Long.class) return LONG;
    else if (number.getClass() == Integer.class) return INTEGER;
    else if (number.getClass() == Short.class) return SHORT;
    else if (number.getClass() == Byte.class) return BYTE;
    else return UNKNOWN;
  }

  public static int returnType(Number a, Number b) {
    return Math.max(getType(a), getType(b));
  }

  public static Number plus(Number a, Number b) {
    switch (returnType(a, b)) {
      case DOUBLE: return new Double(a.doubleValue() + b.doubleValue());
      case FLOAT: return new Float(a.floatValue() + b.floatValue());
      case LONG: return new Long(a.longValue() + b.longValue());
      case INTEGER: return new Integer(a.intValue() + b.intValue());
      case SHORT: return new Short(a.shortValue() + b.shortValue());
      case BYTE: return new Byte(a.byteValue() + b.byteValue());
      default: throw new RuntimeException("Invalid number type");
    }
  }
}
```

The Piccola-level wrapper service can be implemented using the above defined class:

```
NumberOp = Host.class "ch.unibe.piccola.NumberOp"
wrapNumber X:
  X
  _+_ Y: NumberOp.plus(val1 = X, val2 = Y)
```

The primitive operations for booleans are defined in a similar way. In order to implement branching and looping services in Piccola, a select method similar to the select message of boolean in SmallTalk has to be defined. The method has to take a boolean and a form as argument and project the form to different labels depending on the boolean value:

```
public class BooleanOp {
  public Form select(Boolean condition, Form cases) {
    return cases.project(condition ? "iftrue" : "iffalse");
  }
  public Boolean and(Boolean a, Boolean b) {
    return new Boolean(a.booleanValue() && b.booleanValue());
  }
}
```

In Piccola, the wrapper service is again defined based on the `BooleanOp` class.

```
BooleanOp = Host.class "ch.unibe.piccola.BooleanOp"
wrapBoolean X:
  X
  select cases: BooleanOp.select(val1 = X, val2 = protect cases)
  _&_ Y: BooleanOp.and(val1 = X, val2 = Y)
```

Note that the second argument for the `select` method is protected, as the original form is expected as argument, not the downed value.

3.2.2 Wrapping Arrays

Java arrays are not primitive values, but objects. The drawback is, that array objects do not provide any methods to access the array's elements. However, the `java.lang.reflect.Array` class of the Java reflection API provides such methods, and can be used to implement a Piccola-level array wrapper.

Another issue is that Java has an array class for every array component type. It is not feasible to register the array wrapper for every possible array class. Therefore, a pseudo class name `array` is introduced that allows the registration of an array wrapper for all arrays.

An example array wrapper is provided below:

```
Array = Host.class "java.lang.reflect.Array"
wrapArray X:
  X
  size: Array.getLength X
```

```
  get anIndex: Array.get(val1 = X, val2 = anIndex)
  set args: Array.set(val1 = X, val2 = args.at, val3 = args.value)
  forEach aService: for
    from: 0
    to:   size() - 1
    do i: aService(get i)

registerWrapper "array" wrapArray

Package = Host.class "java.lang.Package"
arr = Package.getPackages()
arr.forEach \anElement: println anElement
```

The above code registers the `wrapArray` service as generic array wrapper. The `getPackages` method returns an array that will be wrapped. Now the `forEach` service can be used to print a list of all known Java packages.

3.2.3 Dynamic Method Overload Resolution

The method overloading feature of Java leads to the situation that a Piccola label of a wrapped object does not represent one method, but all methods of that object with the same name. For example the class `java.io.PrintStream` defines 10 overloaded methods with the name `print`.

```
System = Host.class "java.lang.System"
System.out.println "Hello"  # Which of the 10 overloads should be called?
```

As Java resolves overloaded method calls at compile time, the Java reflection API doesn't provide the mechanism at runtime. It has to be implemented from scrach. An algorithm to find the most specific Java method at runtime for a given name and arguments has been described by [TT01].

Chapter 4

Typing Components

In this chapter, a type system for the form calculus is introduced. Based on these *contractual types*, and a representation of components in the form calculus, the notion of component categories is defined. Contractual types allow us to infer types for components that both express what is *provided* and *required* by a component. Categories introduce a notion of *sub-types* to the contractual type system and make is possible to formulate generic *interfaces* for components, laying a basis for formalizing compositional styles.

4.1 Contractual Types

Contractual types are a type system for typing form expressions. The type system presented here is based on the contractual types in [Nie03]. Additionally, the typing of the explicit context (**R**) is supported. The incorporation of the explicit context leads to a different, more uniform representation of requirements.

Contractual types are defined for any form expression, also for open ones. The type of an open form expression not only expresses what it provides, but also what it requires from its environment. Thus, type judgements have the following form:

$$\vdash F :: P \mid C$$

where P represents what F provides and C is a set of constraints on the free type variables in P, representing requirements derived from the usage of the free variables in F.

According to Cardelli,

> the fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. [Car97]

Being a type system for the form calculus, the contractual types are bound to detect the possible execution errors of the calculus, namely

$$
\begin{array}{rcl}
P, S & ::= & \epsilon \mid \sigma^+ \mid x{:}P \mid P{\cdot}P \mid R{\rightarrow}P \mid \rho \\
R, Q & ::= & \epsilon \mid \sigma^- \mid x{:}R \mid R{\wedge}R \mid P{\rightarrow}R \\
\phi, \tau & ::= & \sigma^+ \mid \sigma^- \mid \rho \\
C & ::= & \top \mid \bot \mid C, P \hookrightarrow R \\
T & ::= & P \mid C
\end{array}
$$

Table 4.1: Syntax of contractual types

- *lookup errors*, caused by the usage of labels not bound in the environment (e.g. $\varepsilon{;}x$), and
- *application errors*, arising from application expressions missing a service definition in their left-hand argument (e.g. $\varepsilon\ \varepsilon$).

The type system presented here goes one step further by deriving types incorporating *constraints* for open expressions, specifying *requirements* for an environment. Such an expression is guaranteed to execute error-free, if its environement fulfills the requirements stated by its type.

4.1.1 Syntax

The syntax of contractual types is presented in table 4.1. The provided types express what a form expression provides. A provided type is defined for every form pre-value syntax construct. For example, the provided type $x{:}P$ expresses that a form provides a label bound to a form with the provided type P. This is the provided type of the form expression $x{\mapsto}F$, given that F has the provided type P. Provided type variables σ^+ represent the type of form variables. In the provided type for services $R{\rightarrow}P$, R represents what is required of the service's argument, and P is the provided type of the service's result. The provided type of the environment of a form expressions is denoted by a dedicated type variable ρ.

Required types express what is required of a form expression. Requirements arise from applications and form variables. An application EF requires the left-hand form expression E to provide a service, expressed by the required type $P{\rightarrow}R$. A form variable x requires the environment to provide a binding for x. This is expressed by required type of the form $x{:}R$. Multiple requirements can be formulated with the construct $R{\wedge}Q$.

Type variables are special: excepting the context type variable ρ, they always occur in pairs, such as α^+, α^-. The positive variable α^+ is a placeholder for the unknown provided type of a form variable x. Its inverse, the negative type variable α^- is used in required type expressions stating constraints for the type of the form that will eventually bound by x. Sometimes, it is not necessary to distinguish between positive, negative, and the context type variables. In that case, the meta-variables ϕ and τ are used as place-holders for a type variable.

The structural equivalences of closed provided and required types are given in

$$
\begin{array}{rlrl}
\epsilon{\cdot}P &\equiv P & R{\wedge}Q &\equiv Q{\wedge}R \\
P{\cdot}\epsilon &\equiv P & R{\wedge}\epsilon &\equiv R \\
P{\cdot}(Q{\cdot}S) &\equiv (P{\cdot}Q){\cdot}S & x{:}(R{\wedge}Q) &\equiv (x{:}R){\wedge}(x{:}Q) \\
x{:}S{\cdot}R{\to}P &\equiv R{\to}P{\cdot}x{:}S & R{\to}P{\cdot}Q{\to}S &\equiv Q{\to}S \\
x{:}P{\cdot}x{:}S &\equiv x{:}S & P \mid C &\equiv P \mid C, \top \\
x{:}P{\cdot}y{:}S &\equiv y{:}S{\cdot}x{:}P \quad (x \neq y) & P &\equiv P \mid \top
\end{array}
$$

Table 4.2: Type equivalences

$$
\frac{}{\vdash \varepsilon :: \epsilon}\ \mathit{Empty} \qquad \frac{}{\vdash \mathbf{R} :: \rho}\ \mathit{Root} \qquad \frac{}{\vdash x :: \sigma^{+} \mid \rho \hookrightarrow x{:}\sigma^{-}}\ \mathit{Lookup}
$$

$$
\frac{\vdash E :: P \mid C}{\vdash x{\mapsto}E :: x{:}P \mid C}\ \mathit{Label} \qquad \frac{\vdash E :: P \mid C_1 \qquad \vdash F :: S \mid C_2}{\vdash E{\cdot}F :: P{\cdot}S \mid C_1, C_2}\ \mathit{Extend}
$$

$$
\frac{\vdash E :: P \mid C}{\vdash \lambda x.E :: \sigma^{-}{\to}P[\rho{\mapsto}\rho{\cdot}x{:}\sigma^{+}] \mid C[\rho{\mapsto}\rho{\cdot}x{:}\sigma^{+}]}\ \mathit{Abstract}
$$

$$
\frac{\vdash E :: P \mid C_1 \qquad \vdash F :: S \mid C_2}{\vdash E;F :: S[\rho{\mapsto}P] \mid C_1, C_2[\rho{\mapsto}P]}\ \mathit{Close}
$$

$$
\frac{\vdash E :: P \mid C_1 \qquad \vdash F :: S \mid C_2}{\vdash E\ F :: \sigma^{+} \mid C_1, C_2, P \hookrightarrow S{\to}\sigma^{-}}\ \mathit{Apply}
$$

Table 4.3: Type inference rules

table 4.2. The equivalences of provided types are derived from the corresponding form equivalences.

A type constraint $P \hookrightarrow R$ expresses that the provided type P must *satisfy* the requirements given by R.

4.1.2 Type Inference

The type inference for a form expression is defined recursively based on the syntax of the form calculus presented in chapter 1. The type inference rules are given in table 4.3.

The empty form provides nothing and has no requirements, and therefore the type ϵ.

The type of the context is represented by the dedicated type variable ρ.

A form variable requires the context to bind the variable. It provides whatever the form bound to the variable in the context provides. This is expressed by

the type $\sigma^+ \mid \rho \hookrightarrow x{:}\sigma^-$. For example, the type of the form expression y would be

$$\frac{}{\vdash y :: \alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}\ \mathit{Lookup}$$

The type of a binding form $x{\mapsto}E$ is derived from the type of E as given by the *Label* inference rule. The provided type is $x : P$ given that E has the provided type P. The requirements of the binding are the same as those of E. Given the type of y above, the type of $x{\mapsto}y$ is

$$\frac{\vdash y :: \alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}{\vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}\ \mathit{Label}$$

The type of the form expression $E{\cdot}F$ is derived from the type of E and F. Given that E has the provided type P and F provides S, the provided type of the expression is $P{\cdot}S$. The requirements of the expression consist of the unification of both the requirements for E and F. As an example, consider the form extension $(x{\mapsto}y){\cdot}y$. Given the types of both parts, the type of the expression is inferred as follows:

$$\frac{\vdash y :: \beta^+ \mid \rho \hookrightarrow y{:}\beta^- \qquad \vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}{\vdash y{\cdot}(x{\mapsto}y) :: \beta^+{\cdot}x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\beta^-, \rho \hookrightarrow y{:}\alpha^-}\ \mathit{Extend}$$

The service expression $\lambda x.E$ binds the open form variable x in E. Therefore, the context type variable ρ is substituted in the type of E with the type $\rho{\cdot}x{:}\sigma^+$, σ^+ standing for the type of x. As the service returns the value of E, its provided type has the form $\sigma^-{\rightarrow}Q$, given Q is the the provided type of E with the substitution applied. The additional constraint $\sigma^+ \hookrightarrow \epsilon$ ensures that the services parameter type will be ϵ, if E is a closed expression:

$$\frac{\vdash \varepsilon :: \epsilon}{\vdash \lambda x.\varepsilon :: \alpha^-{\rightarrow}\epsilon \mid \alpha^+ \hookrightarrow \epsilon}\ \mathit{Service}$$

As another example, the type of the abstraction $\lambda y.(x{\mapsto}y)$ is inferred:

$$\frac{\vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}{\vdash \lambda y.(x{\mapsto}y) :: \beta^-{\rightarrow}x{:}\alpha^+ \mid \rho{\cdot}y{:}\beta^+ \hookrightarrow y{:}\alpha^-, \beta^+ \hookrightarrow \epsilon}\ \mathit{Service}$$

In a sandbox expression $E;F$, the context E binds all open form variables in F. As a consequence, the context type variables ρ in the type of F are substituted by the provided type P of the context. This substitution is both applied to the provided type of F and its constraints C_2.

Table 4.4 defines type variable substitution for all type expressions. The meta-variables ϕ and τ stand for any type variable, that is a positive variable σ^+, a negative variable σ^- or the context variable ρ. The type expression T is either

$$
\begin{array}{rcl}
\epsilon[\phi{\mapsto}T] & = & \epsilon \\
\phi[\phi{\mapsto}T] & = & T \\
\tau[\phi{\mapsto}T] & = & \tau \qquad (\tau \neq \phi) \\
(x{:}P)[\phi{\mapsto}T] & = & x{:}P[\phi{\mapsto}T] \\
P{\cdot}R[\phi{\mapsto}T] & = & P[\phi{\mapsto}T]{\cdot}R[\phi{\mapsto}T] \\
R{\rightarrow}P[\phi{\mapsto}T] & = & R[\phi{\mapsto}T]{\rightarrow}P[\phi{\mapsto}T] \\
x{:}R[\phi{\mapsto}T] & = & x{:}R[\phi \mapsto T] \\
R{\wedge}Q[\phi{\mapsto}T] & = & R[\phi{\mapsto}T]{\wedge}Q[\phi{\mapsto}T] \\
P{\rightarrow}R[\phi{\mapsto}T] & = & P[\phi{\mapsto}T]{\rightarrow}R[\phi{\mapsto}T] \\
\top[\phi{\mapsto}T] & = & \top \\
(C, P \hookrightarrow R)[\phi{\mapsto}T] & = & C[\phi{\mapsto}T], P[\phi{\mapsto}T] \hookrightarrow R[\phi{\mapsto}T]
\end{array}
$$

Table 4.4: Type variable substitution

a provided type (if the variable ϕ to be substituted is negative) or a required type (otherwise).

In the following example, the type of the sandbox expression $x{\mapsto}y;x$ is deduced from the known type of the context, and the type of x, which is analogous to the known type of y:

$$
\frac{\vdash x{\mapsto}y :: x{:}\alpha^{+} \mid \rho \hookrightarrow y{:}\alpha^{-} \qquad \vdash x :: \beta^{+} \mid \rho \hookrightarrow x{:}\beta^{-}}{\vdash x{\mapsto}y;x :: \beta^{+} \mid \rho \hookrightarrow y{:}\alpha^{-}, x{:}\alpha^{+} \hookrightarrow x{:}\beta^{-}}\ \mathit{Close}
$$

Note that the context type variable ρ is replaced only in the second constraint, not in the one originating from the context.

Finally, the type of an application expression EF is inferred the following way: the provided type of the expression is represented by a type variable. Both the constraints on E and F also hold for the whole expression, and an additional constraint states that E must provide a service that satisfies the service application. As an example, the above typed service is applied to the empty form:

$$
\frac{\vdash \lambda y.(x{\mapsto}y) :: \alpha^{-}{\rightarrow}y{:}\alpha^{+} \qquad \vdash \varepsilon :: \epsilon}{\vdash \lambda y.(x{\mapsto}y)\ \varepsilon :: \mathit{beta}^{+} \mid \alpha^{-}{\rightarrow}x{:}\alpha^{+} \hookrightarrow \epsilon{\rightarrow}\beta^{-}}\ \mathit{Apply}
$$

The service constraint expresses that the service may not have any requirements on the argument (since it is the empty form), and that it can return a form of an arbitrary type β^{-}.

The presented rules allow us to infer a contractual type for any form expression. In the next section, a type checking algorithm is presented that allows us to reduce the inferred type expressions, and detect type errors.

4.1.3 Type Checking

Contractual types can be reduced by simplifying constraints using equivalences and by eliminating constraints. A type checking algorithm attempts to satisfy the constraints C in type judgement $\vdash F :: P \mid C$ by reducing the judgement to one of the following forms:

$$\vdash F :: P \mid \top \qquad \vdash F :: P \mid C \qquad \vdash F :: P \mid \bot$$

A judgement $\vdash F :: P \mid \top$ states that the form expression F is valid in *any* context. Given a form pre-value U, the form expression $U;F$ will evaluate without producing an error.

If the resulting type judgement has the form $\vdash F :: P \mid \bot$, the form expression F is erroneous and cannot be evaluated.

A type judgement $\vdash F :: P \mid C$ expresses that F will only evaluate in a context U that satisfies the constraints C.

The type checking algorithm first tries to simplify each constraint to one of the following forms:

$$\sigma^+ \hookrightarrow R \quad (R \neq \tau^-) \qquad P \hookrightarrow \sigma^- \quad (P \neq \tau^+) \qquad \sigma^+ \hookrightarrow \tau^-$$

A constraint $\sigma^+ \hookrightarrow R$ states that the provided type represented by the positive variable σ^+ must satisfy the type requirements R. The variable's inverse σ^- is a placeholder for such requirements. The constraint can thus be resolved by substituting the negative variable with the requirements R. This is formalized in the following *type checking rule*:

$$\frac{\vdash F :: P \mid C, \sigma^+ \hookrightarrow R \qquad (R \neq \tau^-)}{\vdash F :: (P[\sigma^- \mapsto R] \mid C[\sigma^- \mapsto R]} \; \sigma^+ \text{ does not occur in } P \mid C \qquad \textit{Bind left}$$

Similarly, constraints having the form $P \hookrightarrow \sigma^-$ are resolved by substituting inverse type variable σ^+ with the provided type P. This leads to the rule

$$\frac{\vdash F :: PC, S \hookrightarrow \sigma^- \qquad (P \neq \tau^+)}{\vdash F :: P[\sigma^+ \mapsto S] \mid C[\sigma^+ \mapsto S]} \; \sigma^- \text{ does not occur in } P \mid C \qquad \textit{Bind right}$$

Special treatment is needed for type constraints of the form $\sigma^+ \hookrightarrow \tau^-$. Such a constraint means that the types represented by σ and τ are equivalent. In this case, the constraint can be resolved by *renaming* one variable, substituting both the positive *and* negative forms of one variable with the other, as formulated by the following type checking rule:

$$\frac{\vdash F :: P \mid C, \sigma^+ \hookrightarrow \tau^-}{\vdash F :: P[\sigma^+ \mapsto \tau^+][\sigma^- \mapsto \tau^-] \mid C[\sigma^+ \mapsto \tau^+][\sigma^- \mapsto \tau^-]} \; \textit{Rename}$$

$P \hookrightarrow \epsilon \equiv \top \qquad (P \neq \phi)$		Trivial
$P \hookrightarrow Q \wedge R \equiv P \hookrightarrow Q, P \hookrightarrow R$		Split
$S' \cdot x{:}S \hookrightarrow x{:}R \equiv S \hookrightarrow R$		Resolve label
$S' \cdot y{:}S \hookrightarrow x{:}R \equiv S' \hookrightarrow x{:}R \qquad (x \neq y)$		Skip label 1
$\epsilon \hookrightarrow x{:}R \equiv \bot$		No label 1
$Q \rightarrow P \hookrightarrow x{:}R \equiv \bot$		No label 2
$S' \cdot (Q \rightarrow S) \hookrightarrow P \rightarrow R \equiv P \hookrightarrow Q, S \hookrightarrow R$		Resolve service
$S' \cdot x{:}S \hookrightarrow P \rightarrow R \equiv S' \hookrightarrow P \rightarrow R$		Skip label 2
$\epsilon \hookrightarrow P \rightarrow R \equiv \bot$		No service
$\alpha^+ \hookrightarrow R, \alpha^+ \hookrightarrow Q \equiv \alpha^+ \hookrightarrow R \wedge Q$		Join variable
$\rho \hookrightarrow R, \rho \hookrightarrow Q \equiv \rho \hookrightarrow R \wedge Q$		Join context
$C, \bot \equiv \bot$		Type error

Table 4.5: Type constraint equivalences

Type constraints are simplified using constraint equivalences presented in table 4.5. The empty type requirement ϵ is satisfied by any provided type, leading to the *Trivial* equivalence. The *Split* rule states that constraints having multiple requirements can be split up to multiple constraints. The next four *label* equivalences explain how a constraint of the form $P \hookrightarrow x{:}R$ are resolved: If the provided type P has a matching binding, the type provided by the binding must satisfy the requirement R, as expressed in the *Resolve label* rule. If P provides no matching binding, a type error has been detected ($\bot$). The *Resolve service* equivalence explains how service constraints are resolved. The service requirement $P \rightarrow R$ arises from an application, P being the argument's provided type, and R the requirements for the result of the application. A provided service type $Q \rightarrow S$ satisfies the service requirement, if the type of the argument satisfies the service's argument requirements Q and the result type of the service S satisfies R. If the functor E of an application EF does not provide a service, a type error has occurred (*No service*). The *Join* equivalences are used to collect multiple requirements for provided type variable. When one type error has been detected, the type checking should terminate, producing the type judgement $\vdash F :: P \mid \bot$. This is ensured by the *Type error* equivalence.

Using the type checking rules and the constraint equivalences presented above, a type checking algorithm can be formulated:

To check the type of a form expression F, follow these instructions:

1. Infer the type $F :: P \mid C$ of the form expression using the type inference rules.
2. Simplify each constraint using the equivalences in table 4.5 to constraints of the form $\alpha^+ \hookrightarrow R$, $P \hookrightarrow \alpha^-$, or $\rho \hookrightarrow R$.
3. Merge constraints over the same positive type variable or the context variable using the *Join* equivalences.
4. Resolve constraints by substituting type variables with type expressions using the type checking rules *Bind left*, *Bind right*, and *Rename*.

5. Iterate over points 2 to 4 until all constraints are resolved ($\top$), a type error is detected ($\bot$), or no further substitutions are possible.

Obviously, this algorithm will always produce one of the type judgements introduced at the beginning of this section. Every type judgement containing constraints is a *conditional* judgement $\vdash F :: P \mid C$. If we are lucky, the algorithm reduces the constraints either to $\top$ or $\bot$, producing a *valid* or *invalid* judgement. The different possible results of the algorithm and their meaning are summarized below:

$$\vdash F :: P \mid \top \quad F \text{ has the } \textit{valid} \text{ type } P$$

$$\vdash F :: P \mid C \quad F \text{ has the } \textit{conditional} \text{ type } P \mid C$$

$$\vdash F :: P \mid \bot \quad F \text{ has an } \textit{invalid} \text{ type}$$

If a form expression F has a valid type, it will evaluate error-free in any environment, especially in the empty environment ε. If F has an invalid type, there is no context in which F would evaluate error-free. If the expression is conditional, F will evaluate error-free only in certain environments.

4.1.4 Examples

In order to get a better understanding of the contractual types, some examples will be presented in this section. First, the type checking algorithm will be applied to the inferred types of the examples in section 4.1.2.

First, the example for the *Extend* inference rule is revisited. After the inference of the type of the extension, the constraints can only be simplified by merging them into one constraint using the *Join context* equivalence. A further simplification is allowed by the type equivalence expressing that the conjunction of required types distributes over label binding:

$$\dfrac{\dfrac{\dfrac{\vdash y :: \beta^+ \mid \rho \hookrightarrow y{:}\beta^- \qquad \vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}{\vdash y{\cdot}(x{\mapsto}y) :: \beta^+{\cdot}x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\beta^-, \rho \hookrightarrow y{:}\alpha^-}}{\vdash y{\cdot}(x{\mapsto}y) :: \beta^+{\cdot}x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\beta^- \wedge y{:}\alpha^-}}{\vdash y{\cdot}(x{\mapsto}y) :: \beta^+{\cdot}x{:}\alpha^+ \mid \rho \hookrightarrow y{:}(\beta^- \wedge \alpha^-)}$$

In the first *Service* example, the constraint of the inferred type for the service $\lambda x.\varepsilon$ can be immediately removed using the *Bind left* rule:

$$\dfrac{\dfrac{\vdash \varepsilon :: \epsilon}{\vdash \lambda x.\varepsilon :: \alpha^- {\to} \epsilon \mid \alpha^+ \hookrightarrow \epsilon}}{\vdash \lambda x.\varepsilon :: \epsilon {\to} \epsilon}$$

The service's type states that there are no requirements for the argument, and the service will return the empty form. As the service is a closed expression, the type has no requirements.

The simplification of the type derived for the second *Service* example is somewhat more difficult. As β^+ occurs more than once in the constraints, the second constraint cannot be resolved by the *Bind left* rule. However, the first constraint can be simplified using the *Resolve label* equivalence. Now, the two constraints can be merged as stated by the *Join variable* equivalence. The required type of the resulting constraint is equivalent to α^-. Finally, the remaining constraint is resolved by substituting β with α.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-}{\vdash \lambda y.(x{\mapsto}y) :: \beta^- {\to} x{:}\alpha^+ \mid \rho{\cdot}y{:}\beta^+ \hookrightarrow y{:}\alpha^-, \beta^+ \hookrightarrow \epsilon}}{\vdash \lambda y.(x{\mapsto}y) :: \beta^- {\to} x{:}\alpha^+ \mid \beta^+ \hookrightarrow \alpha^-, \beta^+ \hookrightarrow \epsilon}}{\vdash \lambda y.(x{\mapsto}y) :: \alpha^- {\to} x{:}\alpha^+ \mid \beta^+ \hookrightarrow \alpha^- {\wedge} \epsilon}}{\vdash \lambda y.(x{\mapsto}y) :: \alpha^- {\to} x{:}\alpha^+ \mid \beta^+ \hookrightarrow \alpha^-}}{\vdash \lambda y.(x{\mapsto}y) :: \alpha^- {\to} x{:}\alpha^+}$$

In the *Close* example, the type of the sandbox expression $x{\mapsto}y;x$ has been inferred (see step 1 below). The algorithm will first simplify the second constraint using the *Resolve label* equivalence (step 2). The resulting constraint allows us to apply the type checking rule *Rename* leading to a conditional type (step 3).

$$\dfrac{\dfrac{\dfrac{\vdash x{\mapsto}y :: x{:}\alpha^+ \mid \rho \hookrightarrow y{:}\alpha^- \qquad \vdash x :: \beta^+ \mid \rho \hookrightarrow x{:}\beta^-}{\vdash x{\mapsto}y;x :: \beta^+ \mid \rho \hookrightarrow y{:}\alpha^-, x{:}\alpha^+ \hookrightarrow x{:}\beta^-}}{\vdash x{\mapsto}y;x :: \beta^+ \mid \rho \hookrightarrow y{:}\alpha^-, \alpha^+ \hookrightarrow \beta^-}}{\vdash x{\mapsto}y;x :: \beta^+ \mid \rho \hookrightarrow y{:}\beta^-}$$

The type of the sandbox expression can be interpreted as follows: The expression can be evaluated in an environment that provides a binding for y, and its type will be the same as the one of the form bound to y.

In the *Apply* example, the empty form ε has been applied to the service $\lambda y.(x{\mapsto}y)$. Having inferred the type, the service constraint is resolved first using the *Resolve service* equivalence. The resulting constraints are satisfied by substituting β^+ with $y{:}\alpha^+$ and α^+ with ϵ as specified by the *Bind right* rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \lambda y.(x{\mapsto}y) :: \alpha^- {\to} x{:}\alpha^+ \mid \qquad \vdash \varepsilon :: \epsilon}{\vdash \lambda y.(x{\mapsto}y)\ \varepsilon :: \beta^+ \mid \alpha^- {\to} x{:}\alpha^+ \hookrightarrow \epsilon {\to} \beta^-}}{\vdash \lambda y.(x{\mapsto}y)\ \varepsilon :: \beta^+ \mid \epsilon \hookrightarrow \alpha^-, x{:}\alpha^+ \hookrightarrow \beta^-}}{\vdash \lambda y.(x{\mapsto}y)\ \varepsilon :: x{:}\alpha^+ \mid \epsilon \hookrightarrow \alpha^-}}{\vdash \lambda y.(x{\mapsto}y)\ \varepsilon :: x{:}\epsilon}$$

The resulting type $x{:}\epsilon$ is valid, stating that the service will return the empty form bound to x, as expected.

In the previous example, the service $\lambda y.(x{\mapsto}y)$ binds the argument to the label x. Now let's consider a service that extracts the form bound to x in its argument $\lambda y.(y;x)$. The type of this service is deduced as follows: First, the type is inferred using the *Lookup*, *Close*, and *Service* rules. The constraint $\alpha^+ \hookrightarrow x{:}\beta^-$ is removed by the *Bind left* rule. The remaining constraint is simplified using the *Lookup label* equivalence, and finally resolved by applying the *Bind left* rule again.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash y :: \alpha^+ \mid \rho \hookrightarrow y{:}\alpha^-} \qquad \dfrac{}{\vdash x :: \beta^+ \mid \rho \hookrightarrow x{:}\beta^-}}{\vdash y;x :: \beta^+ \mid \rho \hookrightarrow y{:}\alpha^-, \alpha^+ \hookrightarrow x{:}\beta^-}}{\vdash \lambda y.(y;x) :: \gamma^- {\to} \beta^+ \mid \rho{\cdot}y{:}\gamma^+ \hookrightarrow y{:}\alpha^-, \alpha^+ \hookrightarrow x{:}\beta^-}}{\vdash \lambda y.(y;x) :: \gamma^- {\to} \beta^+ \mid \rho{\cdot}y{:}\gamma^+ \hookrightarrow y{:}x{:}\beta^-}}{\vdash \lambda y.(y;x) :: \gamma^- {\to} \beta^+ \mid \gamma^+ \hookrightarrow x{:}\beta^-}}{\vdash \lambda y.(y;x) :: x{:}\beta^- {\to} \beta^+}$$

The type checking algorithm judges the service to have the valid type $x{:}\beta^- {\to} \beta^+$, expressing that the service looks up the label x in its argument.

The next two examples illustrate how type errors are detected. One kind of type errors arises from variable lookups in a context. The simplest way to produce such an error is to lookup the label x in the empty context. The *Close* inference rule substitutes the context type variable in the constraint with the empty type ϵ of the environment. It is obvious that the resulting constraint is not resolvable, formally expressed by the *No label 1* equivalence. The resulting judgement says that the expression has an invalid type.

$$\dfrac{\dfrac{\dfrac{}{\vdash \varepsilon :: \epsilon} \qquad \dfrac{}{\vdash x :: \alpha^+ \mid \rho \hookrightarrow x{:}\alpha^-}}{\vdash \varepsilon;x :: \alpha^+ \mid \epsilon \hookrightarrow x{:}\alpha^-}}{\vdash \varepsilon;x :: \alpha^+ \mid \bot}$$

The other kind of errors arise from applications with an invalid functor. Such an error can be most easily caused by applying the empty form to itself. The constraint generated by the *Service* inference rule is unsresolvable as stated by the *No service* equivalence:

$$\dfrac{\dfrac{\dfrac{}{\vdash \varepsilon :: \epsilon} \qquad \dfrac{}{\vdash \varepsilon :: \epsilon}}{\vdash \varepsilon\ \varepsilon :: \beta^+ \epsilon \hookrightarrow \epsilon {\to} \beta^-}}{\vdash \varepsilon\ \varepsilon :: \beta^+ \bot}$$

4.2 Component Categories

Components provide services and may require services from their environment. Therefore, we propose to represent components as *abstractions over their envi-*

$$
\begin{array}{rcl}
\bar{P}, \bar{S} & ::= & \epsilon \mid x{:}\bar{P}^v \mid \bar{P}{\cdot}\bar{P} \mid \bar{R}^v{\rightarrow}\bar{P}^v \\
\bar{R}, \bar{Q} & ::= & \epsilon \mid x{:}\bar{R}^v \mid \bar{R}{\wedge}\bar{R} \mid \bar{P}^v{\rightarrow}\bar{R}^v \\
\bar{P}^v & ::= & \sigma^+ \mid \bar{P} \\
\bar{R}^v & ::= & \sigma^- \mid \bar{R} \\
\bar{T} & ::= & \bar{R}^v{\rightarrow}\bar{P}^v
\end{array}
$$

Table 4.6: Syntax of component categories

ronment. This representation is motivated by Dami and Nierstrasz, who define a component as

> a static abstraction with plugs. [ND95]

Thus, in the form calculus, a component is represented as a service having the form

$$\lambda e.(e;F)$$

where F is the *implementation* of the component. The contractual type of such a service has the form $P{\rightarrow}R \mid C$.

In practice, we want to be able to *categorize* components. A component category $\mathcal{C}$ specifies the sets of services that a component

- must *at least provide*, and
- may *at most require*,

in order to be part of that category ($c \in \mathcal{C}$). Furthermore, we want to introduce a *partial order* of component categories.

In the following, the syntax for component categories is introduced as a subset of contractual types. This approach allows us to check the membership of a component in a category by an algorithm similar to the type-checking algorithm presented in section 4.1.3.

4.2.1 Syntax

A *component category* specifies what services a component must provide, and what services may be required by that component. A component category has the form $\bar{R}{\rightarrow}\bar{P}$, where $\bar{R}$ represents the *maximal* set of required services, $\bar{P}$ the *minimal* set of provided services. Both $\bar{P}$ and $\bar{R}$ are subsets of the provided and required types in the contractual type system (see table 4.6).

The empty type is represented by ϵ. Type variables stand for the type of data passed between components. Bindings that must be provided or may be required by a component are expressed by $x{:}\bar{P}^v$, or $x{:}\bar{R}^v$ respectively. The type $\bar{P}{\cdot}\bar{S}$

$(\sigma^-)^{-1} \equiv \sigma^+$	$(x{:}\bar{P})^{-1} \equiv x{:}\bar{P}^{-1}$	$(x{:}\bar{R})^{-1} \equiv x{:}R^{-1}$
$(\sigma^+)^{-1} \equiv \sigma^-$	$(\bar{P}{\cdot}\bar{S})^{-1} \equiv \bar{P}^{-1}{\wedge}\bar{S}^{-1}$	$(\bar{R}{\wedge}\bar{Q})^{-1} \equiv \bar{R}^{-1}{\cdot}\bar{Q}^{-1}$
$\epsilon^{-1} \equiv \epsilon$	$(\bar{R}{\to}\bar{P})^{-1} \equiv \bar{R}^{-1}{\to}\bar{P}^{-1}$	$(\bar{P}{\to}\bar{R})^{-1} \equiv P^{-1}{\to}R^{-1}$

Table 4.7: Inversion of component categories

states that both $\bar{P}$ and $\bar{S}$ must be provided by a component. Multiple allowed requirements are written as $\bar{R}{\wedge}\bar{Q}$. The type of a provided service is denoted by $\bar{R}^v{\to}\bar{P}^v$, $\bar{R}^v$ being the type of the argument, $\bar{P}^v$ the type of the result.

As the building blocks $\bar{P}$ and $\bar{R}$ of a category $\bar{R}{\to}\bar{P}$ are subsets of the provided and required contractual types. A component category is a special case of a provided type $R{\to}P$.

In contrast to general provided types of this form, component categories have two important properties:

- First, type variables are not allowed in the composed expressions $\bar{P}{\cdot}\bar{S}$ and $\bar{R}{\wedge}\bar{Q}$.
- Second, required and provided types are symmetrical.

The first property allows us to write conjunctions in a canonical form: We presuppose an ordering of labels by assuming that labels are words constructed from an ordered alphabet. The *canonical* or *normal* form of the provided and required parts of a category are structured as follows: If present, the service type $\bar{S}_p$, or $\bar{S}_r$ respectively, is placed left-most. Then the conjunction of bindings follows, with the bindings ordered according to their labels:

$$
\begin{array}{lcll}
\bar{P}^v & ::= & \sigma^+ & \mid \quad \bar{S}_p{\cdot}x_1{:}\bar{P}_1^v{\cdot}x_2{:}\bar{P}_2^v{\cdot}\ldots{\cdot}x_n{:}\bar{P}_n^v \qquad (x_1 < x_2 < \cdots < x_n) \\
\bar{S}_p & ::= & \epsilon & \mid \quad \bar{R}^v{\to}\bar{P}^v \\
\bar{R}^v & ::= & \sigma^- & \mid \quad \bar{S}_r{\cdot}x_1{:}\bar{R}_1^v{\wedge}x_2{:}\bar{R}_2^v{\wedge}\ldots{\wedge}x_n{:}\bar{R}_n^v \qquad (x_1 < x_2 < \cdots < x_n) \\
\bar{S}_r & ::= & \epsilon & \mid \quad \bar{P}^v{\to}\bar{R}^v
\end{array}
$$

The type equivalences given in table 4.2 allow us to find a canonical form for any component category. From now on, we assume that component categories are in canonical form.

The second property of categories along with the possibility to write the provided and required parts of a component category in a canonical form allow us to translate provided to required parts and vice versa. The *inverse* required part of a provided part $\bar{P}$ is designated by $\bar{P}^{-1}$. The straight forward translation is given in table 4.7. Note that we may not directly invert any expression of the form $\bar{P}{\cdot}\bar{S}$, but we can invert the corresponding canonical form.

As an example, consider the following component category, defining a category of logger components:

$$\text{log:}(\alpha^- \to \epsilon) \quad \to \quad \text{fileOpen:}(\beta^+ \to (\text{write:}(\alpha^+ \to \epsilon) \wedge \text{close:}(\epsilon \to \epsilon))$$

The expression states that a component of that category provides a *log* service that takes arguments of a unspecified type. The component requires its environment to provide an *openFile* service also taking an argument of an unspecified type. The service must return a component providing both a *write* and a *close* service. Furthermore, the *write* service must take arguments of the same type as the *log* service.

4.2.2 Membership Checking

Assume a component implementation given as a form expression F, and a component category $\bar{R} \to \bar{P}$. The goal is to decide whether the component is a member of the category or not.

Given the type judgement of the component implementation $\vdash F :: P \mid C$, the judgement for the component itself can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash e :: \alpha^+ \mid \rho \hookrightarrow e{:}\alpha^- \qquad \vdash F :: P \mid C}{\vdash e;F :: P[\rho \mapsto \alpha^+] \mid \rho \hookrightarrow e{:}\alpha^-, C[\rho \mapsto \alpha^+]}}{\vdash \lambda e.(e;F) :: \beta^- \to P[\rho \mapsto \alpha^+] \mid \rho{\cdot}e{:}\beta^+ \hookrightarrow e{:}\alpha^-, C[\rho \mapsto \alpha^+]}}{\vdash \lambda e.(e;F) :: \beta^- \to P[\rho \mapsto \alpha^+] \mid \beta^+ \hookrightarrow \alpha^-, C[\rho \mapsto \alpha^+]}}{\vdash \lambda e.(e;F) :: \alpha^- \to P[\rho \mapsto \alpha^+] \mid C[\rho \mapsto \alpha^+]}$$

The type judgement shows that every component is a (provided) service. In order to belong to the category $\bar{R} \to \bar{P}$, the component has

- to provide at least everything specified by $\bar{P}$, and
- to require no more than specified by $\bar{R}$.

This situation is very similar to a service application, where the service needs to provide a result fulfilling certian requirements, and it may not require more from the argument that the argument provides. In this analogy, the provided type of the argument takes the place of $\bar{R}$, the maximal requirements allowed by the category, and the type of the result stands for $\bar{P}$, the minimal set of what must be provided by the component. Thus, the component category $\bar{R} \to \bar{P}$ takes the place of a required service type. We construct the required type $\bar{R}^{-1} \to \bar{P}^{-1}$, and in analogy to the *Apply* inference rule, we write down the following set of constraints:

$$C[\rho \mapsto \alpha^+], \alpha^- \to P[\rho \mapsto \alpha^+] \hookrightarrow \bar{R}^{-1} \to \bar{P}^{-1}$$

which can be simplified using the *Resolve service* constraint equivalence to the following form:

$$\frac{C, \sigma^+ \hookrightarrow R \qquad (R \neq \tau^-)}{C[\sigma^- \mapsto R]}\ \sigma^+ \text{ does not occur in } C \qquad \textit{Bind left}$$

$$\frac{C, S \hookrightarrow \sigma^- \qquad (P \neq \tau^+)}{C[\sigma^+ \mapsto S]}\ \sigma^- \text{ does not occur in } C \qquad \textit{Bind right}$$

$$\frac{C, \sigma^+ \hookrightarrow \tau^-}{C[\sigma^+ \mapsto \tau^+][\sigma^- \mapsto \tau^-]}\ \textit{Rename}$$

Table 4.8: Constraint resolution rules

$$C[\rho \mapsto \alpha^+], \bar{R}^{-1} \hookrightarrow \alpha^-, P[\rho \mapsto \alpha^+] \hookrightarrow \bar{P}^{-1}$$

Applying the same considerations that led to the *Bind right* type inference rule, the constraint $\bar{R}^{-1} \hookrightarrow \alpha^-$ can be removed by substituting α^+ with $\bar{R}^{-1}$, leading to the following constraints:

$$C[\rho \mapsto \bar{R}^{-1}], P[\rho \mapsto \bar{R}^{-1}] \hookrightarrow \bar{P}^{-1}$$

The above set of constraints can be interpreted as follows: The context type variable ρ is substituted by the *provided* type $\bar{R}^{-1}$ representing the maximal environment allowed for the component. The last constraint expresses that the component's provided type must satisfy the minimal *required* type $\bar{P}^{-1}$. The component is considered member of the category, if the set of constraints can be completely resolved.

The above considerations lead to the following *membership checking algorithm*:

To check whether a component with the implementation F is member of a given component category $\bar{R} \rightarrow \bar{P}$, written as $F \in \bar{R} \rightarrow \bar{P}$, apply the following algorithm:

1. Infer the contractual type $P \mid C$ of F using the type checking algorithm presented in section 4.1.3.
2. Construct the following constraints:
$$C[\rho \mapsto \bar{R}^{-1}], P[\rho \mapsto \bar{R}^{-1}] \hookrightarrow \bar{P}^{-1}$$
3. Simplify type constraints using the equivalences in table 4.5 to constraints of the form $\alpha^+ \hookrightarrow R$, $P \hookrightarrow \alpha^-$, or $\rho \hookrightarrow R$.
4. Merge constraints over the same positive type variable or the context variable using the *Join* equivalences.
5. Resolve constraints by substituting type variables with type expressions using the constraint resolution rules *Bind left*, *Bind right*, and *Rename* as presented in table 4.8.

6. Iterate over points 3 to 5 until all constraints are resolved ($\top$), a type error is detected ($\bot$), or no further substitutions are possible.

The component with the implementation F is member of the category $\bar{R}{\rightarrow}\bar{P}$ if, and only if the above algorithm can resolve all constraints to $\top$.

As a first, simple example, we define the following component category:

$$y{:}\gamma^- \quad \rightarrow \quad x{:}\gamma^+$$

and we contend that the component having the implementation $y{\cdot}x{\mapsto}y$ is a member of that category. This claim is verified by applying the algorithm presented above. The contractual type of the implementation has already been deduced in section 4.1.4:

$$\vdash y{\cdot}(x{\mapsto}y) :: \beta^+{\cdot}x{:}\alpha^+ \mid \rho \hookrightarrow y{:}(\beta^- \wedge \alpha^-)$$

Then, the constraints are constructed and simplified. The constraints can be reduced to $\top$ by applying the *Resolve label* equivalence twice, then the *Split* equivalence once and finally the *Rename* resolution rule thrice:

$$\dfrac{\dfrac{\dfrac{y{:}\gamma^+ \hookrightarrow y{:}(\beta^- \wedge \alpha^-), \beta^+{\cdot}x{:}\alpha^+ \hookrightarrow x{:}\gamma^-}{\gamma^+ \hookrightarrow (\beta^- \wedge \alpha^-), \alpha^+ \hookrightarrow \gamma^-}}{\gamma^+ \hookrightarrow \beta^-, \gamma^+ \hookrightarrow \gamma^-, \alpha^+ \hookrightarrow \gamma^-}}{\top}$$

As the algorithm reduces the constraints to $\top$, the component is a member of the category.

4.2.3 Sub-Categories

In order to define a partial order on component categories, we introduce the notion of *sub-categories*. We demand that a sub-category of a component category $\bar{R}{\rightarrow}\bar{P}$ may not have

- less specific requirements for the provided type of a member component than $\bar{P}$, and
- more allowances for the required type of a member component than $\bar{R}$.

In other words, a sub-category typically has more requirements for the provided type and less allowances for the required type. Thus, the sub-category describes a member component more precisely than one of its super-categories.

Formally, the sub-category relation is defined as follows:

The component category $\bar{R}_1{\rightarrow}\bar{P}_1$ is a *sub-category* of $\bar{R}_2{\rightarrow}\bar{P}_2$, written as

$$\bar{R}_1{\rightarrow}\bar{P}_1 \subset \bar{R}_2{\rightarrow}\bar{P}_2$$

if, and only if the following set of constraints can be completely resolved:

$$\bar{P}_1 \hookrightarrow \bar{P}_2^{-1}, \bar{R}_2^{-1} \hookrightarrow \bar{R}_1$$

Now, we will argue that a component F is member of the category $\bar{R}_2 \rightarrow \bar{P}_2$, if it is a member of the sub-category $\bar{R}_1 \rightarrow \bar{P}_1$. We assume that F is a member of the sub-category. We know that the following constraints will resolve:

$$C[\rho \mapsto \bar{R}_2^{-1}], P[\rho \mapsto \bar{R}_2^{-1}] \hookrightarrow \bar{P}_2^{-1}$$

Knowing that the constraints $\bar{P}_1 \hookrightarrow \bar{P}_2^{-1}, \bar{R}_2^{-1} \hookrightarrow \bar{R}_1$ will resolve, too, it can be concluded that the constraints

$$C[\rho \mapsto \bar{R}_1^{-1}], P[\rho \mapsto \bar{R}_1^{-1}] \hookrightarrow \bar{P}_1^{-1}$$

will also resolve, showing that the component is a member of the category $\bar{R}_2 \rightarrow \bar{P}_2$, too. This conclusion can be proved recursively over the definition of categories.

As an example, the following component category is defined:

$$\epsilon \quad \rightarrow \quad x{:}\alpha^+ \cdot z{:}(\beta^- \rightarrow \beta^+)$$

We claim that this is a sub-category of the category $y{:}\gamma^- \quad \rightarrow \quad x{:}\gamma^+$, introduced as example in the previous section. The set of constraints is constructed and reduced to the empty constraint $\top$:

$$\dfrac{\dfrac{\dfrac{x{:}\alpha^+ \cdot z{:}(\beta^- \rightarrow \beta^+) \hookrightarrow x{:}\gamma^-, y{:}\gamma^+ \hookrightarrow \epsilon}{x{:}\alpha^+ \cdot z{:}(\beta^- \rightarrow \beta^+) \hookrightarrow x{:}\gamma^-}}{x{:}\alpha^+ \hookrightarrow x{:}\gamma^-}}{\top}$$

The second constraint can be removed immediately using the *Trivial* equivalence. The first constraint is simplified by appying the *Resolve label* constraint equivalence and removed by the *Rename* resolution rule.

4.3 Discussion

4.3.1 Contractual Types

Contractual types seem to offer a promising type system for form calculi. The version presented here is based on the original type system [Nie03], but has some improvements:

- The system becomes more expressive by allowing us to type the explicit environment.

- The type expressions are more uniform, as the requirements on the environment are also expressed as constraints, unlike in the original version.
- The type inference rules are more uniform, as they only rely on the subsitution operation. The original system relied on two additional operations to explain the *Service* inference rule.

The experimental implementation of the contractual type system for the JPiccola engine has shown that meaningful types can be derived for many Piccola expressions.

The problem with typing recursively defined services, as mentioned in [Nie03], remains in this version of the contractual type system. Also, Piccola expressions leading to types with large, complex constraints indicate that the type checking algorithm needs some improvement.

4.3.2 Component Categories

The introduction of component categories as a subset of contractual types brings many benefits. The most important are the following:

- Required services are more naturally expressed in the required of a category than in the constraints of a type.
- Categories allow to introduce a sub-type relationship to components.
- Component categories are designed to formalize architectural styles in a natural way, while allowing to verify implementations in the form calculus against such formalizations.

The concept of component categories will be validated in the next chapter by formalizing an architectural style.

Chapter 5

Compositional Styles

In this chapter, we put the pieces together. The introduced *component categories* allow a formal specification of architectural or *compositional styles*. The *form calculus* can be used to define formal definitions for connectors. The composition language *Piccola* provides mechanisms to embed external components as forms. *Contractual types* can be used to type such embedded components and thus verify them against the requirements of a compositional style.

5.1 Defining Compositional Styles

We introduce *compositional styles* as a formalization of architectural styles. A composition style defines

- *component sorts*, encoded as component categories,
- *connector descriptions*, also formulated as component categories, and
- *connector implementations*, encoded as form expressions.

In this section, the formal definition of a compositional style is illustrated on the familiar example of the pipes and filters style.

5.1.1 Component Sorts

A component sort specifies the services a component *provides* and *requires*. This can be formulated as a component category. The component sorts of the pipes and filters style are defined as follows:

$$
\begin{array}{lcrcl}
\mathit{Pipe} & = & & \epsilon & \rightarrow \epsilon \\
\mathit{Source} & = & \text{send:}(\alpha^{+}\rightarrow\epsilon)\wedge\text{close:}(\epsilon\rightarrow\epsilon) & \rightarrow & \epsilon \\
\mathit{Sink} & = & & \epsilon & \rightarrow \text{send:}(\alpha^{-}\rightarrow\epsilon)\cdot\text{close:}(\epsilon\rightarrow\epsilon) \\
\mathit{Filter} & = & \text{send:}(\alpha^{-}\rightarrow\epsilon)\wedge\text{close:}(\epsilon\rightarrow\epsilon) & \rightarrow & \text{send:}(\alpha^{+}\rightarrow\epsilon)\cdot\text{close:}(\epsilon\rightarrow\epsilon)
\end{array}
$$

A *Sink* provides a *send* service consuming data of an unspecified type α^+. The provided *close* service notifies the sink of the end of the data flow. No services are required by a sink. A *Source* requires both the *send* and *close* services, but provides no services. A *Filter* both provides and requires the two services, as it acts both as source and sink. A *Pipe* represents a closed stream, neither providing nor requiring services.

When checking a component against a sorts, the required and provided parts of a category are used separately. In order to faciliate notation, we introduce a notation for named sorts like *Filter*. The provided part will be written as $Filter^+$, the required part as $Filter^-$:

$$Filter = Filter^- \rightarrow Filter^+$$

Having defined the sorts of the style, we can check whether component implementations adhere to a sort by using the category membership checking algorithm. As an example, the empty form ε is checked against the source and sink sorts. Having the contractual type ϵ, we can see immediately that the empty form is an implementation of a *Source* component

$$\frac{\dfrac{\epsilon \hookrightarrow (Source^+)^{-1}}{\epsilon \hookrightarrow \epsilon}}{\top}$$

but not of a *Sink*

$$\frac{\dfrac{\epsilon \hookrightarrow (Sink^+)^{-1}}{\epsilon \hookrightarrow \text{send:}(\alpha^+ \rightarrow \epsilon) \wedge \text{close:}(\epsilon \rightarrow \epsilon)}}{\bot}$$

Note that the right part of the above constraint is the *inverse* of the provided type of the *Sink* category.

5.1.2 Connector Descriptions

Connectors are high-level abstraction that *compose* components, yielding a *composite* component. In the form calculus, connectors are thus represented as curried services that take at least two components as arguments and yield another component. For example, the contractual type of a binary connector has the following *signature*:

$$(P_1 \rightarrow R_1) \rightarrow (P_2 \rightarrow R_2) \rightarrow (R_3 \rightarrow P_3)$$

where $P_1 \rightarrow R_1$ and $P_2 \rightarrow R_2$ are the *required* types of the components to be composed, and $P_3 \rightarrow R_3$ is the *provided* type of the resulting component.

Having defined the component sorts for the pipes and filters style, the connectors of the style (see table 1.1) can be formalized by defining the following signatures:

$$
\begin{array}{rcl}
pipe & = & (Source^{-1} \rightarrow Sink^{-1}) \rightarrow Pipe \\
pipe & = & (Source^{-1} \rightarrow Filter^{-1}) \rightarrow Source \\
pipe & = & (Filter^{-1} \rightarrow Filter^{-1}) \rightarrow Filter \\
pipe & = & (Filter^{-1} \rightarrow Sink^{-1}) \rightarrow Sink \\
seq & = & (Source^{-1} \rightarrow Source^{-1}) \rightarrow Source \\
par & = & (Source^{-1} \rightarrow Source^{-1}) \rightarrow Source \\
fork & = & (Sink^{-1} \rightarrow Sink^{-1}) \rightarrow Sink
\end{array}
$$

Note that the *inverse* of the categories are used to describe the component sorts *required* by the connectors. It is also remarkable that connector signatures are syntactically component categories.

5.1.3 Connector Implementations

A compositional style is not only required to provide the signature for the connectors (that is, their syntax), but also their semantics. We use the form calculus to express the semantics formally.

As an example, we try to find an implementation for the pipe connector. This connector composes a component requiring the *send* and *close* services with another providing these. The resulting component should require whatever the right-hand R component requires and provide what the left-hand L component provides. A a consequence, the resulting component has to

- provide the right-hand component with a yet unknown environment e, and
- pass the provided services of the second component to the first.

This can be formalized with the following form expression:

$$\text{pipe} = \lambda L.\lambda R.\lambda e.L\ (R\ e)$$

In order to verify the proposed implementation, we first infer the contractual type of the expression, which is:

$$(\alpha^{+} \rightarrow \beta^{-}) \rightarrow (\gamma^{+} \rightarrow \alpha^{-}) \rightarrow (\gamma^{-} \rightarrow \beta^{+})$$

It is obvious that the proposed implementation is indeed a binary connector. Second, we have to make sure that the *pipe* connector adheres to the signatures defined in the previous section. By matching the connector's type with a signature, a mapping of the type variables to the provided and required parts of the component sorts can be found, as illustrated for the first signature of the pipes and filters style. Writing the signature in the extended form

$$((Source^{-})^{-1} \rightarrow (Source^{+})^{-1}) \rightarrow ((Sink^{-})^{-1} \rightarrow (Sink^{+})^{-1}) \rightarrow (Pipe^{-} \rightarrow Pipe^{+})$$

we can immediately determine the following mappings:

$$\begin{array}{ll} \alpha^+ \mapsto (Source^-)^{-1} & \alpha^- \mapsto (Sink^+)^{-1} \\ \beta^+ \mapsto Pipe^+ & \beta^- \mapsto (Source^+)^{-1} \\ \gamma^+ \mapsto (Sink^-)^{-1} & \gamma^- \mapsto Pipe^- \end{array}$$

The meaning of the mappings for the type variable α for example are that a sink must provide the services that a source requires. The mappings define a substitution that will transform the connector's contractual type to the investigated signature. To verify the consistency of the signature, we have to check the trivial constraints

$$\alpha^- \hookrightarrow \alpha^+, \beta^- \hookrightarrow \beta^+, \gamma^- \hookrightarrow \gamma^+$$

which will become non-trivial after substitution. For the example signature, the following set of constraints has to be checked:

$$\frac{(Sink^+)^{-1} \hookrightarrow (Source^-)^{-1}, (Source^+)^{-1} \hookrightarrow Pipe^+, Pipe^- \hookrightarrow (Sink^-)^{-1}}{\frac{\text{send:}(\alpha^+ \to \epsilon) \wedge \text{close:}(\epsilon \to \epsilon) \hookrightarrow \text{send:}(\alpha^- \to \epsilon) \cdot \text{close:}(\epsilon \to \epsilon), \epsilon \hookrightarrow \epsilon, \epsilon \hookrightarrow \epsilon}{\top}}$$

Having replaced the sorts placeholders with their definitions, the constraints can be entirely resolved. Thus we can conclude that the given implementation of the *pipe* connector adheres to the first signature. The other three signatures of this connector can be verified analogously.

5.2 Styles in Piccola

As a 'real world' example, we want to implement the pipes and filters style in JPiccola. The Java classes of the `java.io` package are the external components that will be adapted to the style. The pipe connector and sample filter components will be implemented in Piccola. External and native components can then be composed to a sample application.

5.2.1 Wrapping External Components

In order to adapt Java objects of the `java.io` classes, we define *generic* wrapper services. These services can be registered with the wrapping facility of JPiccola, ensuring that all objects are automatically adapted to component of the pipes and filters style.

The implementation of the service wrapping `java.io.Writer` objects as *sinks* is straight forward:

```
wrapWriter aWriter:
  aWriter
  \:
    send = aWriter.write
    close = aWriter.close

Host.registerWrapper "java.io.Writer" wrapWriter
```

An object is wrapped by extending it with a service taking no arguments and returning bindings for the `send` and `close` services. The `close` service is mapped to the Java method

```
    void java.io.Writer.close();
```

and the `send` service is mapped to the overloaded Java methods

```
    void java.io.Writer.write(char[]);
    void java.io.Writer.write(char[], int, int);
    void java.io.Writer.write(int);
    void java.io.Writer.write(java.lang.String);
    void java.io.Writer.write(java.lang.String, int, int);
```

Knowing this, we can conclude that the contractual type such a wrapped object will have the form

$$P{\cdot}(\epsilon \quad \rightarrow \quad \text{send}{:}(\alpha^{-}{\rightarrow}\epsilon){\cdot}\text{close}{:}(\epsilon{\rightarrow}\epsilon))$$

where α^{-} represents the different possible arguments for the `write` method. Using the membership checking algorithm, we can prove that such wrapped objects are components of the *Sink* category.

The wrapper service for `java.io.Reader` objects is somewhat more complicated, as the implementation of a *Source* component has to produce the data.

```
wrapReader aReader:
  aReader
  \env:
    'def loop: if aReader.ready()
      then:
        'ch = aReader.read()            # (3)
        if (ch != -1)
          then: (env.send ch, loop())   # (1)
    ''loop()
    ''env.close()                       # (2)

Host.registerWrapper "java.io.Reader" wrapReader
```

Again, the object (`aReader`) is extended with a service representing the component. The component now requires the `send` (1) and `close` (2) services from

its environment `env`. When instantiated, the component will get data from the `Reader` object and push it along the stream using the `send` service as long as data is available. Finally, the *Source* will signal the end of the stream by invoking the `close` service.

The contractual type of a wrapped `Reader` object is determined as follows: The quoted expressions (`'`) ensure that the empty form is returned from the component service. The required `close` service has no requirements for its parameter nor its return value, thus being of the type $\epsilon \rightarrow \epsilon$. The `send` service has no requirements to its return value, but takes an argument of an unspecified type α^+. The `if` service is provided by the standard JPiccola environment and thus accounted for. We can conclude that the type of an object wrapped by the `wrapReader` service will have the form

$$P \cdot (\text{send}{:}(\alpha^+ \rightarrow \epsilon) \wedge \text{close}{:}(\epsilon \rightarrow \epsilon) \quad \rightarrow \quad \epsilon)$$

and can be proven to be a component of the *Source* category.

Note that α^+ representing the type of the elements sent along the stream can be pinned down to represent a Java `Integer` object by looking at the line (3) in the above code. Thus, when a `Writer` object is attached as a *Sink* to a `Reader` source, the following of the overloaded `write` methods will be selected:

```
void java.io.Writer.write(int);
```

Generally, mismatches of the type of data sent along the stream can be detected by substituting the type of the produced and consumed data in the constraint $\alpha^- \hookrightarrow \alpha^+$.

5.2.2 Implementing Connectors

Implementing the connectors of a style in Piccola is straight forward. As the style provides definitions of the connectors in terms of form expressions, they can be directly translated to Piccola. To define the *pipe* connector as global operator `|` in Piccola, its implementation has to be bound to the default operator label `_|_default` in the `DefaultOp` form:

```
DefaultOp =
  DefaultOp
  _|_default = \L R e: L (R e)
```

5.2.3 Sample Filter Components

Components can be implemented directly in Piccola. As an example, consider a filter component that counts the number of elements pushed along the stream. The `countFilter` component is an abstraction over the environment returning a new instance of the component on each invocation.

A component instance is created as follows: The native Piccola service `newCounter` is used to get a counter for keeping track of the number of elements. A `getCount` service is provided for reading the counter. The two provided services `send` and `close` let the component adhere to the *Filter* sorts.

```
countFilter env:
  'aCounter = newCounter 0
  getCount = aCounter.get
  send el:
    'aCounter.inc()
    'env.send el
  close:
    'println "# of Elements: " + aCounter.get()
    'env.close()
```

The `send` service increments the counter and sends the element further downstream using the `send` service provided by the environment. The `close` service prints the total number of elements to the console, and forwards the end of stream signal using the required `close` service.

5.2.4 Composing Components

Having implemented the wrappers, the connector and a native Piccola components, we can write a sample application by composing components:

```
StringReader = Host.class "java.io.StringReader"
FileWriter   = Host.class "java.io.FileWriter"

source = StringReader.new "Hello World"
sink   = FileWriter.new "hello.out"
filter = countFilter

# Compose pipe component
pipe   = source | filter | sink

# Instantiate and run pipe component
pipe()
```

The `source` component is a wrapped `java.io.StringReader` object that will produce the characters of the string `"Hello World"`. The `filter` is a native counting filter, the `sink` a wrapped `java.io.FileWriter` object that will write received characters to a file named `"hello.out"`.

The components are composed using the global pipe operator. First, the `source` and `filter` are connected to a composite source, which is composed with the `sink` to the `pipe` component. In order to start the data flow, the `pipe` component must be instantiated. When the `source` signals the end of stream, the counting `filter` component prints out the number of processed characters.

Chapter 6

Conclusion

This chapter concludes the thesis by summarizing what we have learned, comparing the approach to others, and giving an overview of yet open issues.

6.1 Lessons Learned

The aim of the research described in this thesis was to provide a formalization for architectural styles that is applicable in practice. The base for our investigation has been the concept of forms. Forms indeed seem to be a good foundation for modelling components and composition, for the following reasons:

- Provided services can naturally be represented by bindings.
- The fact that forms are always evaluated in an environment provides a straight forward way to model required services.
- Both a composition language (Piccola) and a type system (contractual types) are based on forms.

The last point is most important, since it allows us to bridge the gap between the theoretical side providing means to formalize styles, and the practical side allowing to integrate with 'real world' applications and components.

One one side, forms are used by the Piccola composition language, that allows us to integrate external components written in mainstream languages (currently SmallTalk and Java). This guarantees that our approach stays in touch with practice. Also, special features of Piccola allow to implement *generic* wrappers and connectors.

On the other side, the contractual type system for forms allows us to formally reason about provided and required plugs. The enhancements we proposed to the type system allows us to type most Piccola expressions. Furthermore, the introduced notion of component categories allows us to define requirements for component sorts formally, and even check Piccola scripts and external components against the category specifications.

On the example of the pipes and filters style, we have shown, how an architectural style can be formalized as a *compositional style* using the contractual type system and component categories. We have also exemplified that the formalized style can be used in practice by composing Java components.

6.2 What's Next

There remain yet unresolved problems with the contractual type system: To be able to type all Piccola expressions, a typeable fixed point construct must be introduced in the form calculus used here. Also, the type system isn't yet able to type external components. We think that a promising approach would be to introduce the notion of external components and services in the form calculus itself. This would also help to integrate the Piccola bridging behaviour into the calculus itself.

Altough the Piccola language is powerful, its syntax and semantics are hard to grasp even by experienced programmers. To make our composition approach more practicable, it would be necessary to design a 'mainstream' composition languge with different syntax and probably some different semantic behaviour.

To validate our formalization of styles further, the mechanism will have to be applied to more complex general architectural styles as well as domain- specific styles. Finally, an unresolved issue remains in the component categories: Although the provided services of a component can be split up among different sorts (interfaces), it is not yet possible to do so for required services.

Bibliography

[AAG95] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. Cmu-cs-95-111, Carnegie Mellon University, January 1995.

[Ach02] Franz Achermann. *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Berne, January 2002.

[AKN00] Franz Achermann, Stefan Kneubuehl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Jav] Java compiler compiler. http://www.experimentalstuff.com/Technologies/JavaCC/.

[Meu98] Wolfgang De Meuter. Agora: The story of the simplest MOP in the world — or — the scheme of object–orientation. In J. Noble, I. Moore, and A. Taivalsaari, editors, *Prototype-based Programming*. Springer-Verlag, 1998.

[NA00] Oscar Nierstrasz and Franz Achermann. Separation of concerns through unification of concepts. In *ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, 2000.

[ND95] 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.

[Nie02] Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, Oct. 2002.

[Nie03] Oscar Nierstrasz. Contractual types. submitted for publication, 2003.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.

[Sch01] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[TT01] Satyam Tyagi and Paul Tarau. A most specific method finding algorithm for reflection based dynamic prolog-to-java interfaces. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 322–336. Springer-Verlag, 2001.