Supporting Pure Composition by Inter-language Bridging on the Meta-level

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Nathanael Schärli

September 2001

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Franz Achermann

Institut für Informatik und angewandte Mathematik

Abstract

Inter-language bridging is an important issue of scripting language design and implementation. Most of the popular languages such as Python, Perl, Tcl, and Ruby use a bridging approach based on wrappers that are written in the external language (usually C/C++) and serve as a glue layer between the languages. This allows a wide flexibility in defining the glue abstractions, but it requires the user to specify them on the level of the implementation language, and it therefore impairs the higher-level scripting process. In contrast, the first implementations of JPiccola, a scripting and composition language implemented in Java, use a generic bridging strategy based on information provided by Java's runtime introspection facilities. This strategy makes accessing of external objects more lightweight, but it does not provide the necessary means of abstraction and leads to a very tight coupling between the two language levels.

In this thesis, we present a new bridging strategy for Piccola that combines the advantages of the two approaches. We minimize the bridging functionality that is hardcoded in the virtual machine by making it a meta-aspect of the language Piccola. This allows the programmer to use the unrestricted expressive power of the scripting language to specify the glue abstractions at a higher level and adapt them dynamically. As a second contribution, we present a lazy evaluation technique that significantly reduces the performance overhead introduced by the meta-level bridging layer. In order to apply this lazy evaluation technique to Piccola services in general, we develop a partial evaluation algorithm that separates the side effects of a service and turns the individual expressions into closures. Finally, we give an overview of SPiccola, a Squeak-based Piccola implementation with thread-aware debugging tools.

Acknowledgements

I'd like to thank all the people who were, directly or indirectly, involved in this work. Special thanks to Franz Achermann for introducing me to Piccola and supporting me throughout the work that finally resulted in this thesis. In particular, I'd like to thank him for the countless hours we were sitting together and discussed about the Piccola language and its implementations. It was so much fun!

Special thanks also to Stéphane Ducasse who motivated me to join the SCG and to learn Smalltalk. The interesting discussions about Smalltalk, other programming languages and meta-object protocols were the starting point for all my work in these fields. And of course, I especially want to thank him for introducing me to Squeak and initiating the contact to Alan Kay, which allowed me to spend two summers as an intern at Alan's research group in Los Angeles. The time I have gotten to spend there was invaluable in many aspects that go far beyond computer science, and I'd like to thank Alan and all the other members of the Squeak Central for giving me this great opportunity.

I'd also like to thank Oscar Nierstrasz, head of the SCG, for giving me the opportunity to work in his group, for all the fruitful discussions about Piccola, and for the careful reading of this thesis and the constructive comments that helped me to improve it.

Thanks also to all the other members of the SCG for their support and the great time I have gotten to spend together with them. I also want thank some of my fellow students during my studies in computer science, mathematics, and physics. I enjoyed it very much!

Last but not least, I would like to thank my family and friends, who supported me during all my life and allowed me to be at the point where I am now.

Thank you all!

Nathanael Schärli,

September 2001

Contents

Chapter 1

Introduction

Today, applications are more and more frequently specified and implemented as the composition of components [11]. These components may be built elsewhere and are written in different languages. It is the purpose of a scripting or composition language to provide the necessary glue to make these components cooperate and wire them together. Components are per definition elements of a component framework. They adhere to a particular component architecture or architectural style that defines the plugs, the connectors and the corresponding composition rules. Examples for compositional styles are Unix pipes and filters, C++ template composition, GUI composition, and GUI event composition [3][8].

Many scripting languages such as Python [20], Ruby [25], Tcl [26], and Perl [27] have a rich set of programming constructs and built-in features that facilitate composition of components according to predefined compositional (architectural) styles [1][2]. Piccola, in contrast, is a *small* and *general-purpose* composition language. It has a small syntax and a minimal set of features needed to specify different styles of software composition [1][2][6]. This allows us to specify our own styles that define a kind of component algebra [1][2] that is well suited for the given problem-domain. Instead of low-level wiring, this allows us to *plug* components together on a higher level [1][2][3][5]. In addition, Piccola is a *pure* composition language. This means there is only a minimal set of primitives providing the necessary composition abstractions. All the other features of the language are provided by exchangeable components, and there are no datatypes or values built into the language. Even simple arithmetic or string operations are performed by external components. These components are represented as forms, which are the only first-class values in Piccola.

Accessing and controlling external components is one of the crucial issues of scripting languages, and due to its design as a pure composition language, it is especially important for Piccola. Other languages such as Python, Ruby, Tcl, and Perl use a bridging approach based on glue code that has to be written in the external language (usually C/C++). This allows a wide flexibility in defining the glue abstractions, but it requires the user to specify them on the level of the implementation language. Therefore, this approach is not high-level and lightweight enough for a pure composition language like Piccola, where everything is represented as an external component. Instead, JPiccola 2, the Piccola implementation on top of Java [36], uses a generic bridging strategy based on information provided

by Java's runtime introspection facilities, which makes accessing of external objects more lightweight. When a programmer accesses a Java object, the inter-language bridge, which is part of the JPiccola virtual machine, automatically creates an according Piccola form that represents the Java object and provides access to its methods. Similarly, a Piccola forms gets converted into a Java object when passed as an argument to an external (Java) method.

In our work with JPiccola, we identified several limitations of this bridging strategy. The main problem is caused by a lack of abstraction between the two language levels. In fact, the inter-language bridge only makes the passed entities technically compatible to the object model of the other language, but it does not allow the programmer to specify how to adapt them in order to fit the needs of the application and to cooperate with the other components. This leads to a very tight coupling between the two languages, and dealing with external components basically becomes "Java programming within Piccola". Due to the different philosophies of the languages, this results in code that does not suit the Piccola paradigm. In addition, the resulting code is inherently Java dependent and cannot be used on other Piccola platforms.

This gets even more problematic since the Piccola forms that result from converting external objects, we call them *external forms*, cannot be modified without sacrificing their external identity. This means that a modified external form is not converted to the original object when it is passed back to the Java language. On the one hand, this guarantees that the modifications performed in Piccola do not get lost, but at the same time, it causes the resulting object to lack the Java characteristics of the original one. In particular, the resulting object has a different class and can therefore not be used like the original object. Since most of the external components used in Piccola are eventually passed back to the Java language, this circumstance prevents the Piccola user from adapting the interface and the glue associated with an external component.

In this thesis, we tackle this problem and develop a more appropriate and flexible design for an inter-language bridge of a composition language. Basically, our solution consists of two main concepts:

Separating the different aspects of external forms. There are two different aspects associated with an external form, namely its external identity (i.e. the relation to the associated component) and its Piccola interface together with the glue. We use a nested structure to separate these two aspects. This allows the programmer to adapt the Piccola interface and the glue of an external component without sacrificing its external identity.

Moving the variable part of the bridge into Piccola. The task of a flexible inter-language bridge can be divided into two parts. First, the bridge has to convert the entities to make them technically compatible to the foreign language model. And

second, it should adapt the entity in order to fit the needs of the application and to cooperate with the other components. Whereas the first part is generic, the second one is completely variable and should be easy for a programmer to influence and adapt. We achieve this by moving the second part of the inter-language bridge into Piccola's meta-level, which allows dynamic and high-level specification of the components, their interfaces and the associated glue.

We show that this solution supports Piccola's concept of being a general-purpose and pure composition language. It allows external components to be accessed through a level of indirection that can be entirely defined in Piccola and allows the user to modify the external forms in a natural way.

However, a price has to be paid for the indirection introduced by moving the variable part of the inter-language bridge into Piccola, and this price is performance loss. Because Piccola is a pure composition language, external components are used everywhere and each of these components has to pass the inter-language bridge. Profiling a typical Piccola scripts shows that there is an enormous amount of time spent in the variable part of the bridge. This part consists of ordinary Piccola code that *wraps* the external components and provides the necessary glue for the components to cooperate. Analyzing the usage of external components shows that an average component only uses a small amount of the code built by the wrappers. Therefore, we propose a solution based on lazy evaluation to overcome this performance bottleneck.

In particular, we use a partial evaluation technique to transform a Piccola script into a semantically equivalent script that allows effective lazy evaluation. Then, we represent the result of service applications as lazy forms that only evaluate the effectively needed expressions. Our partial evaluation strategy allows us to apply lazy evaluation for any Piccola service by using two main techniques:

Separating the side effects. A service is transformed into an equivalent service where side effects are separated from the referentially transparent part. When such a service is invoked, we only have to execute the side effects immediately and can return the referentially transparent part as a lazy form.

Turning expressions into closures. A service is transformed into a service with closed expressions. This means that these expressions do not contain free identifiers except the ones referring to the service arguments. This makes it possible to evaluate only the expressions associated to the effectively needed bindings of a lazy form.

It turns out that Piccola is well suited for partial evaluation because of its simple syntax and semantics based on forms. Form expressions exhibit the right kind of information to statically separate the side effects of services and resolve internal dependencies. Furthermore, there is a direct and natural mapping between the

syntactical form expressions and the actual forms, which provide lightweight introspection facilities. Piccola has also no built-in datatypes that would complicate reasoning. Nevertheless, we believe that the presented partial evaluation technique can also be applied to other languages.

The rest of this thesis is structured as follows: In Chapter 2, we give an overview of the language Piccola. We present its architecture and syntax, introduce forms, and illustrate the key features with a few examples. In Chapter 3, we present the implementation of JPiccola 2 and especially focus on the problems caused by its inter-language bridge. Chapter 4 shows how we can solve these problems by separating the different aspects of an external form and moving the variable part of the bridge into Piccola's meta-level. In Chapter 5, we analyze the performance penalty caused by this bridging strategy and present a partial evaluation technique that allows effective lazy evaluation of general Piccola services. Chapter 6 concludes the paper and Chapter 7 addresses related and future work. In Appendix A, we give an overview of SPiccola, a Piccola implementation based on Squeak [10] with thread-aware debugging features.

Chapter 2

The Piccola Language

In this chapter we give an overview of the language Piccola [1][2]. In Section 2.1, we present Piccola's concept and architecture. Section 2.2 focuses on the notion of forms, which are the only first class values in Piccola. In Section 2.3, we present Piccola's syntax, and in Section 2.4, we illustrate Piccola's key features with a few examples.

2.1 Piccola — A pure composition language

In this section, we first give an overview of Piccola's concept and design goals, and then we present its architecture.

2.1.1 What is Piccola?

Piccola is a scripting and composition language. That is, it is a language for composing software components that may be written in a separate implementation language. Piccola is designed to express how such components are configured, and to provide the connectors, coordination abstractions, and glue abstractions to wire them together on a higher level [1][2].

Most of the scripting and fourth-generation languages such as Perl or Python have a rich set of programming constructs and built-in features that facilitate composition of components according to a predefined compositional style [1]. Piccola, in contrast, is a *small*, *pure* and *general-purpose* composition language:

Small. Piccola has only a small syntax and provides a small set of primitives needed for specifying different styles of software composition [1][6]. The primitives facilitate inspection of forms, spawning parallel agents, and creation of communication channels (cf. Section 2.1.2).

Pure. Piccola is a pure composition language because there is only a small set of primitives providing the necessary composition abstractions. All the other features of the language are provided by exchangeable components. This means that even basic programming entities such as numbers and strings are represented by components that can be dynamically reconfigured.

General-purpose. Piccola is a general-purpose composition language because it supports composition of components corresponding to different compositional (architectural) styles. This means that Piccola allows us to specify our own styles that define a kind of component algebra [1][2][3][6]. The sorts of such an algebra are different kinds of components, each characterized by different plugs and sockets that represent required and provided services. The operators of the algebra are the connectors. Instead of using low-level wiring, this allows us to *plug* components together. Thus, a script is just an expression that composes components, where each subexpression is also a component [1][2][3][5].

2.1.2 Architecture – Forms, agents and channels

In order to achieve a simple framework for component composition and definition of compositional styles, Piccola uses the following primitives that unify various concepts [6]:

Forms embody structure. A form is an immutable set of bindings that associate labels to values (i.e. forms). They can be extended with additional bindings, which yields a new form. Forms are the only first class values and unify data-structures, services, keyword based arguments, namespaces, and interfaces.

Agents embody behavior. Agents are concurrent communicating entities whose behavior is specified by a script. Agents implement the connections between components, and they unify concurrency and composition.

Channels embody state. Channels are mailboxes that agents use to communicate. Channels unify synchronization and communication.

Unlike forms, agents and channels do not appear in the syntax of Piccola but they can be directly instantiated by means of the primitive services `run` and `newChannel`.

2.2 Everything is a form

In Piccola, forms are the only first class values, which means that every first-class entity is represented as a form [1][2][6]. This section covers the semantics of Piccola forms and presents how the different language aspects are modeled with forms. Finally, it compares Piccola forms to the traditional object-oriented approach with objects and classes.

2.2.1 Semantics of forms

A Piccola form is an immutable and unordered set of bindings that associate labels to values (i.e. forms). The empty form contains no bindings. The following five basic operations are defined on forms:

Polymorphic extension. Polymorphic extension `F, G` of a form `F` with a form `G` yields another form containing all the bindings of the form `G` and the bindings of the form `F` whose labels are not used within the form `G`. This means that the bindings of the form `G` override equally labeled bindings of the form `F` in the resulting form.

```
F =                  # A nested form with three bindings name, value and size
   name = "Foo"
   value = 15
   size =            # The label size is bound to the form (x = 10, y = 20)
      x = 10
      y = 20

G =                  # A nested form with two bindings value and size
   value = 7
   size = (x = 10, y = 28)

println (F, G)       # Prints: (name = Foo, value = 7, size = (x = 10, y = 28))
```

Projection. Projection allows us to retrieve the form that is bound to a certain label. This means that a projection `F.l` returns the form that is bound to the label `l` within the form `F`. If this form does not contain a binding labeled `l` the operation results in a runtime exception.

```
F =                  # A nested form with three bindings name, value and size
   name = "Foo"
   value = 15
   size =            # The label size is bound to the form (x = 10, y = 20)
      x = 10
      y = 20

println F.name       # Prints: Foo
println F.size       # Prints: (x = 10, y = 20)
println F.size.x     # Prints: 10
println F.abc        # Error! (F does not contain a binding labeled abc)
```

Application. Everything in Piccola is a form, and therefore, services (Piccola abstractions) are also represented as forms. The application `F G` invokes the service represented by the form `F` with the argument form `G` and yields the resulting form. Note that a form can represent a service and have bindings at the same time (cf. Section 2.2.2).

```
# The form F gets defined as a service taking an argument X
F X:                          # Alternative definition: F = \X: ...
   value = X
   predecessor = X - 1
   successor = X + 1

println (F 3)                 # Prints: (value = 3, predecessor = 2, successor = 4)
```

Restriction. Restriction allows the user to remove a binding labeled `l` from a form F. If this form does not contain a binding labeled `l` the operation results in an error.

```
F =                           # A form with two bindings labeled name and size
   name = "Foo"
   size = 15

# The service label returns an arbitrary first class label bound in the argument form.
# Here, it returns size because this is the only label in the argument form
labelSize = label(size = ())
G = labelSize.restrict F      # Form restriction

println F                     # Prints: (name = Foo, size = 15)
println G                     # Prints: (name = Foo)
```

Inspection. Inspection is used to find out whether a form contains bindings, represents a service or is the empty form. The primitive service `inspect` is Curried. As a first argument, it takes the form that gets inspected. The second argument contains three services. Depending on the structure of the inspected form, either of these services gets invoked. If a form contains bindings, inspection can be used to retrieve an arbitrary first class label that is available within the inspected form.

```
# Define the three services for the second argument of the inspect service
Cases =
   isEmpty: println "Form is empty"
   isService: println "Form is a service and has no bindings"
   isLabel L: println "Form with label " + L.name()

inspect () Cases              # Prints: Form is empty
inspect (\X: X) Cases         # Prints: Form is a service and has no bindings
inspect (a = 5) Cases         # Prints: Form with label a
```

Whereas the Piccola syntax provides structures for polymorphic extension, projection, and application, there are no syntactical structures for the rarely used operations for restriction and inspection, which can be performed using the primitive `inspect` or first class labels, respectively. Note that a combination of restriction and inspection allows iteration through the bindings of a form.

2.2.2 Forms as a unifying concept

Forms are the only first class values in Piccola. In the following, we show how forms are used to naturally represent the different language concepts:

Data-structures (objects). Piccola uses (nested) forms to define data-structures. These data-structures are basic objects that may consist of structure and behavior (services). The following example shows such a data-structure and illustrates how projection can be used to access individual elements.

```
person =                                        # A nested form
   name = (first = "Peter", last = "Brown")
   yearOfBirth = 1970
   city = "Los Angeles"
   getAge YearNow: YearNow - yearOfBirth        # A service (behavior)

println person.city                             # Prints: Los Angeles
println person.name.last                        # Prints: Brown
println (person.getAge 2001)                    # Prints: 31
```

Services (behavior). Services are abstractions over an arbitrary form. *Internal services* are defined by Piccola scripts, whereas *external services* are provided by external components. Both, internal and external services are represented as forms. This allows us to define higher order services. In the following example, we define a service calculating the absolute value of a number and show how to use it. Note that a form may represent a service and contain bindings at the same time.

```
abs X:              # The form abs represents an internal service taking an argument X
   if X >= 0
     then: X
     else: (-X)

result = abs (-3)                   # Invokes the service with the argument -3.
println result                      # Prints: 3

newAbs = (abs, name = "Peter")      # Extends the form abs with a binding
println newAbs(-5)                  # Prints: 5
println newAbs.name                 # Prints: Peter
```

Keyword-based arguments. The structure of forms permits the strictly monadic Piccola services to use keyword-based arguments with optional default values. The following example shows how we can define a service `myPrint` that takes arguments that are associated to the keywords `stream` and `value`. Note that the service provides a default value for the argument `stream`. The *quote* expressions (`'`) are used to modify the static namespace without affecting the form being constructed (cf. Sections 2.3 and 2.4.5). Note that the example requires the initial context to provide two bindings `anOutputStream` and `aFileStream`.

```
# Service taking an argument with keywords stream and value.
myPrint X:
   # Use anOutputStream as a default if the argument X does not specify a specific stream
   'stream = (stream = anOutputStream, X).stream
   stream.print X.value

myPrint (value = "Hello")                          # Use default stream
myPrint (stream = aFileStream, value = "World")    # Use aFileStream
```

Namespaces. Piccola supports both static and dynamic namespaces, which are modeled as (nested) forms. Since these namespaces are first-class values, one can implement various abstractions to support modules and packages. The keyword ***root*** refers to the static namespaces (root context) in which identifiers are looked up. The dynamic namespace (dynamic context) is a form with a special semantics that is bound to the label *dynamic* of the root context [7]. Refer to Sections 2.4.5 and 2.4.6 for examples about namespaces and scope.

Channels. New channels are created by the primitive service `newChannel`, which returns a form that gives access to a channel. This form consists of two services for sending respectively receiving. Consult Section 2.4.2 for an extended example that illustrates the use of channels and agents. Note that channels do not necessarily preserve the order of the sent forms.

```
ch = newChannel()
ch.send 1
ch.send 2
println ch                          # Prints: (send = (service), receive = (service))
println ch.receive()                # Prints: 1 or 2
println ch.receive()                # Prints: 2 or 1
```

Labels. Piccola has the notion of first class labels, which provide a non-syntactic alternative for form extension (`bind`), restriction (`restrict`), and projection (`project`). Furthermore, they can be used to find out whether a form contains a specific label (`exists`). First class labels may be created using the built-in service `label`, which returns an arbitrary first class label that is available in the argument form. In the following example, we create a first class label *color* by invoking the service `label` with the argument form `(color = ())` that contains only one binding.

```
label = label (color = ())      # Returns first class label color
form = label.bind("blue")       # Binds the label color to "blue" in the resulting form
println form                    # Prints: (color = blue)
val = label.project(form)       # Project on the label color of the argument form
println val                     # Prints: blue
form = label.restrict(form)     # Remove the label color from the argument
println form                    # Prints: ()
```

External components. Piccola represents external components (respectively their interfaces) as forms. Figure 2.1 shows an external Piccola form that represents the object 9. All the (public) methods of the object are mapped to the corresponding labels of the form. Thus, the external form can be considered an interface or a proxy for the associated object.

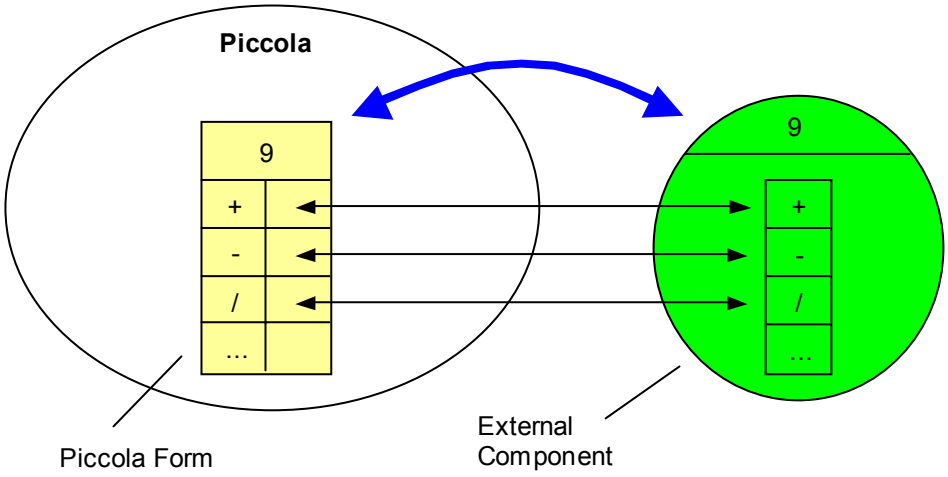

Figure 2.1: Form as interface to an external component

2.2.3 Forms versus objects

Today, most programming languages are designed according to the object-oriented paradigm with objects and classes as their basic entities. Every object is an instance of a certain class that defines its structure and its behavior. Usually, these classes are statically defined and they can inherit state and behavior from other classes.

In Piccola, forms play the role of objects. Forms are first class values and may contain both state and behavior. But compared to objects, forms are simpler data-structures and are not instances of classes describing their structure. As a consequence, dealing with forms is more lightweight:

No self. There is no reference to the active form (i.e. the form where the currently executed service is defined).

Immutability. Forms are extensible but immutable data structures and so there is no need for a copy-semantics.[1]

[1] In Piccola, forms are also used to represent entities with state, such as channels and external components. In these cases the immutable forms serve as interfaces to mutable entities.

Prototype-based instantiation. Forms may be built by adding new bindings to an existing form. There is no need to specify a corresponding class first. This approach is similar to that used by prototype-based object-oriented languages such as Self [12].

Dynamic extension. Polymorphic form extension can be used as a very primitive subclassing mechanism. As with traditional subclassing in object-oriented languages, the extended form (derived class) is compatible[1] to the original form (base class). This means that an extended form can play the role of the original one (cf. Section 3.4.1 for an example). Unlike traditional subclassing, which is performed on static classes, form extension is completely dynamic and directly applies to forms as runtime entities.

Whereas the traditional object-oriented paradigm is appropriate for creating component frameworks from scratch [11], we claim that the form approach is suitable as a unifying concept for high-level composition, because forms are more lightweight and dynamic:

- In the previous chapter we presented how forms are used to represent interfaces of external objects. The fact that forms can be dynamically extended facilitates dynamic adapting of these interfaces according to different needs and composition styles.
- Piccola uses forms to provide keyword-based arguments. This is possible because forms can be naturally created without having to specify a corresponding class first.

It is important to know that the rather primitive form concept of Piccola does not prevent the user from using more complex abstraction mechanisms [8]. As an example, Jean-Guy Schneider has developed an architectural style that conforms to the traditional OO programming paradigm [8].

[1] In Piccola, a value bound to a certain label *x* within a form *F* can be overridden with an arbitrary value when *F* is extended with another form *G*. As an example, we can override a label bound to a service and bind it to the empty form. Therefore compatibility of an extended form with the original form is not guaranteed. However, also in traditional object-oriented languages like Eiffel [37], compatibility between a base class and a derived class is not guaranteed. In Eiffel, methods of a base class can be removed or renamed in the derived class.

2.3 Language syntax

In this section, we give a brief overview of the Piccola language syntax. Instead of explaining all the available features, we give examples of the most important ones in Section 2.4. A complete description of the language syntax is given in Achermann's thesis [9].

Form ::=	
root	*static namespace*
identifier	*label*
literal	*constant literal*
\ [Param] : Form	*anonymous service*
Form . identifier	*projection*
Form Form	*application*
Form op Form	*infix application*
op Form	*prefix application*
Form , Form	*extension*
op⌊ [FormList] op⌋	*collection*
([Form])	*parenthesis*
root *= Form [, Form]*	*sandbox*
[***def*** *] Label [Param] : Form [, Form]*	*service binding*
[***def*** *] Label = Form [, Form]*	*binding*
' Form [, Form]	*quote*
FormList ::=	
[FormList ,] Form	*collection composition*
Param ::=	
identifier [Param]	
([identifier]) [Param]	
Label ::=	
[***root*** *.] identifier*	*simple label*
Label . identifier	*nested label*

Table 2.2: Piccola language syntax

The fact that Piccola uses only a few unifying concepts leads to a small syntax that is presented in Table 2.2. We use the keyword ***root*** to denote the static namespace where *identifiers* are looked up. *Literal* numbers and strings provide access to the associated external components. Piccola supports *infix* and *prefix operators* (*op*) that can be sequences of operator characters such as + and *. Bracket operators (*op⌊* and *⌋op*) may be used to specify user-defined collections, but they are not used within this

thesis. We use *sandbox* for specifying the static namespace that is used to evaluate the subsequent *Form*. As an alternative, we can use *quote*, which extends the static namespace instead of replacing it. In fact, the quote expression *'E, F* is syntactic sugar for the sandbox ***root*** = *(**root**, E), F*. Anonymous abstractions are specified by *anonymous service*, whereas *service binding* defines named abstractions. Finally, we use the keyword ***def*** in *binding* respectively *service binding* to define a recursive forms (fixed-point).

The reader may have noticed that we use the term *form* to denote both first-class runtime entities and syntactical expressions. In the following, we sometimes use the term *form expressions* to make it explicit that we mean syntactical forms.

2.4 Piccola by example

In this section we illustrate the most important Piccola features with examples. First, we introduce Piccola scripts, which are services taking the initial context as an argument. Then, we illustrate the use of agents and channels and show how nested bindings simplify modification of nested forms. Finally, we present some examples of namespaces and their scope.

2.4.1 Piccola scripts

Piccola programs are written as scripts that describe a sequence of form expressions. According to Piccola's architecture, these scripts are executed by an agent that provides the initial context (namespace). This context can be used and extended by the script. Therefore, a Piccola script can be considered as the body of a service definition that takes the initial context as an argument and makes it the static namespace by assigning it to ***root*** (*sandbox*). When executed, this service yields a new form that is the result of the script.

If `B` is the definition of a script, the corresponding service is:

```
\Root:                  # Anonymous abstraction taking an argument Root
  root = Root           # Use the argument form as the static namespace (sandbox)
  B
```

Here is an example script that defines a recursive service `fact`, invokes it with the argument `5` and assigns the result to the label `result`. The script uses the identifiers `fact`, `if`, `argument`, `result`, and `println` that are looked up in the static namespace. Since the bindings for `if` and `println` are not defined by the script, they have to be provided by the initial context.

```
def fact N:              # The keyword def is used to define recursive services (fixed-point)
  if N > 1
    then: N * (fact N - 1)
    else: 1

argument = 5
result = fact argument
println result           # Prints: 120
```

2.4.2 Agents and channels

Piccola provides the primitive services run and `newChannel` to spawn new agents and create channels for communication and synchronization. The service `run` spawns a new asynchronous agent executing the service that is bound to the label `do` of its argument. The service `newChannel` creates a new blocking communication channel that is represented by a form containing the services `send` and `receive`.

In the following example, we spawn a new agent that uses two channels `in` and `out` for communication. The agent reads a number from the channel `in`, increments it and sends the result to the channel `out`. We use this agent by sending a few numbers to the channel `in` and reading the results from `out`.

```
in = newChannel()              # Create new channel
out = newChannel()             # Create new channel
def incService:                # Define recursive service
  value = in.receive()         # Blocking read from the channel in
  out.send value + 1           # Send to the channel out
  incService()                 # Loop after sending the incremented value

run (do = incService)          # Spawn a new agent executing incService

in.send 5
println out.receive()          # Prints: 6
in.send 10
println out.receive()          # Prints: 11
```

2.4.3 Nested bindings

Nested bindings are syntactic sugar used to extend nested forms. In the following example, we show how they can be used to extend a nested form in a shorter and more natural way. The syntax of nested bindings is defined in Table 2.2.

In the following example, we create a nested form `person` and extend it using nested bindings. Then we show the clumsy alternative without nested bindings.

```
# Definition of a nested form
person =
   age = 23
   name =
      first = "Peter"
      last = "Brown"

# Using nested binding to extend the previously created nested form
person.name.middle = "Michael"

# Alternative without the use of nested bindings
person =
   person
   name =
      person.name
      middle = "Michael"
```

2.4.4 Operators

Piccola supports user-defined infix and prefix operators. An operator is a sequence of operator characters such as * and +. Infix and prefix operators can be defined using the special identifiers *_op_* respectively *op_*, which can be defined as ordinary services (*op* denotes an operator).

In the following example, we create a form `two` that contains infix and prefix operators. Then, we show how the operators can be used.

```
two =
   _+_(Left): 2 + Left        # Define infix operator +
   _-_(Left): 2 - Left        # Define infix operator -
   -_: -2                     # Define prefix operator -

println two + 100             # Prints: 102
println two - 1               # Prints: 1
println (-two)                # Prints: -2
```

Piccola also supports default operators, but we do not cover them in this thesis.

2.4.5 Static namespace and scope

In Piccola, the static namespace is an ordinary form that can be accessed with the keyword ***root***. The scopes of Piccola namespaces are local, and therefore, bindings defined in a subform do not affect the outer scope. Furthermore, Piccola provides hidden bindings (*quote*) that are added to the static namespace but not to the value that is being constructed.

In the following example, we first define a binding `val` that is added to both the current form and the static namespace. Then, we create a subform and bind it to the label `rectangle`. Note that the *quoted* (`'`) binding for `val` is added to the static namespace only and does not affect the form bound to `rectangle`. Since Piccola scopes are local, the redefinition of `val` is not visible outside this subform. Finally, we use *sandbox* to replace the static namespace by another form. Since `val` is not defined in this form, a reference to it results in an error. Note that overriding of the static namespace is a crucial operation that may have serious consequences. In our example, we make sure that the dynamic namespace (`dynamic`) and the service `println` are available in the new static namespace.

```
val = 100                  # Add binding to the static namespace and the current form

rectangle =
   'val = 3 * val          # Add binding to the static namespace only (quote)
   width = val - 10        # The identifier val refers to the local definition
   height = val + 10

println val                # Prints: 100
println rectangle          # Prints: (width = 290, height = 310)

root =                     # Explicitly replace the static namespace (sandbox)
   dynamic = dynamic       # Reuse the dynamic namespace in the new namespace
   println = println       # Reuse the service println in the new snamespace
   color = "blue"
   name = "Peter"

println color              # Prints: blue
println val                # Error!
```

2.4.6 Dynamic namespace

By default, all identifiers are looked up in the static namespace, which is an ordinary form that is accessible using the keyword ***root***. In addition, Piccola also provides a dynamic namespace. The dynamic namespace is represented by a form bound to the label `dynamic` of the static namespace. This form gets implicitly passed whenever a Piccola service is invoked [6].

The following example script defines a service `printState` that prints a string onto a stream defined in the dynamic context. Because this context is implicitly passed on every service invocation, the programmer can dynamically specify which stream to use. Note that the script requires the initial context to provide bindings an`OuputStream`, a`FileStream` and my`Print`.

```
# Use anOutputStream as the defaultStream in the dynamic context
dynamic.defaultStream = anOutputStream        # Nested binding

printState X:
   myPrint
      stream = dynamic.defaultStream
      value = "State of the script: " + X

printState "Hello"                            # Prints to anOutputStream

# Use errorStream as the defaultStream of the dynamic context
dynamic.defaultStream = aFileStream           # Nested binding
printState "World"                            # Prints to aFileStream
```

Chapter 3

JPiccola 2 and its inter-language bridge

The original implementation of Piccola is called JPiccola, and it is implemented on top of Java. This means that the parser and the virtual machine are implemented in Java, whereas other parts like a simple development environment and a small library are built in Piccola by using Java components. Since Piccola is designed as a composition language, using external components is a core principle of JPiccola, and it strongly influences its implementation.

In this chapter, we focus on JPiccola 2 and analyze its strategy for inter-language bridging. Section 3.1 gives an overview of JPiccola's virtual machine. In Section 3.2, we reason about inter-language bridging in general, and Section 3.3 presents the bridging strategy used by JPiccola. In Section 3.4, we show that this strategy is not flexible enough and causes serious incompatibilities with Piccola's goals and architecture.

3.1 Concept of JPiccola's virtual machine

JPiccola's virtual machine reflects the fact that Piccola is designed as a composition language. It consists of a special part called the *inter-language bridge* that facilitates accessing external components and their methods from within Piccola. Whereas most other virtual machine implementations provide many primitives to perform basic system operations such as integer arithmetic, the JPiccola virtual machine uses the inter-language bridge to delegate these operations to external components.

Apart from that, the JPiccola virtual machine has a structure similar to other virtual machines such as the one of Smalltalk-80 [13][14], and it consists of the following parts:

- Interpreter
- Runtime data structures (forms)
- Primitive services
- Inter-language bridge

Because there is no byte-code compiler for Piccola yet, the *interpreter* directly operates on the parse trees[1]. Interpretation of the parse tree nodes results in forms, which are the only first-class values in Piccola. Every form is represented by a Piccola *runtime data structure* that is an instance of a Java class providing the five basic form operations, namely polymorphic extension, projection, application, restriction, and inspection (cf. Section 2.2.1). Since everything in Piccola is a modeled as a form, the form data structures are also used to implement namespaces.

When a Piccola service is invoked, the interpreter usually responds by executing the Piccola code that is associated with the service. Some services, however, are realized by executing a virtual machine primitive. Similar to other virtual machines, the JPiccola virtual machine uses such *primitive services* to perform basic operations that cannot be performed or can only be performed inefficiently without a primitive. But, whereas other virtual machine implementations usually need lots of primitives for arithmetic operations, arrays and streams, input/output, storage management, and system operations, Piccola only needs the four primitives shown in Table 3.1.

Primitive	**Description**
`run`	Spawns a new asynchronic agent executing the service that is bound to the label do of its argument.
`newChannel`	Creates a blocking communication channel.
`inspect`	Inspection is used to find out whether a form contains bindings, represents a service or is the empty form. If the form contains bindings, inspection returns an arbitrary label that is used within the form.
`external`	Provides access to external components[2].

Table 3.1: The Piccola primitives

All the other basic operations are delegated to external components that are accessed through the *inter-language bridge* that is a core part of the Piccola virtual machine and allows passing of runtime entities across the language boundary in a bi-directional way.

---

[1] In the first versions of JPiccola, the Piccola code is first translated to the polyadic π-calculus, which serves as a semantic foundation for Piccola and allows formal reasoning about program behavior [4]. This additional language layer is not relevant in this context and is therefore neglected.

[2] Note that the primitive to access external components depends on the host language. Therefore, the name and the semantics of this service are implementation specific. In JPiccola2, this primitive is named `javaClass`.

3.2 Bridging between two nested language models

In this section, we analyze the situation of having two nested language models and the consequences for an inter-language bridge. Although we are starting with JPiccola and Java as the example, the analysis stays at an abstract level and is independent of particular programming languages.

By implementing the language Piccola on top of Java, we are dealing with two nested language models. On the one hand, we have the Piccola model with forms as its runtime entities. From a Piccola point of view, everything is a form and every form has the properties specified by the semantics in Section 2.2.1. On the other hand, there is the Java model, where everything is an object[1]. Since Piccola is running within the Java model, every Piccola form is actually a Java object. However, Java objects are incompatible with the form-based Piccola model and so they cannot be accessed within Piccola.

To abstract away from our concrete situation, we use the same terminology as in Agora [15]. The term *down level* refers to the implementation language (such as Java or Squeak), whereas *up level* means the language that is implemented and evaluated on top of the down level (Piccola). We assume that the down level provides some object-like first class entities and call them *objects*. At the same time, the first class entities of the up level are named *forms*. If we say that an object is passed *upwards*, we mean that the object is passed from the down level to the up level. Accordingly, we say a form is passed *downwards* if it is passed from the up level to the down level. Figure 3.2 gives an illustration of the two language models and the terminology.

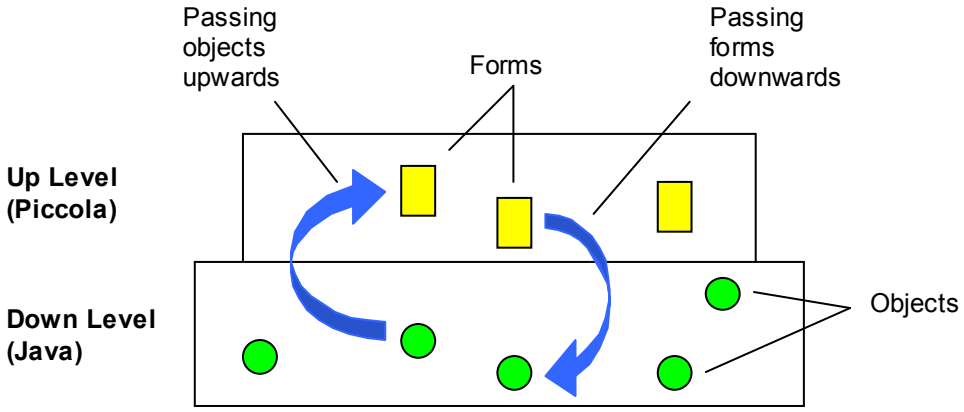

Figure 3.2: Nested language models (up level and down level)

[1] In fact, Java also has primitive data-structures such as `int` and `bool`, which are not objects. Since they can be converted into corresponding objects, this issue is neglected here, although it complicates the implementation of the Piccola virtual machine.

A strategy for bridging between two languages has to specify how runtime entities of either language are passed across the language boundary. In the case of nested language models, this is not symmetric, and we have to consider both directions:

Passing objects upwards

Since the language model of the up level cannot deal with generic objects, they have to be converted into forms in order to be accessible. This means that the inter-language bridge has to create an appropriate form representation of every object that is passed to the up level. If the passed object is already a form, the bridge does not have to do anything and can directly forward the form.

As a consequence, the up level knows two substantially different kinds of forms. The most obvious ones are the forms that are ex nihilo created within the up level, and we call them *plain forms*. This means that a plain form may be the empty form or a form that results from binding values (i.e. forms) to labels within Piccola. The other forms are the ones that are automatically created by the inter-language bridge whenever an object is passed upwards. They actually represent a down object within the up level, and we therefore name them *external forms*. Sometimes, we use the term *associated object* to refer to the object represented by an external form.

Passing forms downwards

From a down-level point of view, every form is a special object in its own language model. Therefore, a form can be passed to the down level just as it is. Although this makes sense for plain forms, it is usually not what we need in case of external forms that represent a down object. In this case, we often want the down level to operate on the object associated to the form rather than on the form itself. Deciding on which of the two possible entities has to be passed in a certain situation is one of the crucial points of an inter-language bridging strategy and as we will show in Section 4.5, the bridging layer does not have enough information to reasonably take this decision by itself.

3.3 JPiccola's bridging approach

After analyzing the strategy for inter-language bridging on an abstract level, we present the approach that is used by JPiccola and illustrate its key issues with two examples.

3.3.1 The bridging strategy

This section contains a description of what happens if an entity is passed across the language boundary in either direction.

Up. Passing objects from Java to Piccola

A. If the object is already a form (i.e. it is an instance of a form data structure), it is directly passed up to the Piccola language.

B. Otherwise, the following happens:

   B1. An object is converted into a form that contains a label for every public method that is implemented or inherited by the class of the object. Each of these labels is bound to a service that represents this method for the given object.[1] This means that this object is used as *self* when the service is invoked.

   B2. Forms representing special objects such as numbers, strings or booleans are extended with additional bindings that make them more appropriate from a Piccola point of view.

We say that a form represents an external (Java) object if it was created by step *Up.B* of the inter-language bridge. According to the terminology in the previous section, we call these forms *external forms*. Note that a form that is built by extending or restricting an external form is not considered an external form anymore.

Passing forms from Piccola to Java (Down)

A. If the form represents an external object (i.e. it is an external form) this object is passed down to Java.

B. Otherwise the form itself is passed down to Java.

[1] Since Piccola services have a different structure than Java methods, we use the labels `val`, `val1`, `val2`, etc. to specify the arguments of the corresponding method invocation. Similarly, we use the labels `type`, `type1`, `type2`, etc. to specify the type of the arguments, which may be important for selecting a particular overloaded method.

3.3.2 Examples

In the following we present two examples that illustrate the different aspects of JPiccola's bridging strategy. Whereas the first example just passes external objects up and down, the second one also passes plain forms across the language boundary. Note that the following examples and the other examples in this Chapter are written in JPiccola 2.

Example 1

In this example, we first use the primitive service `javaClass` to create a new instance of the Java class `FileWriter` that refers to the file named `"FileName"`. Then we create an instance of the class `PrintWriter` that is attached to the `FileWriter` instance and print `"Hello World"` on it. Note that we use the labels `val` and `type` in the argument of external services. This is necessary to specify the argument value and the argument type of the corresponding Java method.

```
1.  writer = javaClass("java.io.FileWriter").new
      val = "FileName"
2.  stream = javaClass("java.io.PrintWriter").new
      val = writer, type = "java.io.Writer"
3.  stream.println (val = "Hello World")
```

Execution of this script triggers the following bridging related virtual machine operations:

1. The operations triggered by line 1 can be separated as follows:
   a. `"java.io.Filewriter"`. The virtual machine creates the Java string object and the bridge converts it into the corresponding form when it is passed upwards *(Up.B1, Up.B2)*.
   b. `javaClass("java.io.FileWriter"). javaClass` is a primitive service referring to a Java method. Since the argument is an external form the bridge passes the associated string object downwards *(Down.A)*. The result of this operation is a form representing the Java class `FileWriter`.
   c. `writer = javaClass("java.io.FileWriter").new(val = "FileName").` The service new refers to the constructor of the class `FileWriter`. Because the argument value `"FileName"` is an external form it gets converted to the corresponding string object by the bridge *(Down.A)*. Finally, the newly created `FileWriter` instance is converted to a form when passed back to Piccola *(Up.B1)*.

2. The second line triggers similar bridging operations as the first line. The main difference is the call to the `new` service. We use a service of the class `PrintWriter` and pass the form `writer` created in the first line as an argument. In addition, we specify the constructor by defining the type of the argument. Both the argument (`writer`) and the type (`"java.io.Writer"`) are external forms and therefore the associated Java objects are passed downwards *(Down.A)*.
3. First, the string object `"Hello World"` is created and passed up to Piccola *(Up.B1, Up.B2)*. Then, the resulting form is used as an argument for the service `println` and is converted back to the string object by the inter-language bridge *(Down.B)*.

Example 2

In the second example, we use the primitive service `javaClass` to create a new instance of the class `Vector`. Then we create the string `"Hello World"` and add a binding labeled `length` to it. Finally, we append the resulting form to the `vector`, read it again and print it.

```
1.  vector = javaClass("java.util.Vector").new()
2.  originalForm =
      "Hello World"
      length = 11
3.  vector.add (val = originalForm)
4.  readForm = vector.firstElement()
5.  println readForm.length                    # Prints: 11
```

Most of the bridging operations have already been described in the previous example. Therefore, we mainly focus on line 3 and 4 where new operations are used:

1. First, the string object `"java.util.Vector"` is passed upwards *(Up.B1, Up.B2)* and then the corresponding form is passed downwards as the argument to the service `javaClass` again *(Down.A)*. Finally, the newly created vector is passed upwards and converted to a form *(Up.B1)*.
2. The literals `"Hello World"` and `11` are created by the virtual machine and passed upwards *(Up.B1, Up.B2)*.
3. The `add` service of the vector is invoked with the form `orginalForm` as the argument. Since the `orginalForm` is not an external form, the bridge passes the form object downwards as it is *(Down.B)*. This means that the form is not converted and the form object itself is directly added to the Java vector.
4. The invocation of the Java method `firstElement` yields the form object that was added in the previous line. Since this object is already a form, the bridge directly passes it up to Piccola *(Up.A)*.

5. The label `length` is bound to the external form representing 11 and therefore, the corresponding object is passed down to Java (*Down.B*) and printed.

Lines 3, 4, and 5 nicely illustrate that it is important that only unmodified forms are replaced by the associated object when passed downwards. Otherwise, the form `originalForm` would be converted to the string object `"Hello World"` when stored in the vector. As a consequence, the retrieved form `readForm` would not contain the binding `length` anymore. This would violate the usual semantics of the collection (vector), which says that elements are not modified when inserted and retrieved.

3.4 Limitations of JPiccola's bridging approach

As the previous examples show, JPiccola's bridging strategy basically fulfills its task and allows the user to access external components wrapped up as forms. In this section, we have a closer look at this strategy and point out why it is still not flexible enough. We explain why using the external forms as provided by the bridge causes many incompatibilities with Piccola's core concepts and show that the problems are coupled in a way that prevents a user from working around some of the problems without running into others.

3.4.1 The problems

In this section we show that there are many problems caused by JPiccola's bridging strategy. First, we explain why external forms do not behave like internal Piccola forms. Then, we illustrate that the bridge uses a very direct way to convert external components into Piccola forms. Amongst others, this leads to a very tight coupling of the two languages. Finally we claim that hardcoding the structures of special objects such as integer and strings in the Piccola virtual machine is not flexible enough.

I. Incoherent behavior of external forms

As a pure composition language, which uses external components even for basic operations, Piccola should make it possible to work with forms representing such components in the same way as with internal forms. This is particularly important for form extension. As explained in Section 2.2.3, form extension can be used as a simple but dynamic subclassing mechanism in Piccola. This means that we can extend a form with new bindings without sacrificing compatibility to the original form. The extended form can then play the role of the original in similar way as a subclass can play the role of its base class. Due to Piccola's semantics, this is automatically true for every form as long as it is used inside of Piccola.

The following example uses a service `printDate` that prints the date represented by the argument form. This service requires the argument to provide at least the bindings `month`, `day` and `year`. First, we invoke it with a form that represents a date and only contains the required bindings. Then, we extend the form with bindings specifying the time and show that is still compatible with the service.

```
printDate X:
  println X.month + "/" + X.day + "/" + X.year

date =                          # Defines a form representing a date
  month = "7"
  day = 2
  year = 2001
printDate date                  # Prints: "7/2/2001"

dateAndTime =                   # Defines a form representing a date and a time
  date
  hour = 10
  minute = 33
printDate dateAndTime           # Prints: "7/2/2001"
```

Unfortunately, this basic Piccola concept does not hold for forms that are passed to external services. As soon as an external form is extended it is not considered an external form anymore and it behaves totally different when passed down to the host language. An external form gets converted into the associated external object *(Down.A)* whereas any other form is not converted, and the form object itself is passed to the host language *(Down.B)*. Since the user has no means to determine whether a certain form is external, it is also not possible to find out whether extending this form will affect the way it is handled by the bridge.

The following example illustrates this problem. We first create a service `newButton`, which builds an external form representing a button and extend it with a service `setText`. Then, we call the service `newButton` to create a new button and use `setText` to set its label. Finally, we try to add the button to a Java panel, but unfortunately, Piccola does not behave as we expect. Because the argument `okButton` has been modified it is not considered an external form anymore, and the inter-language bridge passes the form object itself *(Down.B)* and not the associated button object down to the Java language *(Down.A)*.

```
# This service creates a new button and extends it with an additional interface binding
newButton:
   'button = javaClass("java.awt.Button").new()
   button
   setText(S): button.setLabel(val = "Piccola-Button: " + S)

okButton = newButton()
okButton.setText("Ok")          # Uses interface service to set the label
# XPiccola.piccolaPanel.buttons is a Java panel of the Piccola user-interface
panel = XPiccola.piccolaPanel.buttons
# Attention! The whole Piccola form and not only the button object is passed down to Java
panel.add(val = okButton)
```

II. Direct mapping conflicts with Piccola's principles

JPiccola's inter-language bridge uses a very direct way to convert an external component into a Piccola form. In fact, the method interface of the object gets entirely mapped to the resulting form. As a consequence, Piccola operates basically on the level of the host language, which leads to the following problems:

No separation between the language levels. Most of the components used by JPiccola are Java objects. Because of the direct bridging strategy, dealing with these components basically becomes „Java programming within Piccola" and due to the different philosophies of the languages, this results in code that does not suit the Piccola paradigm. In addition, the resulting code is inherently Java dependent and cannot be used on other Piccola hosts. This gets especially obvious with service invocations. Whereas Java is a typed language with tuple based method arguments, Piccola does not have types and uses a concept of monadic higher order services. Therefore, invocation of Java methods from Piccola looks very unnatural and clumsy. See Section 3.3.2 for examples.

External forms are cluttered with inappropriate lower level services. Java objects often provide rich interfaces and therefore the corresponding Piccola forms contain many bindings. Besides the fact that this makes them very complex, it also contradicts Piccola's philosophy. Within Piccola, Java objects are viewed as components with a limited set of services used to plug them together according to a specific compositional style. Therefore, most of a Java object's public methods should not be visible on the composition level.

Hard to use components with incompatible interfaces. The lack of abstraction for accessing components prevents the programmer from using components with incompatible interfaces (plugs). As an example we consider several components representing numerical structures (integers, floats, fractions, etc.) that were written independently and have different interfaces. Rather than directly using

these interfaces, the bridge should convert them according to the current style for composing numerical structures in Piccola.

III. Hardcoding structures in the virtual machine is inflexible

In order to make some of the frequently used Java objects look more natural in Piccola, the inter-language bridge adds some special bindings to the external forms representing these objects in step *Up.B2*. Although this helps to avoid some of the problems caused by the direct mapping, hardcoding the extension in the virtual machine is not a flexible solution. Piccola is supposed to be a general-purpose composition language and the programmer should be able to use it for many different problem domains with different requirements on the used components. Hardcoding the structure of the components in the virtual machine is completely static and cannot be modified without replacing the virtual machine. Therefore it contradicts Piccola's goal of being a flexible and general-purpose language.

This can be illustrated with number components. Most applications only use a very limited set of numerical operations. Therefore it is suitable to provide standard number components with a small, clear and easy to use interface. However, for other applications (e.g. mathematical encryption) it is necessary to have numeric components providing a rich set of operations. In this case, the standard number representation is not sufficient anymore, and the user should be able to easily modify it according to her needs.

3.4.2 The problems are coupled and hard to overcome

Unfortunately, the problems mentioned in the previous section are coupled in a way that makes it hard for a user to work around some of them without running into others.

A programmer can tackle the direct mapping problem (*II*) by manually adapting the structure of external forms and make them appropriate to Piccola's paradigm and the used compositional style. This adaptation includes changing method names and argument structure, cleaning up the interface by removing irrelevant methods and adding Piccola forms and services to provide more information and allow new operations. This makes the external forms look like internal ones, and as long as they are passed to internal Piccola services, everything works out. However, because of the incoherent behavior of external forms (*I*), a modified external form loses its external identity, and this completely changes the way it is treated when passed as an argument to an external service.

On the other hand, the incoherent behavior of external forms (*I*) can be avoided by keeping the original external forms as the bridge provides them. Like that, these forms are treated the right way when passed to the host language, but we cannot

handle them like ordinary Piccola forms and have therefore no means to avoid the direct mapping to the host language (*II*).

Hardcoding the structure of external forms in the virtual machine is an approach that avoids the problem of direct mapping without having to modify the external forms in Piccola. But as we mentioned above, it leads to a completely static behavior that cannot be modified or extended without replacing the virtual machine.

In the rest of this thesis we present a solution that solves the problems identified in this section and show how it can be implemented efficiently.

Chapter 4

Inter-language bridging as a meta-aspect of Piccola

After pointing out the limitations of JPiccola's original inter-language bridge in the previous chapter, we present a bridging strategy that is more suitable for Piccola. This strategy is based on two main concepts: Separating the different aspects of an external form and moving the variable part of the inter-language bridge onto Piccola's meta-level.

This chapter is structured as follows: Section 4.1 gives an overview of our solution, and Section 4.2 presents the nested structure of external forms in detail. In Section 4.3, we explain why moving the variable part of the bridge into Piccola leads to the required flexibility. Then, we present two possible models for the bridging framework inside Piccola, compare them and explain our decision. In Section 4.4, we give a detailed presentation of our preferred model for this bridging framework. Finally, we use Section 4.5 to introduce a simple service that allows the programmer to control the behavior of the bridge when a form is passed down to the host language.

4.1 Overview of our solution

In this section we present a new bridging strategy that solves the problems identified in the last chapter. The solution is based on two main concepts: First, we use a nested structure to separate the different aspects of an external form, namely its external identity and its higher-level interface (including glue). This makes external forms behave like all the other ones and allows the programmer to add and remove bindings without destroying their external identity. Second, we move the variable part of the inter-language bridge into Piccola. This allows dynamic configuration of the components, their interface and the associated glue directly within Piccola itself.

4.1.1 Terminology

As in the previous chapters, we use the term *external form* to denote a form that represents an external object within Piccola, and we use the term *plain form* for all the

other Piccola forms. In addition, we require that an external form has a nested structure that consists of two parts: The top level part represents the Piccola interface of the object and we therefore call it *interface form* or just *interface*. This form contains a label *peer* that is bound to a subform representing the identity of the external object, which is called *peer form* or just *peer*. It is important to understand that only forms corresponding to this structure are considered external forms. In particular, a form with a label *peer* that is not bound to a *peer form* (i.e. a form representing an external component) is not an external form.

In order to achieve the required flexibility, we separate the inter-language bridge into two parts: The *generic part* is located in Piccola's virtual machine, whereas the *variable part* is situated inside Piccola. When an external object is passed up to Piccola, both of these parts may build an interface for this object. Note that these interfaces usually include glue code. We use the term *generic interface* to denote the interface built by the generic part of the bridge and use the term *specific interface* for the interface built by the variable part. Accordingly, we use the terms *generic external form* and *specific external form* for the external forms created by the generic and the variable part of the bridge, respectively. Since Piccola uses external components even for very basic operations, the ability of specifying their interface (including glue code) in the inter-language bridge allows us to control and influence the behavior of the language. The variable part of the bridge allows the programmer to do this in the Piccola language itself, and we therefore say that it is located in *Piccola's meta-level*.

4.1.2 Illustration of the bridging strategy

Figure 4.1 shows the architecture of the inter-language bridge and illustrates how entities are passed across the language boundary. We see that the inter-language bridge is divided into two parts. The generic part is implemented in the virtual machine, which is part of the down level, whereas the variable part is located in Piccola's meta-level.

On the left side of Figure 4.1, we show what happens when an object not representing a form is passed upwards. In the generic part of the inter-language bridge, the object is converted into an external form consisting of a generic interface and the peer form that represents the identity of the object. Then, the external form is passed to the variable part of the inter-language bridge on Piccola's meta-level. Here, the generic interface gets replaced with a specific interface that can be specified by the programmer. The resulting external form consisting of the specific interface and the peer binding is then used within Piccola. Note that the variable part of the interface may not provide every object with a specific interface. In this case the generic external form is passed to Piccola.

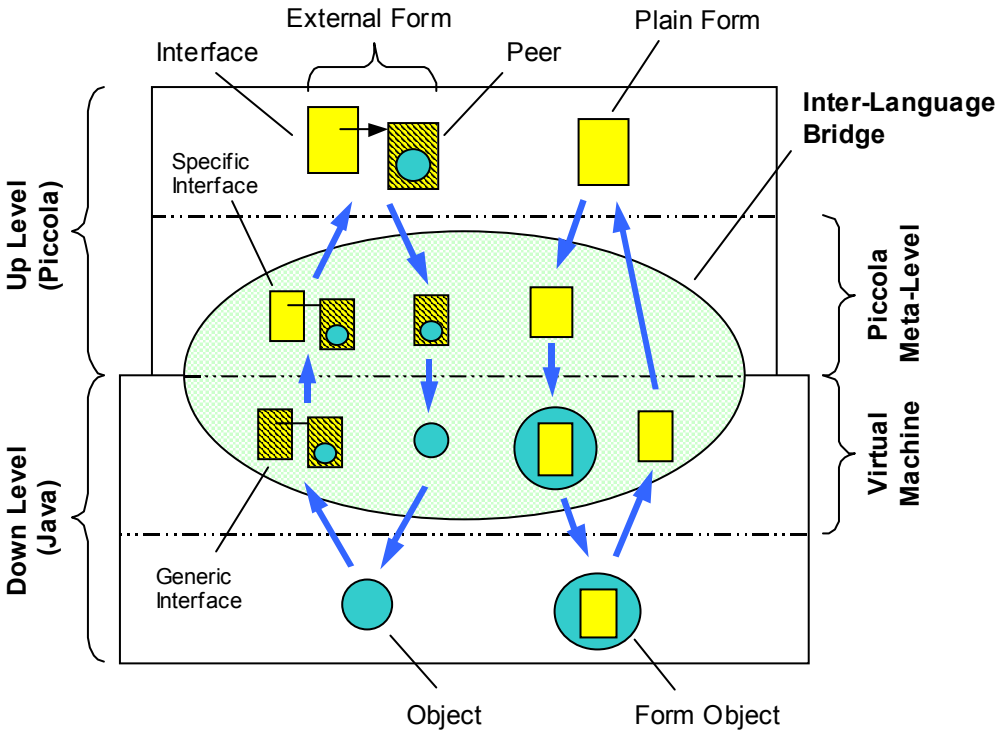

Figure 4.1: Illustration of the inter-language bridge

In the middle, we illustrate how forms are passed downwards. In the first step, the inter-language bridge takes the subform bound to the label *peer* if it is available. Otherwise, it takes the form itself. If this form is a peer form, the associated object is passed down to the host language. Otherwise, the form itself is passed downwards. As a consequence, an external form is converted to the associated external object, whereas a plain form is passed down as it is.

On the right side, an object representing a form is passed upwards. This case is very straightforward and the bridge directly passes the form to Piccola.

4.1.3 Specification of the bridging strategy

After illustrating the new bridging strategy in the previous section, we give a more formal specification in this section.

Up. Passing objects upwards

When an object is passed from the host language up to Piccola, it is the responsibility of the bridge to convert it into an appropriate form. This task is performed according to the following rules:

A. If the object is already a form, it is directly passed to the Piccola language.

B. Otherwise, the bridge builds up an appropriate external form. This task can be divided into the following two steps. The first is performed in the generic part of the bridge, whereas the second is performed in the variable part.

   B1. Create a generic external form consisting of the generic interface and the peer that represents the external identity. Refer to Section 4.2 for more details about the structure of external forms.

   B2. If there is a suitable specific interface for the object, the generic interface is replaced by the specific one. Otherwise, the bridge leaves the external form as it is. See Sections 4.3 and 4.4 for a more detailed explanation of the variable part of the bridge.

Down. Passing forms downwards

When a form *F* is passed from Piccola down to the host language, the bridge has to make sure that it is converted into the appropriate object. This task can be divided into two steps, and the second one does the same as the original inter-language bridge presented in Chapter 3.

1. If the form contains a label *peer*, let *P* be the form bound to this label. Otherwise, *P* denotes the form *F* itself.

2. In this step, we pass the object represented by the form *P* to the host language. This means:

   2A. If the form *P* represents an external object (i.e. it is a peer form), this object is passed down to the host language.

   2B. Otherwise, the form *P* itself is passed down to the host language.

4.2 Representing external objects as nested forms

In the previous section, we have introduce a nested structure for external forms, and we claim that this leads to a coherent behavior of external forms and allows a higher-level interface for external components. This section presents the details of this structure and illustrates them with an example.

4.2.1 The structure of external forms

In JPiccola 2, external objects are converted into flat forms that consist of the bindings representing the object's interface. The information that the form is actually representing an external object is not visible within Piccola. It is implicitly associated with the unmodified interface that has been created by the bridge. As the previous chapter shows, this unification of the object's identity with its Piccola interface prevents the programmer from modifying an external form without destroying its external identity.

We tackle this problem by separating the two different aspects associated with an external object. Thus, an external form is divided into a part that represents the Piccola interface respectively the glue and a part that represents the actual external object. We use the unifying concept of nested forms to achieve this separation in a way that completely conforms to Piccola's structure and does not require additional syntax or semantics. So, every form representing an external object has the following structure:

Interface and glue. The top level of an external form represents the Piccola interface and the glue that is necessary to adapt the object to a specific compositional style.

Peer. The external form contains a label *peer* bound to the peer form that actually represents the external object. This subform contains a binding for every public method available for the object. This means that it has the same structure as the forms created by step *Up.B1* of JPiccola's original inter-language bridge (cf. page 23).

Figure 4.3 illustrates the nested structure of an external form that represents an array component. At the top level of the form, there is the Piccola interface, which consists of services that allow us to access the component in an appropriate way. In addition, there is the label *peer* bound to the peer form representing the identity of the external component. This form contains services that are directly mapped to the external component. (We used the prefix *peer* to indicate that the services are directly mapped to methods of the external object). Note that the Piccola interface may have an entirely different structure than the peer.

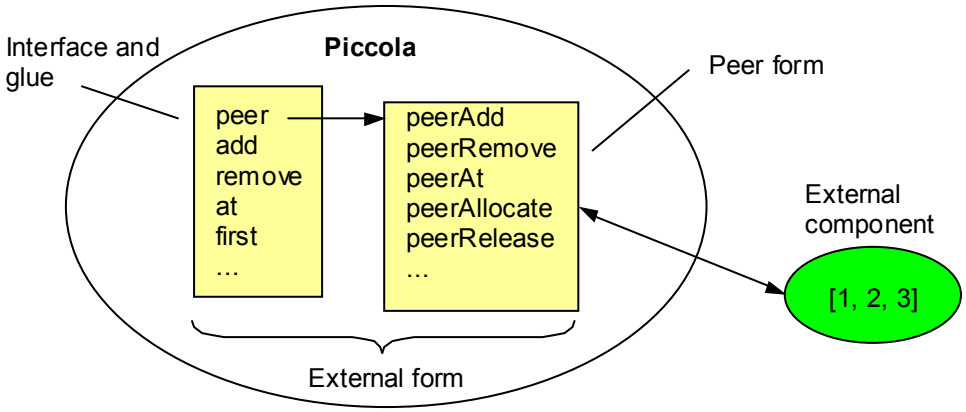

Figure 4.3: An external form consisting of interface and peer

4.2.2 Consequences and example

The explicit separation of the different aspects of an external form allows the programmer to modify these forms without affecting their external identity. In addition, the programmer can naturally influence how external forms are handled by the bridge when they are passed down to the host language (cf. Section 4.5).

The following examples illustrates that we can adapt the interface of external forms in a natural way. We first create a service `newButton`, which creates an external form representing a button and extend it with a service `setText`. Then, we call `newButton` to create a new button and use `setText` to set its label. Finally, we add the button to a Java panel. Because of the nested structure for external forms, the bridge correctly transforms the argument form into the associated button object before it is used as the argument of the `add` method.

```
# This service creates a new button and extends it with an additional interface binding
newButton:
   'button = Host.class("java.awt.Button").new()
   button
   setText(S): button.setLabel(val = "Piccola-Button: " + S)

okayButton = newButton()
okayButton.setText("Okay")      # Uses interface service to set the label
# XPiccola.piccolaPanel.buttons represents a Java Panel in the Piccola user-interface
panel = XPiccola.piccolaPanel.buttons
panel.add(val = okayButton)     # Only the button object (peer) is passed downwards
```

Note that this example is written in JPiccola 3. In Section 3.4.1, we have shown that the corresponding JPiccola 2 script fails because of the limited bridging strategy.

4.3 Wrapping external objects inside Piccola

As described in the overview of our solution, the inter-language bridge consists of two parts. In the following, we explain the structure of the bridge in detail and we particularly focus on the variable part, which is located on Piccola's meta-level. Thereby, we consider two possible models for the architecture of this part; we compare them and present our decision.

4.3.1 Structure of the inter-language bridge

The inter-language bridge presented in Section 4.1 consists of two parts (Figures 4.1 and 4.2). The lower-level part is similar to the original inter-language bridge and it is responsible for converting the entities in order to make them compatible to the other language model. This happens at a technical level and is completely generic. The higher-level part, on the other hand, makes sure that the entities are converted in an appropriate way. In the upward direction, this means that the bridge has to provide the external component with a suitable Piccola interface and glue. This task is not at all generic and may depend on many aspects of the object such as its class or instance variable values. The programmer should also be able to dynamically adapt the interface depending on the compositional style and the specific needs of the application.

We achieve this flexibility by moving the variable part of the inter-language bridge into Piccola. This allows the programmer to define the representation of external components by modifying the bridging framework within Piccola itself. Controlling the behavior of a language within the language itself is called meta-programming [16], and we therefore say that the variable aspect of the inter-language bridge is located in Piccola's meta-level. Since the variable part of the bridge wraps the peer forms with a suitable interface, we sometimes call it the *wrapping part* or the *wrapping layer*.

4.3.2 Two models for the meta-level bridging layer

There are many possible models for the design of the bridging layer in Piccola's meta-level. All of them have to provide an activation strategy and they have to find out how to wrap the external components. We can characterize these models based on the following properties:

Activation strategy. The activation strategy describes how the meta-level bridging layer gets activated when an entity is passed across the language boundary.

Dispatch strategy. The dispatch strategy describes how to decide which interface should be provided for a specific external object.

In our work, we have tested two models with very different activation and dispatch strategies. The rest of this section gives an overview of them, and in the following section we compare the models and explain our decision.

Implicit dispatching

In this model, the meta-level bridging layer is directly coupled to the bridging layer in the virtual machine. This means that after creating the generic form, the bridging layer in the virtual machine automatically calls a predefined hook service and passes the generic form as the argument. This hook is the entry-point and the dispatch service of the meta-level bridging framework, which consists of many wrapping services (wrappers). Each of these services has a name corresponding to a class of the host language. It takes a generic external form as the argument and returns the specific external form consisting of the peer wrapped with the appropriate Piccola interface and glue code. Depending on the class of the object, the dispatch service *implicitly* determines which of these wrappers is the most suitable. This means that it walks up the inheritance chain of the object's class to detect the first class with a corresponding wrapper. Then it calls this service with the generic form as the argument. The result of this operation is also the result of the bridging procedure.

To specify the interface and the glue code for external components, the programmer can add new wrappers and modify existing ones. In addition, she can adapt the dispatch service to take different criteria into account. This means that the dispatch decision may be based on other parameters than the inheritance chain of the object's class.

In SPiccola, the Squeak-based Piccola implementation, this bridging framework could look as illustrated below. The form `wrappers` contains two wrapping services `Object` and `Number`, which are supposed to wrap general Squeak objects and numeric objects, respectively. For conciseness, these services only contain a few bindings. Note that the service `Number` is an extension of the service `Object`. Furthermore, there is the hook service named `dispatch`, which is supposed to be called from the bridging layer in the virtual machine with the generic external form as an argument. The service dispatch then calls the Curried service `recursiveDispatch`, which takes the external form and the class of the external object as an argument. This service creates a first-class label with the name of the class and checks whether the form `wrappers` contains this label. If so, this wrapping service gets called with the generic form as the argument. Otherwise, the service is recursively invoked with the super-class as the second argument. Note that this recursion is guaranteed to terminate as long as there is a wrapping service corresponding to the class `Object`. Since all numeric Squeak classes inherit from `Number`, the number wrapper is called for each numeric object.

```
# This form contains wrapping services that provide the generic external form with a specific
# interface and glue code.
wrappers =
   Object X:                # This service is called if there is no specific wrapper
      peer = X.peer
      _==_ Y: X._==_ Y
   Number X:                # This service wraps any kind of number component
      Object X
      _+_ Y: X._+_ Y

# This service determines the most specific wrapping service by walking up the
# inheritance chain of the class.
def recursiveDispatch GenericForm Class:
   'label = newLabel Class.name()
   if label.exists(wrappers)
      then: label.project(wrappers) Object
      else: recursiveDispatch Object Class.superclass()

# This is the the entry-point (hook) of the bridging framework that is called with
# a generic external form
dispatch GenericForm:
   recursiveDispatch GenericForm GenericForm.class()
```

Explicit dispatching

In this model, the meta-level bridging layer is not directly coupled to the bridging layer in the virtual machine, which means that there is no automatic call of a hook method from within the virtual machine. Nevertheless, the meta-level bridging framework has a similar structure to the one described above and consists of many wrappers that provide the argument form with a specific Piccola interface and glue code. The important difference is that all the services of the interface created by the wrapper have to make sure that their results are also wrapped with a proper interface. This means that every service built by the bridge has to call a suitable bridging service to *explicitly* wrap its result. If this principle is applied consistently, all the services of an external form again return external forms wrapped with an appropriate interface. As a consequence, we only have to make sure that all the initial objects are converted correctly when passed upwards. In Piccola, there are only three different ways to create such initial objects.

Literals. Piccola supports literals for numbers and strings (cf. Section 2.3). To correctly convert them into external forms, there are two predefined hooks that are automatically called whenever a number respectively a string is newly created.

Exceptions. While a Piccola script is executed, host language exceptions may be passed up to the Piccola language. In order to wrap them properly, a special hook service gets automatically called.

***External* primitive.** As described in Section 3.1, Piccola has a primitive service *external*, which is used to access external components. Since this service is accessible within Piccola, it can easily be modified in order to return external forms with a suitable interface. However, usually the user wants to provide a more comfortable abstraction to create specific components anyway, and so he can just add the call of the bridging service there. As an example, the programmer might want to use a specific service `newArray` to create a new array rather than using `Host.class("java.util.Vector").new()`. Note that the name and the structure of the primitive service *external* are host dependent. In JPiccola 3, it is called `Host.class`.

To specify the interface and the glue code for external components, the programmer can add new wrappers to this meta-level bridging framework or modify existing ones. But other than in the previous model, it is the responsibility of these interfaces that each of its services returns an external form that is again wrapped with a proper interface. Figure 4.5 (on page 45) shows the wrapping services for numbers, strings and booleans in SPiccola 0.7.

4.3.3 Comparison of the two bridging models

Moving parts of the virtual machine into the meta-level of the language increases flexibility, but it usually has negative impacts on the performance. In Piccola, everything, even numbers and strings, are represented as external objects and therefore the inter-language bridge is used constantly and has an enormous impact on the overall performance. As a consequence, performance is the most important criteria of the following comparison and it is also the main reason for our decision.

The comparison is structured into two parts, which compare the activation strategy and the dispatch strategy, respectively.

Activation strategy

The *implicit model* uses a completely automatic and generic activation strategy. Whenever an object is passed up to the Piccola language, the virtual machine directly invokes the bridging layer on the meta-level. This has two main advantages. First, the user does not have to care about activating the meta-level bridging layer. Second, the approach needs only one predefined hook service and has therefore a minimal interface to the virtual machine.

In the *explicit model*, the activation of the meta-level bridging layer can happen in two different ways. For literals (numbers and strings) and host language exceptions, there are three hook services that are called whenever a new object is created and passed to Piccola. For everything else, it is the responsibility of the forms representing external components that all their services return objects with a proper Piccola interface. Compared to the first model, this entails the disadvantage that the programmer has to accomplish extra work when she defines the wrapping services.

Regarding the performance, both activation strategies are equivalent. In either case, there is just one additional service invocation to activate the meta-level bridging layer.

Dispatch strategy

The dispatch strategy is the main difference between the two models. In the *implicit model*, there is a dispatch method that dynamically decides which of the wrapping services has to be called for a concrete object. Unfortunately, this task is rather complex and time consuming. Finding out the next class of the inheritance chain requires at least one more call of an external service. Since the result of this call gets also wrapped it causes an indirect recursive invocation of the dispatch service. If there are many wrapping services, also the check whether a corresponding wrapping service exists takes a substantial amount of time.

Besides this performance disadvantage, there are also conceptual problems with the implicit dispatch. Depending on the paradigm of the host language, it might be difficult or time consuming to find out the class and the inheritance chain of a certain object. In particular, this might be the case if Piccola accesses a component that is distributed over the network and is perhaps even written in another language than the Piccola host.

The *explicit model* uses a static dispatch strategy. When the programmer writes an interface for an external component, he specifies for each service which wrapper has to be used to wrap the result. The main advantage of this approach is its runtime efficiency. There is no runtime dispatching necessary, and the only thing to do is looking up and invoking the wrapper.

Considering the structure of the bridging framework, the explicit model has also some advantages. The wrapper that provides an external component with the Piccola interface (including glue) ensures that the return values of the created services are provided with an appropriate interface. As a consequence, all the bridging and gluing code for a component is *explicitly* specified in the wrapper. This makes it easy to understand and adapt the bridging framework.

Our decision

After implementing and testing both of these models we decided to use the explicit model. The main reason for this decision is performance. We implemented both the implicit and the explicit bridging model in SPiccola and compared the time needed to execute different scripts. For both models we specified wrappers for numbers, booleans, strings, collections and streams. The corresponding wrappers provide the components with the same interfaces, which have between 8 (booleans) and 15 (numbers) bindings. Table 4.4 shows that the time T_i used for executing the scripts with the implicit bridging model is 3.4 respectively 5.9 times longer than the time T_e used for executing them with the explicit model. The relatively big difference between the quotients can be explained with the fact that calculating the Fibonacci Numbers creates an enormous amount of number components, and all of them pass the bridging framework.

Script	T_i / T_e
Prelude. A script specifying default Piccola abstractions and building the wrappers	3.4
Fibonacci. A script that recursively calculates Fibonacci Numbers. First, the Fibonacci Numbers are naively calculated and then they are calculated using futures that omit recalculation of the same Numbers.	5.9

Table 4.4: Performance of the explicit and implicit bridging model

Since the performance difference is entirely caused by the dispatch process, it could be decreased by optimization such as caching the dispatch result in dependence of the object's class. However, this optimization would be very specific and could not be used for other aspects of Piccola. In addition, it would prevent the programmer from using other criteria like instance variable values in the dispatch process.

Although our experience with the explicit model is mainly positive, we also encountered some situations where the implicit solution would be advantageous. Further experiments and research will show whether we have to revise our decision.

4.4 The explicit bridging model

In this section, we present the explicit bridging framework that has been introduced in the previous section. First, we illustrate its architecture, then we show its implementation, and finally, we present a few examples.

4.4.1 Architecture

The explicit bridging framework is located on Piccola's meta-level, and its architecture consists of two different kinds of services:

Wrapping services (wrappers). The wrapping services are responsible for wrapping an external object with a Piccola interface and glue. It is the responsibility of these wrappers to make sure that all the services of the created interface return objects that are also suitably wrapped.

Hook services (hooks). Whereas the design of the wrappers makes sure that all the services of an external object return properly wrapped objects, the hook services are used to ensure such an interface for the objects that are directly passed to Piccola. In fact, there are two hook services to wrap literal numbers and strings and one hook service to wrap host language exceptions.

When an object that is not representing a form is passed upwards, the generic part of the inter-language bridge first creates an external form consisting of the generic interface and the peer that represents the external identity (*Up.B1* on page 34). The peer form contains a binding for every public method that is available for the object, and it has the same structure as the form generated by step *Up.B1* of the original inter-language bridge (cf. page 23). The generic interface is identical to the peer form. Thus, a generic external form as the following structure:

```
GenericExternalForm =        # A generic external form
  peer = PeerForm            # Peer representing the external identity
  PeerForm                   # The peer form also serves as a generic default interface
```

In Java, the generic external form representing a `Vector` would therefore look as follows:

```
Vector =                     # The generic external form representing a Java Vector
  peer =
    isEmpty
    size
    iterator
    setElementAt
    ...

  isEmpty
  size
  iterator
  setElementAt
  ...
```

Usually, the generic interface of such a form is replaced through a more appropriate specific interface created by the bridging framework on the meta-level (*Up.B2*). However, in case of components without a specific wrapper, the generic external

form may be passed directly to the Piccola language. Therefore, building the generic form makes sure that every component is represented by a form that fulfills the structure defined in Section 4.2. In particular, this guarantees that external forms can be naturally modified without destroying their external identity, and it therefore ensures a uniform behavior for all Piccola forms.

4.4.2 Implementation

It should be possible to dynamically reconfigure the external components. In Piccola, this can be easily achieved by building the bridging framework within the dynamic namespace.

In Figure 4.5 on page 45, we present a simple framework containing wrappers for numbers, strings, and booleans. All the wrappers are built inside the dynamic context, and in order to avoid code duplication, the common code is factored out in helping services. As described in the previous section, the wrappers are called with the generic external form as an argument, and they return a specific external form with an appropriate interface. Each service created by the wrappers is responsible for wrapping its return value. Note that the example code is written in SPiccola 0.7.

Once we have defined the wrapping services, defining the hooks is trivial because they usually just call the wrappers. For literals, this looks as follows:

```
Hook.wrapString X: dynamic.wrapper.asString X
Hook.wrapNumber X: dynamic.wrapper.asNumber X
```

```
dynamic.wrapper =
   'addBasics X:          # Make sure that every form contains the peer form
     peer = X.peer

   'addEquality X:        # Add equality operators
     _==_ Y: dynamic.wrapper.asBoolean X._=_(Y)
     _!=_ Y: dynamic.wrapper.asBoolean X._~=_(Y)

   'addComparison X:      # Add comparison operators
     addEquality X
     _<_ Y: dynamic.wrapper.asBoolean X._<_(Y)
     _>_ Y: dynamic.wrapper.asBoolean X._>_(Y)
     _<=_ Y: dynamic.wrapper.asBoolean X._<=_(Y)
     _>=_ Y: dynamic.wrapper.asBoolean X._>=_(Y)

   asNumber X:            # Wrapper for external numbers
     addBasics X
     addComparison X
     -_: dynamic.wrapper.asNumber X.negated()
     _+_ Y: dynamic.wrapper.asNumber X._+_(Y)
     _-_ Y: dynamic.wrapper.asNumber X._-_(Y)
     _*_ Y: dynamic.wrapper.asNumber X._*_(Y)
     _/_ Y: dynamic.wrapper.asNumber X._/_(Y)
     abs: dynamic.wrapper.asNumber X.abs()
     trunc: dynamic.wrapper.asNumber X.truncated()

   asString X:            # Wrapper for external strings
     addBasics X
     addComparison X
     _+_ Y: dynamic.wrapper.asString
               newLabel("_,_").project(X) Y
     endsWith Y: dynamic.wrapper.asBoolean X.endsWith_(Y)
     indexOf Y: dynamic.wrapper.asNumber
                   X.findString_startingAt_ Y 0
     startsWith Y: indexOf(Y) == 1
     substring Y: dynamic.wrapper.asString
                   X.copyFrom_to_ Y.from Y.to
     size: dynamic.wrapper.asNumber X.size()

   asBoolean X:           # Wrapper for external booleans
     addBasics X
     addEquality X
     select = X.select
     !_: dynamic.wrapper.asBoolean X.not()
     _&_ Y:
        'block = Host.asZeroArgBlock (\x: Y)
        dynamic.wrapper.asBoolean X.and_ block
     _|_ Y:
        'block = Host.asZeroArgBlock (\x: Y)
        dynamic.wrapper.asBoolean X.or_ block
```

Figure 4.5: A simple wrapping framework in SPiccola

4.4.3 Example

In this section, we show an example that illustrates dynamic reconfiguration of external components. We assume that the following script is executed in a context that contains the wrappers of Figure 4.5 in the dynamic namespace. First we define a service `fact` that calculates the factorial of a number. Then, we create the literal numbers `3` and `2` that are automatically wrapped by calling the hook service for literals. The summation of these literals is again wrapped, because the `_+_` service built by the wrapper also wraps its result. A first call to `fact` creates many new external numbers, and all of them are properly wrapped. Then, we adapt the definition of the wrapper by adding a service `_^_` that is used to exponentiate numbers. Because the wrapper is located in the dynamic context, this extended wrapper is used for all further computations. In particular, this means that this wrapper also wraps the numbers created within the service `fact`. Therefore, the result of the second invocation of `fact` contains the service for exponentiation.

```
def fact N:
   if N > 1
      then: N * fact(N - 1)
      else: 1
argument = 3 + 2
result = fact argument
println result                    # Prints: 120

# Extend the wrapper asNumber
# Remember the original definition of asNumber. (Necessary to avoid a loop when it is called)
originalAsNumber = dynamic.wrapper.asNumber
dynamic.wrapper.asNumber X:
   originalAsNumber X
   _^_ Y: dynamic.wrapper.asNumber (Y * X.ln()).peer.exp()

# All these computations use the extended wrapper.
# Therefore, the result of fact also contains the service for exponentiation.
argument = 2 ^ 2
result = (fact argument) ^ 3  # The result contains the service for exponentiation
println result                    # Prints: 13824
```

4.5 Protecting forms from being converted

In the last two sections we developed a solution that allows the programmer to control the upward direction of the inter-language bridge in a very flexible way. The following example shows that the programmer should also be able to control the behavior of the downward direction. The rest of this section presents a simple and

natural approach to do this and shows how it can be integrated within the wrapping services on the meta-level.

4.5.1 Introductory example

Whenever an external form gets passed down to the host language, it is automatically converted into the associated external object (cf. Section 4.1 and Figure 4.1). In most cases, this makes perfect sense because the host language should deal with the associated object rather than with the form itself. However, there are some cases where an external form should be passed down to the host language as it is. In particular, this is the case with host-based collections. A Piccola programmer expects that adding a form to a collection and retrieving it again yields the exact same form (i.e. it does not modify the form). But as the following example illustrates, this is not true for external Piccola forms.

We assume that the service `newList` creates a new host-based list. Then we create two forms `p` and `e` and add them to this list. The form `p` is a plain Piccola form and therefore it is not modified when it is added to the list and retrieved again. The form `e`, on the other hand, is an external form (representing 5) extended with a service `inc`. When it is added to the host-based list, it is therefore converted into the object 5, and the binding `inc` is lost. As a consequence, the form `e2` does not contain the binding labeled `inc` anymore and the script results in an error.

```
list = newList()     # Creates an empty list
p = (name = "Peter Brown", age = 28)
e =
   5                 # Creates an external form consisting of an interface and a peer
   inc: 6            # Adds a new service to the interface of the number

list.add p           # The form p is passed to the host language as it is
list.add e           # The form e gets converted to the object 5 when passed downwards

p2 = list.at 1
e2 = list.at 2
println p2           # Prints: (name = "Peter Brown", age = 28)
println e2.inc()     # Error! (The binding inc got lost)
```

Figure 4.6 illustrates what happens when we add the form `e` to the list and retrieve it again. When we use `e` as the argument of `add`, the bridge projects on the label *peer* (*Down.1* on page 34), and only the associated external object `5` is passed downwards (*Down.2A*). Thus, the interface containing the service `inc` is discarded. When we read from the list, the object `5` is first converted into a generic external form (*Up.B1*), and afterwards, the generic interface is replaced by the specific interface (*Up.B2*). Note that neither of these interfaces contains the service `inc`.

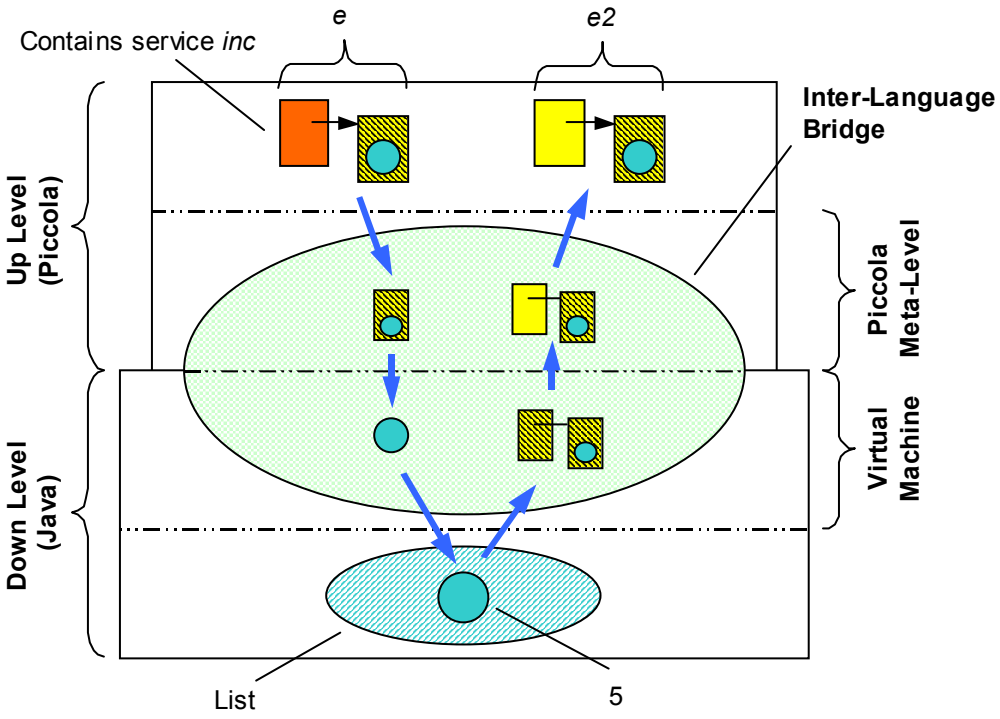

Figure 4.6: Using a collection without protecting the argument form

4.5.2 The protect service

By analyzing the problem in the previous example, we see that external services can be categorized into two non-disjoint groups, and we call them *clients* respectively *containers*:

Clients. Clients are services that use the argument (and its services) for doing some computations. Usually, clients can only deal with specific types of objects (e.g. numbers), and therefore, an external form has to be converted into the associated object before it is passed as an argument to a client. A typical example of this category is the summation service of a numeric object. It takes a number as an argument and returns a newly created number. Most of the clients do not return the argument itself and do not store it for later retrieval.

Containers. Containers are services that store the argument without making use of its services. Usually, containers are designed to take any kind of objects as an argument. A typical example of this category is a service to add an object to a collection.

If an external form is passed as an argument to a client, it is important that the inter-language bridge converts the form into the associated object. At the same time, external forms used as the argument of containers must not be converted to the associated object in order to retain their interface. Unfortunately, the inter-language bridge has no means to find out what category an external service belongs to. In fact, there are even services that are both clients (do some computations with a specific argument type) and containers (store the argument for later retrieval). In this case, the programmer has to decide whether the argument form gets converted. Because these services usually require the argument to provide specific methods they have to be treated like clients, which means that the argument has to be converted to the associated object.

Since nearly all of the external services are clients, the bridge converts an external form to an object by default. Although this is appropriate in most of the cases, it leads to the problem illustrated in the previous section in connection with containers.

Due to the design of external forms and the inter-language bridge, the programmer can easily avoid this problem by using the following simple service:

```
protect X:
   peer = X
```

This service takes an argument `X` and returns a form consisting of a single label `peer` that is bound to the argument form, which means that the argument form `X` itself should be considered as the peer object. If we apply the bridging rule on page 34 to this form, we see that it gets converted to the argument form `X` independent of whether `X` is an external form or not. Thus, the service `protect` ensures that the argument form is never converted when passed to an external service. We therefore say that it *protects* the argument form from being converted.

In the previous example, this service can be used to protect the form `e` from being converted when it is passed as an argument to the service `add` of the list. This means that we must replace the line

```
list.add p
```

by

```
list.add (protect p)
```

in the previous example. Then, the example works as expected and the retrieved form `e2` is identical to the original form `e`.

Figure 4.7 illustrates what happens when we add the form `protect e` to the list and retrieve it again. When we use this form as the argument of `add`, the bridge projects on the label *peer* (*Down.1* on page 34) and the resulting form `e` is passed downwards as it is (*Down.2B*). Then we read from the list, and the form `e` is directly passed to Piccola (*Up.A*).

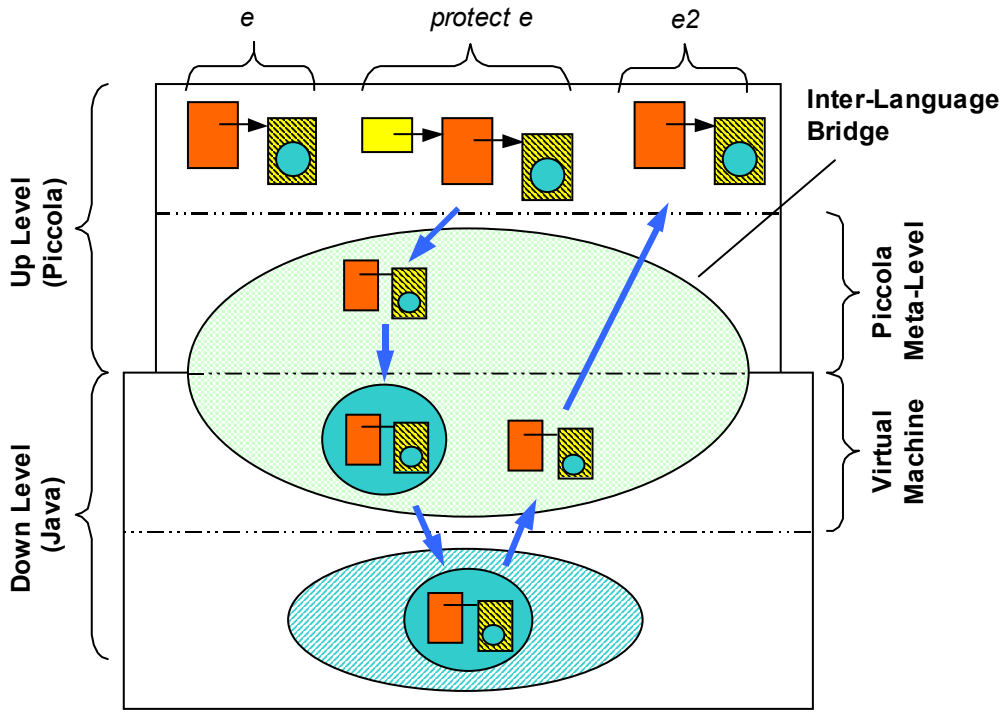

Figure 4.7: Using a collection with a protected argument form

4.5.3 Using protect on the meta-level

Since Piccola is a pure composition language, external services are mainly used within external components that have a corresponding wrapping service in the bridging layer. For every service provided by a component, the programmer knows whether it is a client or container and so she can use the protect service already within the definition of the component's interface on the meta-level. This leads to a clean design where all the technical details of an external component are covered within the wrapping service on the meta-level, and the user of the component does not have to care about them. From the user's point of view, the inter-language bridge is completely transparent and automatically converts the arguments and the results in an appropriate way.

In the following example, we define a wrapper for an external list component that automatically protects the arguments of containers. This allows us to deal with list components in a natural and high-level way. Note that this example is written in SPiccola 0.7 and that we use some services defined in Figure 4.5.

```
# High-level list wrapper that automatically protects the arguments of containers
dynamic.wrapper.asList X:
   peer = X.peer
   at Y: X.at_ Y
   add Y: 'X.add_(protect Y)                    # Protect the argument
   addAll Y: 'X.addAll_ Y
   remove Y: 'X.remove_(protect Y)              # Protect the argument
   removeAll Y: 'X.removeAll_ Y
   _?_ Y: dynamic.wrapper.asBoolean
             X.includes_(protect Y)             # Protect the argument
   size: dynamic.wrapper.asNumber X.size()
   isEmpty: dynamic.wrapper.asBoolean X.isEmpty()

# This service creates a new high-level list component
newList():
   dynamic.wrapper.asList
     Host.smalltalk.at("OrderedCollection").new()

# Using the high-level list component
list = newList()     # Create an empty list
p = (name = "Peter Brown", age = 28)
e =
   5                 # Create an external form consisting of an interface and a peer
   inc: 6            # Add a new service to the interface of the number

list.add p           # The form p is automatically protected and passed downwards as it is
list.add e           # The form e is automatically protected and passed downwards as it is

p2 = list.at 1
e2 = list.at 2
println p2           # Prints: (name = "Peter Brown", age = 28)
println e2.inc()     # Prints: 6
```

Chapter 5

Optimization using lazy evaluation

In the previous chapter we developed a strategy to access external components in a flexible and high-level way that allows dynamic reconfiguration. One of the key concepts of this approach is that the variable part of the inter-language bridge has been moved from the Piccola virtual machine into Piccola's meta-level. The disadvantage of this technique is the performance penalty that is caused by calling the Piccola based bridging abstractions instead of doing all the bridging related work in the virtual machine. In this chapter, we show how we can overcome this performance bottleneck by using lazy evaluation, and we develop a partial evaluation algorithm to transform a Piccola script into a semantically equivalent script that allows us to apply our lazy evaluation technique very effectively.

This chapter is structured as follows: In Section 5.1, we profile a simple expression and show that the meta-level bridging framework spends an enormous amount of time for building interface bindings that are never used. In Section 5.2, we introduce a lazy evaluation technique based on lazy forms and derive the requirements for a Piccola service to allow effective lazy evaluation. Then, we introduce a partial evaluation algorithm that transforms a general Piccola service into a semantically equivalent service fulfilling these requirements. Section 5.3 gives an informal illustration of this partial evaluation algorithm, and in Section 5.4, we present a formal specification. In Section 5.5, we reason about how to prove its correctness, and in Section 5.6, we discuss implementation issues. Finally, we present some application examples in Section 5.7.

5.1 Piccola's inter-language bridge

As a pure composition language Piccola is constantly dealing with external components. In this section, we first examine a simple expression with regard to its performance and show that there is an enormous amount of time spent in the wrapping services of Piccola's bridging framework. Then, we show that most of this time is used for building parts of the interface that are never used.

5.1.1 Profiling a simple expression in SPiccola

In this section, we examine the impact of the inter-language bridge on the overall performance by executing and profiling the simple expression `result = 1 + 2` in SPiccola.

First, the expression is executed with a modified SPiccola version that does not use inter-language bridging at all and directly operates on the external components. The time used for this direct execution serves as a reference, and we define it to be *t*.

In the second step, we execute the same expression with the real SPiccola version. Since Piccola is a pure composition language, even integers are represented by external components, and they pass the bridge in the same way as the other external objects (cf. Section 4.1). In our example, we use integer objects of the host language and wrap them with the service `asNumber` presented in Figure 5.1. This service is called with the generic external form as an argument, and it yields an external form with a suitable Piccola interface as the result. Some of the services built by this wrapper are mapped to a corresponding Squeak method. As an example, the arithmetic plus operator (`_+_`) takes the right-hand side operand `Y`, calls the native Squeak method (available through `X._+_`) and wraps the result by applying the wrapper `asNumber` again. Other services like `abs` are specified using previously defined bindings.

```
dynamic.wrapper =
   'addComparison X:
      _==_ Y: dynamic.wrapper.asBoolean X._=_(Y)
      _!=_ Y: dynamic.wrapper.asBoolean X._~=_(Y)
      _<_ Y: dynamic.wrapper.asBoolean X._<_(Y)
      _>_ Y: dynamic.wrapper.asBoolean X._>_(Y)
      _<=_ Y: dynamic.wrapper.asBoolean X._<=_(Y)
      _>=_ Y: dynamic.wrapper.asBoolean X._>=_(Y)

   def asNumber X:
      peer = X.peer
      addComparison X
      -_: asNumber X.negated()
      _+_ Y: asNumber X._+_(Y)
      _-_ Y: asNumber X._-_(Y)
      _*_ Y: asNumber X._*_(Y)
      _/_ Y: asNumber X._/_(Y)
      abs:
         if (_<_ 0)
            then: -_()
            else: asNumber X
      trunc: asNumber X.truncated()
```

Figure 5.1: A number wrapper for SPiccola

In Table 5.2, we give an overview of how much time is spent in the different parts of the inter-language bridge. We use the names defined in Section 4.1.1 to refer to the different steps of the bridging procedure, and we only list the steps that are effectively used in this example. Recall that *t* is the amount of time used for executing the same expression without inter-language bridging at all.

Bridging activity	**Time**
Up.B1. Create the peer form for the external object. This is performed within the Piccola virtual machine and it is therefore very efficient.	$< 0.05t$
Up.B2. Create the appropriate Piccola interface. This step is accomplished by the bridging framework within Piccola. In our particular example, the wrapping service shown in `asNumber` gets executed, which is very time consuming.	$6t$
Down.1. Check for peer binding. This check can be done in the virtual machine and is therefore very efficient.	$< 0.05t$
Down.2. Passing the associated object to Squeak. This is a trivial operation that virtually needs no time.	$\approx 0t$

Table 5.2: Time spent in the inter-language bridge

These results show that the bridging activities triggered by the execution of the example expression take about 6 times more time than executing the actual code, and about 98% of this extra time is spent in the Piccola part of the inter-language bridge.

5.1.2 Unused interface bindings

The previous section shows that there is an enormous amount of time spent in the wrapping services of Piccola's bridging framework. In the following we have a closer look at the usage of the interfaces built by these services, and we see that only a small percentage of the provided bindings is actually used.

Execution of our example expression `result = 1 + 2` causes the following bridging related activities:

1. The external object representing the number `1` is wrapped by the service `asNumber`, which yields a form `F`.
2. A projection on the label _+_ of the wrapped form `F` is performed.
3. The external object representing the number `2` is wrapped, which yields a form `G`.

4. The wrapped form `G` is passed as an argument to the service _+_. This service sends the message `+` to the Squeak object `1`. As the argument we use the projection on the label `peer` of the form `G`[1], which yields the object `2`.
5. The result, the number 3, is passed back to Piccola and wrapped by the service `asNumber`.

The wrapping service `asNumber` is invoked in steps 1, 3 and 5 and each time, it builds up the whole interface consisting of 14 bindings. But for the forms built in 1 and 3, only one of these bindings is effectively used while all the others are discarded.

Examination of other Piccola scripts show that we usually only use a small amount of the bindings created by the wrapping services. As an example, we can consider numbers or string components. To be usable, they usually provide at least 10 to 20 bindings. But in most of the cases, we use only one or two of them for every component we have created (e.g. we add two numbers, compare two numbers, print a string, concatenate two strings). Also for more complex components like URLs, the situation is similar. A URL component might also have at least 10 bindings, but oftentimes, we only use one or two of them (e.g. we retrieve the contents at a URL).

If we therefore assume that we only use 10% of the bindings created by the wrapping services and we further assume that the time used for setting up such an interface is uniformly distributed over the created bindings, this means that the overhead for building such an interface would be reduced by 90% if we only created the bindings that are actually used. Applied to our example, the time spent in the bridging service would be reduced from $6t$ to $0.6t$ and as a consequence, the inter-language bridge would slow down the execution of an average script by less than factor 2 instead of factor 7. We verified this theoretical result by modifying the wrapping services so that they only generate the effectively used bindings.

It should be noted that these results are based on wrappers that only provide small interfaces. As an example, the number wrapper provides an external number with 14 services. In contrast, the `SmallInteger` objects of Squeak v2.9 understand more than 400 messages. Thus, the performance problems caused by the wrappers will be much more dramatic if an application requires more complex components.

5.2 Lazy evaluation

We have illustrated that we can significantly improve Piccola's performance if the wrapping services only build the parts of the interfaces that are actually used. In this

[1] Projection on the label `peer` is implicitly performed by the generic part of the inter-language bridge *(Down.1)*.

section, we show that this can be achieved by using a lazy evaluation strategy. Then, we derive the requirements for a service to allow effective lazy evaluation, and we see that only a small subset of the Piccola services actually satisfies them. Therefore, we develop a partial evaluation strategy that transforms a general Piccola service (respectively a script) into a semantically equivalent service that fulfills these requirements.

5.2.1 A lazy evaluation strategy using lazy forms

Lazy evaluation is an evaluation strategy combining normal order evaluation with updating. Under normal order evaluation (outermost or call-by-name evaluation) an expression is evaluated only when its value is needed in order for the program to return its result. Updating means that if the value of an expression is needed more than once, the result of the first evaluation is remembered and subsequent requests for it will return the remembered value immediately without further evaluation [19].

In order to use lazy evaluation in Piccola we introduce the notion of *lazy forms*. Lazy forms represent the result of a service invocation that is not yet evaluated or only partially evaluated. This allows us to split up service application into invocation and utilization time. Note that general Piccola services are not referentially transparent and that we have to execute the side effects at invocation time in order to preserve the semantics:

Invocation time. At invocation time, we execute the side effects of the service and return a lazy form that remembers the functional part (i.e. the referentially transparent part) of the service, the form containing the results of the side effects, and the concrete argument.[1]

Utilization time. When a certain binding of a lazy form is used, only the expression associated with this binding is executed and remembered (to avoid subsequent executions of the same expression). This expression may refer to the invocation argument and the results of the side effects that are stored in the lazy form.

5.2.2 Requirements for lazy evaluation

Looking at the lazy evaluation strategy introduced above, we see that there are two critical requirements for Piccola services to evaluate them lazily:

[1] In Piccola, the dynamic context is implicitly passed whenever a service is invoked (cf. 2.4.6). Therefore, a lazy form also has to remember the dynamic context at invocation time. This behavior can be modeled by explicitly passing the dynamic context together with the argument. For simplicity, we do not consider the dynamic context throughout this chapter.

Separated side effects. Piccola is not a pure functional language, which means that Piccola services may have side effects. Applying lazy evaluation in presence of side effects is problematic [17], and we have to make sure that all the expressions causing side effects are executed only once and in the right order. If we apply lazy evaluation to general Piccola services, we therefore execute the side effect of the service at invocation time and return the purely functional part as a lazy form. In order to do that efficiently, the side effects of the service have to be separated from the functional part.

Closed expressions. When a binding of a lazy form is effectively used, only the expression associated to this binding is executed. This can only be done if all the individual expressions of the service are closed, which means that they do not contain free identifiers (except the ones referring to the arguments of the service).

5.2.3 Using partial evaluation to meet the requirements

Although some of the wrappers used in the bridging framework may already fulfill the two requirements for lazy evaluation, general Piccola services do not. In this section, we therefore introduce a transformation based on partial evaluation that allows us to apply lazy evaluation to all Piccola services.

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input [18]. In our situation, the input is not explicitly given, but it can be derived from the static information exhibited by any Piccola script. This information is collected and used to transform a service (respectively a script) into an equivalent service that fulfills the requirements for lazy evaluation and can therefore be executed more efficiently. Although we could express a transformed Piccola script with the standard syntax presented in Section 2.3, we introduce a new syntactic domain that is more suitable to specify Piccola scripts with separated side effects and a referentially transparent part that consists of closed expressions. Elements of this domain are called *lazy form expressions*.

Figure 5.3 gives an overview of all the syntactic and semantic domains involved in our lazy evaluation strategy, and it shows how their elements are transformed respectively evaluated. We use bold letters (such as $\boldsymbol{E}$ or $\boldsymbol{E}$*) to denote the domains and use normal letters to denote individual elements of the domains (such as E or E*). The standard syntactic domain $\boldsymbol{E}$ consists of *form expressions* as described in Section 2.3. The standard Piccola interpreter *eval* takes such an expression E and evaluates it to a *form* F, which has the structure described in Section 2.2. Alternatively, we can use the partial evaluation algorithm *split* to transform a form expression E into a *lazy form expression* E*. Every lazy form expression is represented as a tuple $(P; S)$, where P denotes a referentially transparent *functional expression* and S denotes a *side effect expression*. At runtime, we first evaluate the side effect

expression and return a lazy form *F**, which contains the functional expression *P*, the result of the side effect evaluation, and the concrete service arguments. Finally, the effectively needed bindings of the lazy form *F** are evaluated to forms by applying the interpreter *eval* to the functional expression *P* in *F**. Thereby, we use the result of the side effect evaluation and the concrete service argument that are available in *F**.

In the rest of this chapter, we mainly focus on the partial evaluation algorithm *split*, which transforms a form expression *E* into a lazy form expression *E**. In Section 5.3, we give an informal illustration, and in Section 5.4, we define it formally.

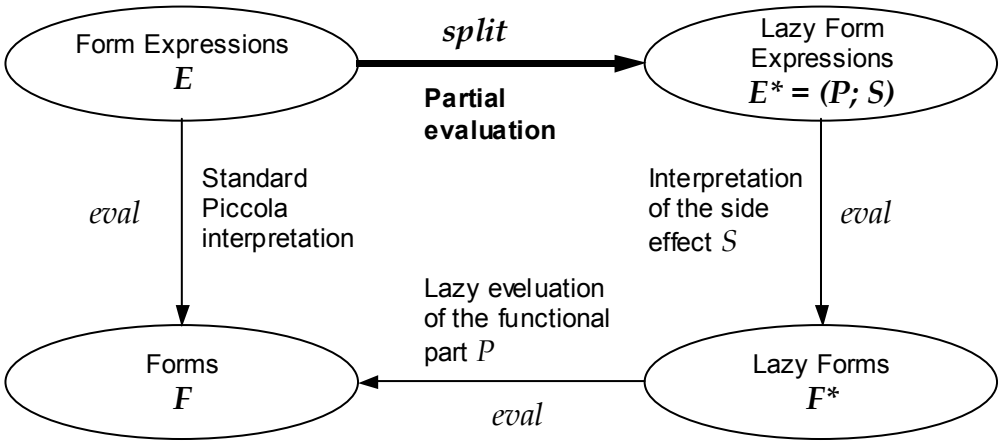

Figure 5.3: Syntactic and semantic domains

5.3 Illustration of the partial evaluation algorithm

In this section, we give an informal illustration of the partial evaluation algorithm *split* that transforms a Piccola service so that it can be evaluated lazily. As we explained in the previous section, the task of this algorithm consists of two parts: First, it separates the side effect of a service from the referentially transparent part, and second, it turns the referentially transparent part into a closure. For a better understanding, this section individually explains these two parts and illustrates them with examples.

It is important to note that the decomposition of the algorithm *split* into these parts only serves for a better understanding. In reality, the two parts are tightly interlinked and it would not be possible to achieve an independent sequential decomposition. See Section 5.4 for a formal specification of the partial evaluation algorithm *split* as a whole.

5.3.1 Part 1 — Separating the side effect

The first requirement in Section 5.2.2 demands that the side effect of a service is separated from its functional part. In the following, we first present the concept of our separation strategy. Then, we give an informal description and illustrate it with an example.

Basic concept

The basic idea of our separation strategy is to replace a general Piccola service *f* by two services f_s and f_p, which fulfill the following properties:

Side effect part f_s. The service f_s contains all the expressions of the service *f* that may cause side effects in the same order as they occur in *f*. As a consequence, f_s causes identical side effects as *f* if invoked with the same argument. The service f_s returns a form that contains the results of the side effect operations so that they are separately accessible.

Functional part f_p. The service f_p is a *referentially transparent* service of order two. The body of f_p builds up the same form as *f*, but all the expressions causing side effects are replaced by appropriate projections on the second argument. If f_p is invoked with an arbitrary form *x* as the first and f_s *x* as the second argument, the return value is identical to the result of *f x*.

More formal, we can say that for any service *f* and an arbitrary form *x*, the service application *f x* is equivalent to the nested application f_p *x* (f_s *x*). Equivalent means that evaluation of the expressions causes *identical side effects* and yields *identical result forms*. Since f_p has no side effect we can directly apply the lazy evaluation technique described in Section 5.2.1.

As an example, we consider the service `chFact`, which reads a value from a communication channel and uses it for some referentially transparent arithmetic operations:

```
chFact Ch:
  value = Ch.receive()          # A blocking read operation on the channel
  factorial = fact value        # Calculate factorial
  status = factorial < 256
```

Reading from a communication channel is not referentially transparent, and therefore, the side effect part of `chFact` is the following service.

```
\Ch:
  y1 = Ch.receive()             # y1 is a fresh identifier in chFact
```

At the same time, the functional part is represented by the following higher order service. Note that the invocation of the non-transparent service receive is replaced by

a projection on the second argument which is supposed to contain the result of the side effects.

```
\Ch Side:
  value = Side.y1
  factorial = fact value
  status = factorial < 256
```

Description of the algorithm

In the following, we give a simplified description of the algorithm that iterates through a Piccola script and transforms each service f into a service f_s representing the side effect part and a service f_p representing the purely functional part. In Piccola, the operations causing side effects can be divided into three categories: Spawning new *agents*, reading respectively writing on *channels*, and invoking *external services* such as `println`[1]. All these operations are represented by service applications, which implies that service applications are the only operations that may cause side effects.[2] As a consequence, the separation algorithm has to focus on service applications and can be described as follows. Note that the service f_p takes two arguments, the second of which is named *side*.

- Except from service applications, the functional part f_p consists of the same expressions as the original service f and the side effect part f_s is empty at the beginning.

[1] Many of the external service do not have side effects and the programmer may use pragmas to indicate referentially transparent external services.

[2] A projection $P.x$ returns a runtime exception if the label x is not bound in P. Therefore, also projections can lead to a side effect. For simplicity, these kinds of side effects are not covered in this section. See the formal specification in Section 5.4 for more details.

- In case of a service application $g\ z$, the following happens:
  - If the functor g is referentially transparent, the application $g\ z$ is directly used in the functional part f_p.
  - If the functor g has already been transformed into g_p and g_s, the side effect part f_s is extended with the binding $y_i = g_s\ z$, and we use $g_p\ z\ side.y_i$ as the functional part in f_p (y_i denotes an identifier that is fresh in f_p). This means that the side effect of g is executed in the side effect part of f whereas the functional part of g is evaluated in the functional part of f. Note that the result of the side effect part g_s is bound to the fresh label y_i in the side effect part. When evaluated, this result is passed as a second argument to the functional part g_p.
  - If the functor g has side effects and cannot be decomposed further (e.g. it is an external function such as *println*), the side effect part f_s is extended with the binding $y_i = g\ z$, and we use $side.y_i$ in f_p (yi denotes an identifier that is fresh in f_p). This means that the service application happens in the side effect part f_s and we just refer to its result in the functional part f_p.

Example

Table 5.4 presents an example script and shows how this algorithm transforms it. This script is kind of artificial but it illustrates all the different aspects of our algorithm. The script defines the services `fact`, `f` and `g`. We assume that the service `if` and the integer operators `+`, `*`, `-`, and `>` are known to be referentially transparent. As a consequence, the service `fact` is also referentially transparent and transforming it is trivial. The service `f` invokes the services `println`, `fact`, `receive`, and `_>_`. The services `fact` and `_>_` are referentially transparent and we therefore directly use them in the functional part $\texttt{f}_\texttt{p}$. The services `println` and `receive` contain side effects or they are not known to be side effect free, respectively. Therefore we invoke them in the side effect part $\texttt{f}_\texttt{s}$ and reference the results in the functional part $\texttt{f}_\texttt{p}$. In the service `g`, the invocation of `f` is particularly interesting, since we have already transformed `f` into $\texttt{f}_\texttt{s}$ and $\texttt{f}_\texttt{p}$. We invoke $\texttt{f}_\texttt{s}$ in the side effect part $\texttt{g}_\texttt{s}$ and use the result as the second argument of the invocation of $\texttt{f}_\texttt{p}$ in the functional part $\texttt{g}_\texttt{p}$. Note that after transforming the services `f` and `g`, every application of the time-consuming service `fact` happens in the functional part and is therefore only executed if it is effectively needed.

Original script	Transformed script

`def fact N:`
`  if N > 1`
`  then: N * fact N - 1`
`  else: N`
	Fact is referentially transparent and therefore it remains as it is.

`f X:`
`  println X`
`  b = fact X`
`  c = ch.recive()`
`  d = b > c`
	f_s `X:`
	`  y1 = println X`
	`  y2 = ch.receive()`
	
	f_p `X Side:`
	`  Side.y1`
	`  b = fact X`
	`  c = Side.y2`
	`  d = b > c`

`g Y:`
`  a = f(Y).b`
`  b = fact Y + a`
	g_s `Y:`
	`  y1 =` f_s `Y`
	
	g_p `Y Side:`
	`  a =` f_p`(Y)(Side.y1).b`
	`  b = fact Y + a`

Table 5.4: Separating the side effect from the functional part

5.3.2 Part 2 — Turning expressions into closures

The second part of our partial evaluation algorithm makes sure that all the expressions of the service are closed, which means that they do not contain free identifiers (except the ones referring to the arguments of the service). As explained in Section 5.2.2, closed expressions are a necessary requirement to evaluate only the effectively needed expressions in the referentially transparent part of a service.

In the following, we give an informal description of our algorithm that transforms a service *f* into an equivalent service *f'* that exclusively contains closed expressions.

Description of the algorithm

This partial evaluation algorithm is based on a static interpreter that works similar to the real Piccola interpreter. But unlike the real one, the static interpreter does not execute any service applications. It only keeps track of the bindings in the root context (the static namespace) and statically resolves and simplifies references to it. All the expressions that modify the root context (*sandbox* and *quote*) are eliminated in the resulting services. As a consequence, these services consist of independent expressions that can be evaluated in an arbitrary order.

If the algorithm is applied to a service *f*, the following happens:

- The state of the interpreter consists of two special forms that can hold partly evaluated form expressions. One of them represents the resulting service (*value*) whereas the other represents the static namespace (*root*). Both are initialized with the empty form and are managed according to Piccola's scoping rules (cf. Section 2.4.5).
- Except from *sandbox* and *quote*, all the form expressions modify the resulting service. When the interpreter encounters such an expression it resolves all the dependencies to the static namespace and tries to simplify it. In particular, this means that it replaces identifiers by projections on *root* and simplifies projections by static pre-evaluation. Then, the modified expression is appended to the resulting service (*value*).
- Every expression that modifies the static namespace gets appended to *root*. According to Piccola's syntax (Section 2.3), these expressions are *sandbox*, *quote*, *binding*, and *service binding*.
- After the whole service is interpreted, *value* contains the transformed service *f'*, which is guaranteed to have closed expressions. This means that the only free identifiers are the service arguments.

Depending on the concrete implementation of this transformation and the interpretation of lazy forms, application of the transformed service *f'* may execute some parts of the original service *f* multiple times. Because the real algorithm (cf. Section 5.4) only applies this transformation to the referentially transparent part of a service, this does not change the semantics. Nevertheless, a real implementation should avoid multiple executions of the same expressions for performance reasons, and it should use pointers (references) to decrease the size of the generated code (cf. Section 5.6).

Examples

As a first example, we consider the service `chFact` defined in Section 5.3.1. We have shown that the separation of the side effect yields the functional part:

```
\Ch Side:
  value = Side.y1
  factorial = fact value
  status = factorial < 256
```

If we apply the algorithm to resolve internal dependencies, this service is transformed into the following service, which has closed expressions that only reference the arguments `Ch` and `Side`.

```
\Ch Side:
  value = Side.y1
  factorial = fact Side.y1
  status = (fact Side.y1) < Side.y2
```

The second example that is shown in Table 5.5 is rather artificial, but it is useful to illustrate the different aspects of our partial evaluation algorithm. It shows a service `f` and its transformation `f'`. Note that `f'` has completely independent expressions and that *quote* and *sandbox* expressions are eliminated. Because the impact of the expression `'X` on the root context cannot be statically determined (the bindings of the argument `X` are not known at compile time), the expression `d = u` gets transformed into `d = (u = 10, X).u`. This means that the label `d` will be bound to `X.u`, if the argument `X` contains a label `u`. Otherwise, the label `d` will be bound to `10`, which is the statically determined value associated to the label `u` in the lexical namespace.

Original script	**Transformed script**

`f X:`
`  a =`
`  u = 10`
`  v = u + X`
`  b = a.v`
`  'a`
`  c = u * v`
`  'X`
`  d = u`
`  root = b`
`  e = abs()`
	`f' X:`
	`  a =`
	`  u = 10`
	`  v = 10 + X`
	`  b = 10 + X`
	
	`  c = 10 * (10 + X)`
	
	`  d = (u = 10, X).u`
	
	`  e = (10 + X).abs`

Table 5.5: Resolving internal dependencies

5.4 Formal specification

In the following, we give a formal specification of our lazy evaluation strategy. First, we define the involved domains and show the definition of the standard Piccola interpreter *eval*. Then, we specify the partial evaluation algorithm *split*, which separates a service into a side effect expression and a functional expression. These expressions are very similar to standard Piccola expressions, but they may contain unevaluated substitutions. Thus, we have to extend the definition of the interpreter *eval* in order to evaluate them correctly. Finally, we illustrate our lazy evaluation technique with a few examples.

5.4.1 The domains

Section 5.2.3 and Figure 5.3 give an overview of the syntactic and semantic domains that are needed for our lazy evaluation strategy. In the following, we define them formally.

Identifiers

We use $\boldsymbol{X}$ to denote the domain of all the identifiers x. In addition, we define the domain $\boldsymbol{Y} \subset \boldsymbol{X}$ to be the domain of all the identifiers y_i that are used to refer to a side effect. To avoid name clashes, we assume that the identifiers y_i in $\boldsymbol{Y}$ are distinct from all the identifiers used in form expressions E.

Form expressions and forms

Although Piccola has a small syntax (cf. Section 2.3) there is a lot of syntactic sugar [9]. For conciseness, we only consider the core expressions in this section. Thus, *form expressions* E are inductively defined as follows:

$$E \ ::= \ \varepsilon \mid x \mid E \cdot E \mid E.x \mid E\ E \mid \backslash x\text{: } E \mid \textbf{\textit{root}} \mid x = E,\ E \mid \textbf{\textit{root}} = E,\ E$$

The meaning of these expressions is the same as described in Sections 2.2 and 2.3: The empty expression is denoted by ε. We use $x \in \boldsymbol{X} \setminus \boldsymbol{Y}$ to range over identifiers, and $E \cdot E$ stands for form extension. We use $E.x$ to express projection, $E\ E$ for service application and $\backslash x\text{: } E$ for service definition. The expression ***root*** refers to the environment where identifiers are looked up (static namespace). Finally, $x = E_1,\ E_2$ stands for *binding* and ***root*** $= E_1,\ E_2$ for *sandbox* (cf. Section 2.3). Note that for both *binding* and *sandbox* the expressions $x = E_1$ respectively ***root*** $= E_1$ modify the environment where E_2 is executed. As for all the other domains used in this chapter, the extension operator $\cdot$ is associative.

The standard Piccola interpreter f takes such a form E and evaluates it to a form F. Forms F are inductively defined as follows:

$$F \ ::= \ \varepsilon \mid x = F \mid \textbf{\textit{Cl}}(x;\ E;\ F) \mid \textbf{\textit{primitive}}_i \mid F \cdot F$$

The empty form is denoted by ε and we use $x = F$ to express a binding. Form extension is expressed by $F \cdot F$, and $\textbf{\textit{Cl}}(x;\ E;\ F)$ denotes a service (closure), where x is the formal argument, E is the expression that defines the body of the service and the form F is the context where free identifiers are looked up. In addition, we use $\textbf{\textit{primitive}}_i$ to model primitive services respectively external services. These services are provided by the Piccola virtual machine respectively the host language and do not have a definition inside Piccola.

Note that we sometimes use $\boldsymbol{AF}$ to denote the domain of *atomic forms*. An atomic form AF is defined to be any form that does not contain form extension.

Lazy form expressions and lazy forms

Our partial evaluation algorithm *split* transforms a Piccola expression E into a *lazy form expression* E^*, where the side effects are separated from the referentially transparent part. Accordingly, a lazy form expression is defined as a tuple consisting of a *functional expression* P and a *side effect expression* S.

$$
\begin{array}{lll}
E^* & ::= & (P;\ S) \\
P & ::= & \varepsilon \mid x \mid P \cdot P \mid P.x \mid x = P \mid \backslash x{:}\ E^* \mid \textbf{\textit{Re}}(P;\ P;\ P) \mid \\
 & & \textbf{\textit{projectEx}} \mid \textbf{\textit{noServiceEx}} \\
S & ::= & \varepsilon \mid x = P\ P,\ S \mid x = P.x,\ S \mid x = S,\ S \mid \textbf{\textit{Re}}(S;\ P;\ P)
\end{array}
$$

The domain $\boldsymbol{P}$ of functional expressions is just a specialization and extension of form expressions $\boldsymbol{E}$. The empty expression ε, identifiers x, form extension $P \cdot P$, and projection $P.x$ have identical semantics as in form expressions $\boldsymbol{E}$. Singleton bindings are expressed by $x = P$, which is just an abbreviation for the binding $x = P,\ \varepsilon$ as it is defined in $\boldsymbol{E}$. We use $\backslash x{:}\ E^*$ to denote service definition, where the identifier x is the service argument and the lazy form expression E^* specifies the service body. The substitution $\textbf{\textit{Re}}(P_1;\ P_x;\ P_y)$ means that free side effect identifiers $y_i \in \boldsymbol{Y}$ in P_1 are replaced by projections on P_y and free identifiers $x \notin \boldsymbol{Y}$ in P_1 are replaced by the values of bindings $x = P$ in P_x (if available). Finally, we use ***projectEx*** and ***noServiceEx*** to denote an illegal projection (i.e. identifier cannot be found) and an illegal service application (i.e. functor does not specify a service), respectively. Note that $\boldsymbol{P}$ does neither contain *binding (x = E, E)* nor *sandbox (root = root, E)*, and as a consequence, evaluation order does not matter.

For convenience, we sometime use $\boldsymbol{AP}$ to denote the domain of *atomic functional expressions*. An atomic functional expression AP is defined to be any functional expression that does not contain form extension.

The domain of side effect expressions $\boldsymbol{S}$ is also derived from form expressions $\boldsymbol{E}$. Thus, the empty expression ε, the composite expressions $x = P\ P,\ S$ and $x = P.x,\ S$ have identical semantics as in $\boldsymbol{E}$. The same holds for $x = S,\ S$. The substitution $\textbf{\textit{Re}}(S;\ P_x;\ P_y)$ has the same semantics as in $\boldsymbol{P}$, which means that free side effect identifiers $y_i \in \boldsymbol{Y}$ in S are replaced by projections on P_y and free identifiers $x \notin \boldsymbol{Y}$ in S are replaced by the values P of bindings $x = P$ in P_x (if available).

Finally, we define lazy forms F^* to be substitutions $\textbf{\textit{Re}}(P;\ F)$. A substitution $\textbf{\textit{Re}}(P;\ F)$ means that bindings in F are used to replace free identifiers x in P**.**

$$
F^* \ ::= \ \textbf{\textit{Re}}(P;\ F)
$$

Side effects

Evaluating a Piccola expression may cause side effects. These side effects are modeled with the domain $\boldsymbol{SE}$, the elements of which are defined as follows:

$$SE ::= \varepsilon \mid \textbf{\textit{effect}}_i \mid \textbf{\textit{projectEx}} \mid \textbf{\textit{noServiceEx}} \mid SE, SE$$

The expression ε denotes the empty side effect (i.e. no side effect) and we use $\textbf{\textit{effect}}_i$ to denote an arbitrary side effect. Side effects caused by an illegal projection and service application are denoted by ***projectEx*** (projection exception) and ***noServiceEx*** (no service exception), respectively. We use *SE, SE* to denote sequential composition of side effects.

5.4.2 Standard Piccola evaluation

In his Ph. D. thesis [9], Achermann formally defines the semantics of Piccola using the Piccola calculus, which is based on the polyadic π-calculus. In this thesis, we do not cover the Piccola calculus, and we represent the Piccola semantics with a semantic function $eval : \boldsymbol{E} \times \boldsymbol{F} \rightarrow \boldsymbol{F} \times \boldsymbol{SE}$. The equation *eval(E; R) = (F; SE)* means that we evaluate a form expression *E* in an environment *R*, which yields a resulting form *F* and causes some side effects *SE*.

Auxiliary functions

In Table 5.7, we inductively define the standard Piccola evaluation *eval* on the definition of form expressions *E*. Thereby, we use the auxiliary functions *project, service*, *apply* and *primApply* as defined in Table 5.6. Note that these definitions consist of several branches, and we always apply the first branch matching the argument structure. This rule is used for all the function definitions in this chapter.

The auxiliary function $project : \boldsymbol{F} \times \boldsymbol{X} \rightarrow \boldsymbol{F} \times \boldsymbol{SE}$ returns the value bound by a label. As an example, *project(x = 3 · x = 5 · y = 7; x)* yields *5*. According to the semantics of the polymorphic form extension (cf. Section 2.2.1), bindings of the form *G* override equally labeled bindings of the form *F* in the extended form *F · G*. Therefore, the definition of *project* treats extended forms from right to left. Note that *project(F; x)* yields the side effect ***projectEx*** when the label *x* cannot be found in the form *F*. In all the other cases, *project* is free from side effects.

The function $service : \boldsymbol{F} \rightarrow \boldsymbol{F}$ returns the service that is represented by a form. For instance, *service(x = 5 · **Cl**(x; E; F) · y = 7)* yields ***Cl**(x; E; F)*. Similar to *project*, the service of the form *G* overrides the service of the form *F* in the extended form *F · G*. If the form *F* does not contain a service, *service(F)* yields the empty form.

The function $apply : \boldsymbol{F} \times \boldsymbol{F} \rightarrow \boldsymbol{F} \times \boldsymbol{SE}$ takes two forms F_1 and F_2 as arguments and evaluates the application $F_1\ F_2$ which yields the resulting form and a side effect. The definition of apply consists of three cases: If the functor F_1 represents a Piccola service ***Cl**(x, E, F)*, we use the semantic function *eval* to evaluate the service body *E* in the environment *F* extended with the argument binding $x = F_2$. If the functor F_1 represents a primitive service, we use the auxiliary function

primApply : $\boldsymbol{F} \times \boldsymbol{F} \rightarrow \boldsymbol{F} \times \boldsymbol{SE}$ to execute it. We do not cover this function in detail here, and we just assume that it executes a primitive service (respectively an external service) and returns both the resulting form and the generated side effects. In the last case, if the functor F_1 does not represent a service, service application yields the empty form as the result and causes the side effect ***noServiceEx***.

$$
\begin{aligned}
project(F_1 \cdot x = F_2;\, x) &:= (F_2;\, \varepsilon) \\
project(F_1 \cdot AF;\, x) &:= project(F_1;\, x) \\
project(F;\, x) &:= (\varepsilon;\, \textbf{\textit{projectEx}}) \\
\\
service(F_1 \cdot \textbf{\textit{Cl}}(x;\, E;\, F)) &:= \textbf{\textit{Cl}}(x;\, E;\, F) \\
service(F_1 \cdot \textbf{\textit{primitive}}_i) &:= \textbf{\textit{primitive}}_i \\
service(F_1 \cdot AF) &:= service(F_1) \\
service(F) &:= \varepsilon \\
\\
apply(F_1;\, F_2) &:= \begin{cases} eval(E;\, F \cdot x = F_2) & \text{if } \textbf{\textit{Cl}}(x;\, E;\, F) = service(F_1) \\ primApply(\textbf{\textit{primitive}}_i;\, F) & \text{if } \textbf{\textit{primitive}}_i = service(F_1) \\ (\varepsilon;\, \textbf{\textit{noServiceEx}}) & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 5.6: Auxiliary functions used to define the Piccola semantics

The semantic function

Table 5.7 shows the semantic function *eval*, which evaluates a Piccola expression and yields a resulting form and a side effect. Evaluation of the empty expression ε is trivial, and it yields the empty form as a result and does not cause side effects. An identifier x is looked up in the environment R, and therefore, evaluation of x is put down to evaluation of the projection ***root***.x. The expression ***root*** denotes the current environment (static namespace), and therefore, evaluation of ***root*** yields the environment R and does not cause side effects. For a form extension $E_1 \cdot E_2$, we evaluate both E_1 and E_2 and return the polymorphic extension of the resulting forms and the sequential composition of the side effects. A service definition $\backslash x\text{: } E$ is turned into a closure ***Cl***$(x,\, E,\, R)$, that consists of the service argument x, the body of the service E and the current environment R, which is used as the static namespace when the closure is applied (cf. Table 5.6). In case of a binding $x = E_1,\, E_2$, we first evaluate the expression E_1 in the environment R, which yields a form F_1 and side effects SE_1. Then, we extend R with the binding $x = F_1$ and use this as the environment to evaluate E_2, which yields a form F_2 and side effects SE_2. Finally, we use the binding $x = F_1$ extended with the form F_2 as the resulting form and the sequential composition $SE_1,\, SE_2$ as the resulting side effect. Evaluation of a sandbox ***root*** $= E_1,\, E_2$ is very similar to evaluation of a binding. We evaluate E_1 and use the resulting form F_1 as the environment for the evaluation of E_2. This yields a form F_2 that is used as the final

result. In case of a projection $E.x$, we first evaluate the expression E, which yields $(F_1; SE_1)$. Then, we use the auxiliary function *project* (cf. Table 5.6) to retrieve the form F bound to the label x within the form F_1. Note that this projection causes a side effect SE, which is empty (ε) if the projection is successful. Finally, we return the form F and the sequential composition of the side effects SE_1 and SE. Evaluation of a service application $E_1\ E_2$ happens in three steps. First, we inductively evaluate the expression E_1, which yields $(F_1; SE_1)$. Second, we evaluate E_2, which yields $(F_2; SE_2)$. And third, we use the auxiliary function *apply* to evaluate the application $F_1\ F_2$, which yields a form F and causes side effects SE. The form F is then used as the resulting form and the sequential composition SE_1, SE_2, SE as the resulting side effect.

$eval(\varepsilon; R)$	:=	$(\varepsilon; \varepsilon)$		*(empty)*
$eval(x; R)$	:=	$eval(\mathbf{root}.x; R)$		*(identifier)*
$eval(\mathbf{root}; R)$	:=	$(R; \varepsilon)$		*(root)*
$eval(E_1 \cdot E_2; R)$	:=	$(F_1 \cdot F_2; SE_1, SE_2)$	$(F_1, SE_1) = eval(E_1; R)$ $(F_2, SE_2) = eval(E_2; R)$	*(extend)*
$eval(\backslash x\colon E; R)$	:=	$(\mathbf{Cl}(x; E; R); \varepsilon)$		*(service)*
$eval(x = E_1, E_2; R)$	:=	$(x = F_1 \cdot F_2; SE_1, SE_2)$	$(F_1, SE_1) = eval(E_1; R)$ $(F_2, SE_2) = eval(E_2; R \cdot x = F_1)$	*(bind)*
$eval(\mathbf{root} = E_1, E_2; R)$	:=	$(F_2; SE_1, SE_2)$	$(F_1, SE_1) = eval(E_1; R)$ $(F_2, SE_2) = eval(E_2; F_1)$	*(sandbox)*
$eval(E.x; R)$	:=	$(F; SE_1, SE)$	$(F_1, SE_1) = eval(E; R)$ $(F, SE) = project(F_1; x)$	*(project)*
$eval(E_1\ E_2; R)$	:=	$(F; SE_1, SE_2, SE)$	$(F_1, SE_1) = eval(E_1; R)$ $(F_2, SE_2) = eval(E_2; R)$ $(F; SE) = apply(F_1; F_2)$	*(apply)*

Table 5.7: Standard Piccola semantics

5.4.3 The partial evaluation algorithm

Using the domains specified above, we define our partial evaluation algorithm by means of the semantic function $split : \boldsymbol{E} \times \boldsymbol{P} \rightarrow \boldsymbol{E}^*$. The equation $split(E; R) = (P; S)$ means that we use the static namespace R to transform the form expression E into the lazy form expression $(P; S)$, which consists of the functional expression P and the side effect expression S. Note that this transformation may duplicate expressions. A real implementation should make sure that this does not affect performance and code size. For example, it may use pointers (references) to avoid duplication of expressions and cache the results of previously evaluated expressions (cf. Section 5.6).

In the rest of this section, we inductively define *split* on the definition of form expressions. Thereby, we use different meta-functions that are discussed and defined in Section 5.4.4. The meta-functions $projectP : \boldsymbol{P} \times \boldsymbol{X} \rightarrow \boldsymbol{P}$, $serviceP : \boldsymbol{P} \rightarrow \boldsymbol{P}$,

replaceP : $\boldsymbol{P} \times \boldsymbol{P} \times \boldsymbol{P} \rightarrow \boldsymbol{P}$ and *replaceS* : $\boldsymbol{S} \times \boldsymbol{P} \times \boldsymbol{P} \rightarrow \boldsymbol{S}$ statically evaluate a projection, a service selection, and a substitution, respectively. The meta-function *labelsP* : $\boldsymbol{P} \rightarrow \{\boldsymbol{X}\}$, returns a set of identifiers that are guaranteed to be bound in the argument expression.

The definition of *split* is shown in Table 5.8. Note that this is very similar to the definition of the standard Piccola evaluation *eval*, but instead of evaluating an expression, the function *split* statically separates it into a functional expression and a side effect expression.

Transformation of the empty expression ε is trivial and yields the empty expression for both the functional part and the side effect. An identifier x can be considered as a projection on the static namespace, and transformation of an identifier is therefore the same as transformation of a projection. The expression ***root*** returns the environment R as the functional part and has no side effect. In case of an extension $E_1 \cdot E_2$, we separately transform the expression E_1 and E_2, and return the form extension of the functional parts and sequential composition of the side effects. For a service definition $\backslash x\text{: } E$ we extend the environment R with the singleton binding $x = a_i$, where a_i denotes a fresh identifier[1] that represents the formal argument of the service. Then we transform the expression E in this new environment. The result $(P; S)$ of this transformation is then used in the service definition $\backslash a_i\text{: } (P, S)$, which is returned as the functional part. Obviously, a service definition does not cause side effect. In case of a binding $x = E_1, E_2$, we first transform the expression E_1, which yields a functional part P_1 and a side effect S_1. Then we extend the environment R with the singleton binding $x = P_1$ and use it to transform the expression E_2, which yields $(P_2; S_2)$. Finally, the functional part of the result is the extension of the binding $x = P_1$ with P_2, and the side effect is the sequential combination of S_1 and S_2. Transformation of a sandbox ***root*** = E_1, E_2 is similar to the transformation of a binding. We first transform the expression E_1, which yields a functional part P_1 and side effect S_1. Then we transform the expression E_2 in the environment P_1, which yields $(P_2; S_2)$. Finally, we return P_2 as the functional part and the sequential composition of S_1 and S_2 as the side effect part.

For a projection $E.x$, we first transform the expression E, which yields $(P; S)$. Now, there are two cases: If it can be statically verified that the label x is available in the functional expression P, we apply the projection *projectP(P, x)* in the functional part and return S as the side effect. Otherwise, we apply the projection in the side effect part. This means that the side effect part consists of the sequential composition of S and the projection *projectP(P; x)*, and the functional part just contains a reference to the result of the projection performed in the side effect part. Note that in the second

[1] A fresh identifier is an identifier that is not used in the service containing the current expression. In particular, this means that the identifier is not used in any nested service definitions within this service.

case, evaluating the projection in the side effect part is necessary because a projection results in a runtime exception (***projectEx***) if the identifier is not available.

$$
\begin{array}{rcll}
split(\varepsilon; R) & := & (\varepsilon; \varepsilon) & (empty) \\
split(x; R) & := & split(\mathbf{root}.x; R) & (identifier) \\
split(\mathbf{root}; R) & := & (R; \varepsilon) & (root) \\
split(E_1 \cdot E_2; R) & := & (P_1 \cdot P_2; S_1, S_2) \quad (P_1; S_1) = split(E_1; R),\ (P_2; S_2) = split(E_2; R) & (extend) \\
split(\backslash x\text{: } E; R) & := & (\backslash a_i\text{: } (P, S); \varepsilon) \quad (P; S) = split(E; R \cdot x = a_i) \text{ where } a_i \in \boldsymbol{X} \setminus \boldsymbol{Y} \text{ denote fresh identifiers} & (service) \\
split(x = E_1, E_2; R) & := & (x = P_1 \cdot P_2; S_1, S_2) \quad (P_1; S_1) = split(E_1; R),\ (P_2; S_2) = split(E_2; R \cdot x = P_1) & (bind) \\
split(\mathbf{root} = E_1, E_2; R) & := & (P_2; S_1, S_2) \quad (P_1; S_1) = split(E_1; R),\ (P_2; S_2) = split(E_2; P_1) & (sandbox) \\
split(E.x, R) & := & \begin{cases} (projectP(P; x); S) & \text{if } x \in labelsP(P) \\ (y_i; S, y_i = projectP(P; x)) & \text{otherwise} \end{cases} \quad \text{where } (P; S) = split(E; R) & (project) \\
split(E_1\ E_2; R) & := & \begin{cases} (replaceP(P; x = P_2; \varepsilon); S_1, S_2)) & \text{if } P_1' = \backslash x\text{: } (P, \varepsilon) \\ (replaceP(P; x = P_2; y_i); S_1, S_2, y_i = replaceS(S; x = P_2; \varepsilon)) & \text{if } P_1' = \backslash x\text{: } (P, S) \text{ and } S \neq \varepsilon \\ (y_i; S_1, S_2, y_i = P_1'\ P_2) & \text{otherwise} \end{cases} \\
 & & \text{where } (P_1; P_2) = split(E_1; R),\ (P_2; S_2) = split(E_2; R),\ P_1' = serviceP(P_1) \text{ and } y_i \in \boldsymbol{Y} \text{ denote fresh identifiers} & (apply)
\end{array}
$$

Table 5.8: The partial evaluation algorithm

The most complex part is service invocation $E_1\ E_2$, which consists of three different cases. In any case, we first transform the expressions E_1 and E_2 which yields $(P_1; S_2)$ and $(P_2; S_2)$ respectively. Then we apply service selection to retrieve the service P_1' represented by the functor P_1. The first case applies when this service has already been separated into $\backslash x\text{: } (P, S)$ and does not contain any side effects $(S = \varepsilon)$. In this case, we substitute the formal argument x in the functor P with the functional part of the argument (P_2), and we return this as the functional part. The side effect consists of the sequential composition of the side effects of the functor (S_1) and the argument (S_2). This means that we evaluate the application in the functional part and that the side effect part only consists of the side effects resulting from evaluating E_1 and E_2. The second case applies when the service P_1' is a service $\backslash x\text{: } (P; S)$ with a side effect that is not empty ($S \neq \varepsilon$). In this case, the substitution in the functional part of the first case is extended with a fresh identifier $y_i \in \boldsymbol{Y}$, which refers to the result of the side effects. At the same time, the side effect part of the first case is extended with an expression that binds the form returned by evaluating the side effect to the label y_i.

This form is the substitution of the formal argument x with the functional part of the concrete argument (P_2) in the side effect S. Finally, if the service P_1' is a primitive service or if we do not have enough static information to determine the structure of this service, we bind the application of the functor P_1' with the functional part of the argument P_2 to the fresh label y_i and append this to the side effects of the functor (S_1) and the argument (S_2). As the functional part, we return the label y_i, which refers to the side effect.

5.4.4 The meta-functions

In this section, we discuss and define the meta-functions used by the function *split*. The functions *projectP*, *serviceP*, *replaceP* and *replaceS* statically evaluate a projection, a service selection and a substitution. This means that they simplify expressions already at compile time (i.e. during the partial evaluation). This is valuable for two reasons: First, it allows us to determine the structure of the resulting forms already at compile time, which is necessary for an effective usage of our partial evaluation algorithm[1]. And second, it decreases the number of operations that have to be executed at runtime.

At the same time, static evaluation influences the resulting code and especially the code size. As an example, a statically evaluated substitution may increase the code size if the substituted expression is large and occurs multiple times. Therefore, a concrete implementation of these meta-functions depends on the structure of the Piccola scripts in memory and the requirements of the user (performance vs. code size). Note that it is possible to perform no static evaluation at all and use the unevaluated projection, the unevaluated service selection[2] and the unevaluated substitution instead:

$$
\begin{aligned}
\mathit{projectP}(P; x) &:= P.x \\
\mathit{serviceP}(P) &:= P \\
\mathit{replaceP}(P; P_x; P_y) &:= \mathbf{Re}(P; P_x; P_y) \\
\mathit{replaceS}(S; P_x; P_y) &:= \mathbf{Re}(S; P_x; P_y)
\end{aligned}
$$

The last meta-function is the function *labelsP*, which returns the set of identifiers that are guaranteed to be bound in the argument expression. This is useful for splitting a projection into a side effect and a referentially transparent part because we can anticipate whether the projection might yield a runtime exception (cf. rule *project* in Table 5.8).

---

[1] As an example, consider the rule *apply* in Table 5.8. There, the structure of the functor is used to determine which of the three cases applies.

[2] Evaluation of a service application implicitly selects the service represented by the functor. Therefore, an unevaluated service selection can be expressed with the original form itself.

In the following, we present the definitions of these meta-functions. For simplicity and conciseness, we present definitions that are relatively easy to define and understand. However, they may not lead to optimal performance and code size.

Projection

The function $projectP : \boldsymbol{P} \times \boldsymbol{X} \rightarrow \boldsymbol{P}$ simplifies a projection on a certain label of a functional expression. For instance $projectP(P_1 \cdot x = P_2; x)$ yields P_2. If the result of the projection cannot be determined, an unevaluated projection $P.x$ is returned. *projectP* is inductively defined on the definition of functional expressions P as follows:

$$\begin{aligned} projectP(\varepsilon, x) &:= \boldsymbol{projectEx} \\ projectP(P_1 \cdot x = P_2; x) &:= P_2 \\ projectP(P_1 \cdot AP, x) &:= projectP(P_1, x) \\ projectP(P, x) &:= P.x \end{aligned}$$

In order to effectively transform projections, we also use the function $labelsP : \boldsymbol{P} \rightarrow \{\boldsymbol{X}\}$, which returns the identifiers that are guaranteed to be bound in an expression. *labelsP* is inductively defined as follows:

$$\begin{aligned} labelsP(x = P) &:= \{x\} \\ labelsP(\boldsymbol{Re}(P; P_x; P_y)) &:= labelsP(P) \\ labelsP(P_1 \cdot P_2) &:= labelsP(P_1) \cup labelsP(P_2) \\ labelsP(P) &:= \varnothing \end{aligned}$$

Service selection

The function $serviceP : \boldsymbol{P} \rightarrow \boldsymbol{P}$ simplifies service selection in a functional expression. For instance $serviceP(P_1 \cdot \backslash x: (P_2, S))$ yields $\backslash x: (P_2, S)$. If the result of the service selection cannot be determined, the argument expression P itself is returned. *serviceP* is inductively defined on the definition of functional expressions P:

$$\begin{aligned} serviceP(\varepsilon) &:= \boldsymbol{noServiceEx} \\ serviceP(P \cdot \backslash x: (P; S)) &:= \backslash x: (P; S) \\ serviceP(P \cdot \boldsymbol{primitive_i}) &:= \boldsymbol{primitive_i} \\ serviceP(P \cdot AP) &:= serviceP(P) \\ serviceP(P) &:= P \end{aligned}$$

Substitution

The function $replaceP : \boldsymbol{P} \times \boldsymbol{P} \times \boldsymbol{P} \rightarrow \boldsymbol{P}$ is used to substitute identifiers in functional expressions. A substitution $replaceP(P_1; P_x; P_y)$ means that free side effect identifiers $y_i \in \boldsymbol{Y}$ in P_1 are replaced by projections on P_y and free identifiers $x \notin \boldsymbol{Y}$ in P_1 are replaced by the values of bindings $x = P$ in P_x (if available). Note that substitutions $replaceP(P; P_x; P_y)$ are performed atomically, which means that only the identifiers

that appear in the original expression P are substituted. As an example, we consider the expression $replaceP(x_1 \cdot x_2 \cdot y_1 \cdot y_2;\ x_1 = x_3 \cdot x_3 = P;\ y_3)$ with $x_i \in \boldsymbol{X} \setminus \boldsymbol{Y}$ and $y_i \in \boldsymbol{Y}$. Because of the binding $x_1 = x_3$ in P_x, the identifier x_1 gets replaced with x_3. Since P_x does not contain a binding with the label x_2, the free identifier x_2 in P remains as it is. Furthermore, both side effect identifiers y_1 and y_2 are replaced by projections on y_3. Thus, the result of the substitution is $x_3 \cdot x_2 \cdot y_3.y_1 \cdot y_3.y_2$.

In the following, we define the function $replaceP(P;\ P_x;\ P_y)$ on the definition of functional expressions P:

A free side effect identifier $y_i \in \boldsymbol{Y}$ is substituted with a projection on the identifier y_i of argument P_y. If the substitution is applied to a free identifier $x \notin \boldsymbol{Y}$ and there is a binding for the label x in the expression P_x, we use *projectP* to replace x with the value of this binding. If there is no such binding in P_x, the general case (last line of the definition) applies, and the identifier x remains as it is. In case of a projection $P.x$, we first apply the substitution to P and perform the projection afterwards. Similarly, if the substitution is applied to a binding $x = P$, we apply the substitution to the expression P and bind the result to the identifier x. In case of a service $\backslash x:\ (P;\ S)$, we apply an unevaluated substitution to both the functional part P and the side effect part S. A nested substitution $replaceP(\boldsymbol{Re}(P;\ P_{x1};\ P_{y1});\ P_{x2};\ P_{y2})$ is transformed into a single substitution by combining the expressions P_{x1} and P_{x2} respectively P_{y1} and P_{y1} with form extension. If the substitution is applied to an extension $P_1 \cdot P_2$, we perform the substitutions for both P_1 and P_2 and use form extension to combine the results. Finally, if no other branch of the definition applies, the substitution *replaceR* returns the argument expression P.

$$
\begin{aligned}
replaceP(y_i;\ P_x;\ P_y) &:= P_y.y_i && y_i \in \boldsymbol{Y} \\
replaceP(x;\ P_x;\ P_y) &:= projectP(P_x;\ x) && x \notin \boldsymbol{Y} \text{ and } x \in labels(P_x) \\
replaceP(P.x;\ P_x;\ P_y) &:= replaceP(P;\ P_x;\ P_y).x \\
replaceP(x = P;\ P_x;\ P_y) &:= x = replaceP(P;\ P_x;\ P_y) \\
replaceP(\backslash x:\ (P;\ S);\ P_x;\ P_y) &:= \backslash x:\ (\boldsymbol{Re}(P;\ P_x;\ P_y);\ \boldsymbol{Re}(S;\ P_x;\ P_y)) \\
replaceP(\boldsymbol{Re}(P;\ P_{x1};\ P_{y1});\ P_{x2};\ P_{y2}) &:= replaceP(P;\ P_{x1} \cdot P_{x2};\ P_{y1} \cdot P_{y2}) \\
replaceP(P_1 \cdot P_2;\ P_x;\ P_y) &:= replaceP(P_1;\ P_x;\ P_y) \cdot replaceP(P_2;\ P_x;\ P_y) \\
replaceP(P;\ P_x;\ P_y) &:= P
\end{aligned}
$$

The function $replaceS : \boldsymbol{S} \times \boldsymbol{P} \times \boldsymbol{P} \rightarrow \boldsymbol{S}$ is used to substitute identifiers in side effect expressions. The semantics of this substitution is identical to the one of *replaceP*, and therefore, the definition looks as follows:

$$
\begin{aligned}
replaceS(\varepsilon;\ P_x;\ P_y) &:= \varepsilon \\
replaceS(y = P_1\ P_2,\ S;\ P_x;\ P_y) &:= y = replaceP(P_1;\ P_x;\ P_y)\ replaceP(P_2;\ P_x;\ P_y),\ replaceS(S;\ P_x;\ P_y) \\
replaceS(y = P_1.z,\ S;\ P_x;\ P_y) &:= y = replaceP(P_1,\ P_x;\ P_y).z,\ replaceS(S;\ P_x;\ P_y) \\
replaceS(y = S_1,\ S_2;\ P_x;\ P_y) &:= y = replaceS(S_1,\ P_x;\ P_y).z,\ replaceS(S_2;\ P_x;\ P_y) \\
replaceS(\boldsymbol{Re}(S;\ P_{x1};\ P_{y1});\ P_{x2};\ P_{y2}) &:= replaceS(S;\ P_{x1} \cdot P_{x2};\ P_{y1} \cdot P_{y2})
\end{aligned}
$$

5.4.5 Evaluating the side effects and the functional part

After defining the partial evaluation algorithm *split* in the previous section, we complete the specification of our lazy evaluation strategy by extending the Piccola evaluation *eval* so that we can use it to evaluate side effect expressions S and functional expressions P.

In Table 5.7, we have defined the semantic function *eval* for form expressions $\boldsymbol{E}$. Now, we extend this function and define it for $\boldsymbol{E} \cup \boldsymbol{P} \cup \boldsymbol{S} \cup \boldsymbol{F}^*$. Since $\boldsymbol{P}$ and $\boldsymbol{S}$ are derived from $\boldsymbol{E}$ (cf. Section 5.4.1) and $\boldsymbol{F}^*$ contains only of a single element ($\boldsymbol{Re}(P; F)$), this extension is simple and consists of the following additional rules:

$$
\begin{array}{rcll}
eval(\boldsymbol{Re}(P; P_x; P_y); R) & := & eval(P; R \cdot F_x \cdot F_y) & (F_x, \varepsilon) = eval(P_x; R) \\
 & & & (F_y, \varepsilon) = eval(P_y; R) \\
eval(\boldsymbol{projectEx}; R) & := & (\varepsilon; \boldsymbol{projectEx}) & \\
eval(\boldsymbol{noServiceEx}; R) & := & (\varepsilon; \boldsymbol{noServiceEx}) & \\
eval(\backslash x: (P; S); R) & := & (\boldsymbol{Cl}^*(x, P, S); \varepsilon) & \\
eval(\boldsymbol{Re}(S; P_x; P_y); R) & := & eval(S; R \cdot F_x \cdot F_y) & (F_x, \varepsilon) = eval(P_x; R) \\
 & & & (F_x, \varepsilon) = eval(P_x; R) \\
eval(Re(P; F); R) & := & eval(P; R \cdot F) &
\end{array}
$$

The domain $\boldsymbol{P}$ contains the expressions $\boldsymbol{Re}(P; P_x; P_y)$, $\boldsymbol{projectEx}$, $\boldsymbol{noServiceEx}$, and $\backslash x: (P; S)$ that are not in $\boldsymbol{E}$. Evaluation of a substitution $\boldsymbol{Re}(P; P_x; P_y)$ happens in two steps. First, we inductively evaluate the expressions P_x and P_y in the environment R, which yields forms F_x, F_y and causes no side effects. Then, we extend R with the resulting forms and use this as the environment to evaluate P. Due to the definition of *eval*, a free identifier x in P is now automatically substituted with the form bound to a label x in F_x respectively F_y. If both F_x and F_y do not contain the label x, the identifier is looked up in the original environment R.[1] Evaluation of the expressions $\boldsymbol{projectEx}$ and $\boldsymbol{noServiceEx}$ are trivial, and we just return the corresponding side effects. Finally, evaluation of a lazy service $\backslash x: (P; S)$ yields a closure $\boldsymbol{Cl}^*(x, P, S)$ that consists of the service argument x, the functional expression P and the side effect expression S. Since references to the environment are already resolved in P and S, the environment R is not used. Note that the original definition of forms F (cf. Section 5.4.1) only defines the closures $\boldsymbol{Cl}(x, E, F)$, which represent services with an argument x, a body E and an environment F. Thus, we need to extend our definition of forms F with the lazy closure $\boldsymbol{Cl}^*(x, P, S)$.

For the domain $\boldsymbol{S}$ we only have to define evaluation of the substitution $\boldsymbol{Re}(S; P_x; P_y)$. (All the other expressions are evaluated using the rules for $\boldsymbol{E}$ and $\boldsymbol{P}$). The substitution $\boldsymbol{Re}(S; P_x; P_y)$ is evaluated very similarly to the substitution $\boldsymbol{Re}(P; P_x; P_y)$ defined above.

---

[1] Due to the definition of the service *split* (cf. Table 5.8), F_y only contains bindings $y_i = P$ ($y_i \in \boldsymbol{Y}$) and F_x only contains bindings $x = P$ ($x \notin \boldsymbol{Y}$). Therefore it does not matter whether we use the extension $F_x \cdot F_y$ or the extension $F_y \cdot F_x$.

First, the functional expressions P_x and P_y are evaluated in the original environment, and then the side effect expression S is evaluated in the extended environment.

Finally, we have to define *eval* for the substitution ***Re****(P; F)* in ***F*.*** Since the second argument F is already a form, we directly extend the environment R with this form and use the result as the environment to evaluate P.

Using the extended version of *eval*, we can now formally specify how an application of the service $\backslash x\text{: } E$ is evaluated lazily: At compile time, we use the function *split* to transform this service into a lazy service $\backslash a\text{: } (P; S)$. At runtime, when this service is invoked with an argument F, we use *eval* to evaluate the side effect expression S in the environment $a = F$, which yields a form F_s and causes some side effects SE. Using the form F_s, we then build the lazy form $\boldsymbol{Re}(P; F_s \cdot a = F)$, which contains no free identifiers and can be evaluated when it is effectively needed.

For convenience, we define the function $lazyApply : \boldsymbol{E} \times \boldsymbol{F} \rightarrow \boldsymbol{F} \times \boldsymbol{SE}$, which takes a service $\backslash x\text{: } E$ and an argument form F and performs a lazy application. The return value consists of the resulting form and the triggered side effects.

$$
\begin{array}{lll}
lazyApply(\backslash x\text{: } E; F) := (F_P, SE) & \backslash a\text{: } (P; S) = split(\backslash x\text{: } E; \varepsilon) & \textit{[compile time]} \\
 & (F_s; SE) = eval(S; a = F) & \textit{[invocation time]} \\
 & (F_P; \varepsilon) = eval(\boldsymbol{Re}(P; FS \cdot a = F); \varepsilon) & \textit{[utilization time]}
\end{array}
$$

5.4.6 Examples

In this section, we apply our lazy evaluation technique to a few examples. These examples show how the function *split* transforms a service into a lazy service and how such a service is evaluated.

Note that we sometimes simplify the resulting terms by removing unnecessary occurrences of the empty expression respectively the empty form. This means that we use the identities $U \cdot \varepsilon = \varepsilon \cdot U = U$ and $V, \varepsilon = \varepsilon, V = V$, where $U \in \boldsymbol{E} \cup \boldsymbol{P} \cup \boldsymbol{S} \cup \boldsymbol{F} \cup \boldsymbol{F}^*$ and $V \in \boldsymbol{E} \cup \boldsymbol{S}$ [9].

Example 1 — A simple script

In this example, we apply our lazy evaluation strategy to the simple script *s*. This script takes an argument *r*, which is then used as the static namespace. Then it invokes a service *e* and binds the resulting form to the label *k*. Note that the service *e* is looked up in the argument *r*. Therefore, its structure cannot be determined at compile time and may cause side effects. Finally, we bind the value represented by the identifier *k* to the label *l*:

s(r):
  root *= r*
  k = e()
  l = k

First, we transform this script into a lazy form expression $\backslash a_1$*: (P; S)* by applying the function *split* with the empty environment ε. In Table 5.9, we show this transformation with all the details.

Note that we use the definition of *split* in Table 5.8 and indicate the applied rule on the right hand side of each line. Usually, evaluation of a rule involves application of other rules. Therefore, we use bold headers and indentation to indicate nesting of rules. Furthermore, we use numbers to refer to different applications of rules. As an example, all the lines marked with *binding1* are used to evaluate a single application of the rule *binding*, whereas the lines marked with *binding2* are used to evaluate a different binding expression.

Collecting all the resulting expressions in Table 5.9 yields the following result for P and S:

$(P; S) = (P_2; S_1, S_2) = (k = P_4 \cdot P_5; \varepsilon, S_4, S_5) = (k = y_1 \cdot l = P_8 \cdot P_9; \varepsilon, y_1 = a_1.e\ \varepsilon, S_8, S_9)$
$= (k = y_1 \cdot l = y_1 \cdot \varepsilon; \varepsilon, y_1 = a_1.e\ \varepsilon, \varepsilon, \varepsilon) = (k = y_1 \cdot l = y_1; y_1 = a_1.e\ \varepsilon)$

This means that the service s is transformed into the lazy form expression $\backslash a_1$*: (*$k = y_1 \cdot l = y_1; y_1 = a_1.e\ \varepsilon$*)*. Note that the functional part $k = y_1 \cdot l = y_1$ is free from side effects and that the only free identifier y_1 refers to the result of the side effects. If we invoke this service with an argument form $e =$ ***primitive*****$_1$**, we first evaluate the side effect expression. As the initial environment we use the form $a_1 =$ *(e =* ***primitive*****$_1$***)* that binds the invocation argument to the formal argument a_1 of the service (cf. Section 5.4.5). This evaluation yields a form $y_1 = F_1$ as the result and causes a side effect ***effect*****$_1$**:

$eval(y_1 = a_1.e\ \varepsilon; a_1 = (e =$ ***primitive*****$_1$***)) = eval(*$y_1 = (e =$ ***primitive*****$_1$***).e*$\ \varepsilon; \varepsilon)$
$=$ *eval(****primitive*****$_1\ \varepsilon; \varepsilon) = (y_1 = F_1;$** ***effect*****$_1$***)* *where* $(F_1;$ ***effect*****$_1$***) = primApply(****primitive*****$_1; \varepsilon)$**

In the next step, we use the resulting form $y_1 = F_1$ of the side effect evaluation to create the lazy form ***Re(****$k = y_1 \cdot l = y_1; y_1 = F_1 \cdot a_1 =$* *(e =* ***primitive*****$_1$***))***. This form represents the result of the service application and since it is free from side effects it can be evaluated when it is effectively needed. Note that this form is a closure, since all the free identifiers are replaced when the substitution is applied. Therefore, we evaluate it in the empty environment ε. This yields the resulting form $k = F_1 \cdot l = F_1$ and does not cause side effects. (The side effect ***effect*****$_1$** was already generated by evaluating the side effect expression).

*eval(****Re****(*$k = y_1 \cdot l = y_1; y_1 = F_1 \cdot a_1 =$ *(e =* ***primitive*****$_1$***)); ε)*
$=$ *eval(*$k = y_1 \cdot l = y_1; \varepsilon \cdot y_1 = F_1 \cdot a_1 =$ *(e =* ***primitive*****$_1$***))***
$=$ *eval(*$k = (\varepsilon \cdot y_1 = F_1 \cdot a_1 =$ *(e =* ***primitive*****$_1$***)).*$y_1 \cdot l = (\varepsilon \cdot y_1 = F_1 \cdot a_1 =$ *(e =* ***primitive*****$_1$***)).*$y_1; \varepsilon)$
$= (k = F_1 \cdot l = F_1; \varepsilon)$

$split(s; \varepsilon) = split(\backslash r\text{: } (\textbf{root} = r,\ k = e\ \varepsilon,\ l = k);\ \varepsilon) = (\backslash a_1\text{: } (P; S);\ \varepsilon)$ *(service1)*

- ***service1:***
  $(P; S) = split(root = r,\ k = e\ \varepsilon,\ l = k;\ \varepsilon \cdot r = a_1)$ *(service1)*
  $= (P_2; S_1, S_2)$ *(sandbox1)*
  - ***sandbox1:***
    $(P_1; S_1) = split(r;\ \varepsilon \cdot r = a_1)$ *(sandbox1)*
    $= split(\textbf{root}.r;\ \varepsilon \cdot r = a_1)$ *(identifier1)*
    - ***project1:***
      $(P_3, S_3) = split(\textbf{root};\ \varepsilon \cdot r = a_1)$ *(project1)*
      $= (\varepsilon \cdot r = a_1;\ \varepsilon)$ *(root)*
      Since $r \in labelsP(\varepsilon \cdot r = a_1)$:
      $(P_1; S_1) = (projectP(P_3; r); S_3) = (projectP(\varepsilon \cdot r = a_1; r);\ \varepsilon) = (a_1;\ \varepsilon)$ *(project1)*
    $(P_2; S_2) = split(k = e\ \varepsilon,\ l = k;\ P_1) = split(k = e\ \varepsilon,\ l = k;\ a_1)$ *(sandbox1)*
    $= (k = P_4 \cdot P_5;\ S_4, S_5)$ *(binding1)*
    - ***binding1:***
      $(P_4; S_4) = split(e\ \varepsilon;\ a_1)$ *(binding1)*
      - ***app1:***
        $(P_6; S_6) = split(e;\ a_1)$ *(app1)*
        $= split(\textbf{root}.e;\ a_1) = \ldots = (a_1.e;\ \varepsilon)$ *(identifier2)*
        $(P_7; S_7) = split(\varepsilon;\ a_1)$ *(app1)*
        $= (\varepsilon;\ \varepsilon)$ *(empty)*
        Since $serviceP(P_6) = serviceP(a_1.e) = a_1.e$:
        $(P_4; S_4) = (y_1;\ S_6, S_7,\ y_1 = serviceP(P_6)\ \varepsilon) = (y_1;\ y_1 = a_1.e\ \varepsilon)$ *(app1)*
      $(P_5; S_5) = split(l = k;\ a_1 \cdot k = P_4) = split(l = k,\ \varepsilon;\ a_1 \cdot k = y1)$ *(binding1)*
      $= (l = P_8 \cdot P_9;\ S_8, S_9)$ *(binding2)*
    - ***binding2:***
      $(P_8; S_8) = split(k;\ a_1 \cdot k = y_1)$ *(binding2)*
      $= split(root.k;\ a_1 \cdot k = y_1) = \ldots = (y_1;\ \varepsilon)$ *(identifier3)*
      $(P_9; S_9) = split(\varepsilon;\ a_1 \cdot k = y_1 \cdot l = P_8) = split(\varepsilon;\ a_1 \cdot k = y_1 \cdot l = y_1)$ *(binding2)*
      $= (\varepsilon;\ \varepsilon)$ *(empty)*

Table 5.9: Transformation of a simple script

Example 2 — Multiple services

In this example, we consider a script s that defines two services f and g. The service f is similar to the service s used in the previous example, and it contains the application of a service e, which is not known at compile time and may cause side effects. The service g consists of two invocations of the service f with the arguments x respectively v.

```
s(r):
  root = r
  f(x):
    k = e x
    l = k
  g(x):
    m = f x
    n = f u
```

In the following, we apply the transformation *split* to this service *s*. We do not show all the details and mainly focus on the applications of the service *f* in the service *g*.

Transformation of $split(s; \varepsilon)$ finally yields a lazy form expression $\backslash a_s\text{: } (P_s, S_s)$. As a consequence of the sandbox (***root*** = *r*) in the first line, the definition of the service *f* is transformed in the environment a_s (a_s is the formal argument of the transformation of *s*). Thus, transformation of the service *f* looks as follows:

$split(\backslash x\text{: } (k = e\ x,\ l = k);\ a_s) = \backslash a_f\text{: } (P_f;\ S_f)$
$(P_f;\ S_f) = split(k = e\ x,\ l = k;\ a_s \cdot x = a_f) = (k = y_1 \cdot l = y_1;\ y_1 = a_s.e\ a_f)$

As a consequence of the rule *binding*, the transformation $\backslash a_f\text{: } (P_f;\ S_f)$ of the service *f* is appended to the environment when the service *g* is transformed. Thus, applications of the service *f* are transformed according to the second case of the rule *apply*. In the resulting side effect expression, the formal argument a_f of the service *f* is substituted with the concrete arguments by applying *replaceS*. In the functional expression, we use *replaceP* to substitute the formal arguments and to indicate the identifier of the side effect. Note that the two invocations of *f* cause different side effects, which are bound to the labels y_2 and y_3 in the side effect expressions and referenced within the functional expression.

$split(\backslash x\text{: } (m = f\ x,\ n = f\ u);\ a_s \cdot f = \backslash a_f\text{: } (P_f,\ S_f)) = \backslash a_g\text{: } (P_g;\ S_g)$
$(P_g;\ S_g) = split(m = f\ x,\ n = f\ u;\ a_s \cdot f = \backslash a_f\text{:}(P_f,\ S_f) \cdot x = a_g)$
$= (m = replaceP(P_f;\ a_f = a_g;\ y_2) \cdot n = replaceP(P_f;\ a_f = a_s.u;\ y_3);$
$y_2 = replaceS(S_f;\ a_f = a_g;\ \varepsilon),\ y_3 = replaceS(S_f;\ a_f = a_s.u;\ \varepsilon))$
$= (m = replaceP(k = y_1 \cdot l = y_1;\ a_f = a_g;\ y_2) \cdot n = replaceP(k = y_1 \cdot l = y_1;\ a_f = a_s.u;\ y_3);$
$y_2 = replaceS(y_1 = a_s.e\ a_f;\ a_f = a_g;\ \varepsilon),\ y_3 = replaceS(y_1 = a_s.e\ a_f;\ a_f = a_s.u;\ \varepsilon))$
$= (m = (k = y_2.y_1 \cdot l = y_2.y_1) \cdot n = (k = y_3.y_1 \cdot l = y_3.y_1);$
$y_2 = (y_1 = a_s.e\ a_g),\ y_3 = (y_1 = a_s.e\ a_s.u))$

If we execute the expression $s(e = \textbf{\textit{primitive}}_1 \cdot u = 3).g(5)$, we first evaluate the side effect expression S_g in the environment that provides the function arguments:

$eval(y_2 = (y_1 = a_s.e\ a_g),\ y_3 = (y_1 = a_s.e\ a_s.u);\ a_s = (e = \textbf{\textit{primitive}}_1 \cdot u = 3) \cdot a_g = 5))$
$= (y_2 = (y_1 = F_1) \cdot y_3 = (y_1 = F_2);\ \textbf{\textit{effect}}_1,\ \textbf{\textit{effect}}_2)$ *where* $(F_1;\ \textbf{\textit{effect}}_1) = eval(\textbf{\textit{primitive}}_1\ 5)$ *and* $(F_2;\ \textbf{\textit{effect}}_2) = eval(\textbf{\textit{primitive}}_1\ 3)$

Then, we use the resulting form to create the lazy form representing the functional part of the executed expression:

$\textbf{\textit{Re}}(m = (k = y_2.y_1 \cdot l = y_2.y_1) \cdot n = (k = y_3.y_1 \cdot l = y_3.y_1);\ a_g = 5 \cdot (y_2 = (y_1 = F_1) \cdot y_3 = (y_1 = F_2)))$

When we finally use this part, we evaluate it in the empty environment. This yields the resulting form $m = (k = F_1 \cdot l = F_1) \cdot n = (k = F_2 \cdot l = F_2)$ and does not cause side effects. (The side effects $\boldsymbol{effect_1}$ and $\boldsymbol{effect_2}$ were already generated by executing the side effect expression)

$eval(\mathbf{Re}(m = (k = y_2.y_1 \cdot l = y_2.y_1) \cdot n = (k = y_3.y_1 \cdot l = y_3.y_1);\ a_g = 5 \cdot (y_2 = (y_1 = F_1) \cdot y_3 = (y_1 = F_2)));\ \varepsilon)$
$= eval(m = (k = y_2.y_1 \cdot l = y_2.y_1) \cdot n = (k = y_3.y_1 \cdot l = y_3.y_1);\ a_g = 5 \cdot (y_2 = (y_1 = F_1) \cdot y_3 = (y_1 = F_2))$
$= (m = (k = F_1 \cdot l = F_1) \cdot n = (k = F_2 \cdot l = F_2);\ \varepsilon)$

Example 3 — Higher order service application

The following script s contains a higher order service application, which means that it invokes a Curried service e with an argument u and then it invokes the resulting service with an argument v.

$s(r)$:
  $\mathbf{root} = r$
  $k = e\ u$
  $l = k\ v$

Applying the transformation *split* to this service yields a lazy service $\backslash a\text{: } (P;\ S)$. Because of the sandbox expression $\mathbf{root} = r$ in the first line, the expression $k = e\ u,\ l = k\ v$ is evaluated in the environment a, which refers to the formal service argument.

$split(\backslash r\text{: } (\mathbf{root} = r,\ k = e\ u,\ l = k\ v);\ \varepsilon) = (\backslash a\text{: } (P;\ S);\ \varepsilon)$
$(P;\ S) = split(\mathbf{root} = r,\ k = e\ u\ v,\ l = k\ v;\ r = a) = \ldots = split(k = e\ u,\ l = k\ v;\ a)$

According to the rule *binding*, transforming the expression $k = e\ u,\ l = k\ v$ means transforming the expression $k = e\ u$ followed by transforming $l = k\ v$ in the extended environment. Transformation of $e\ u$ and $k\ v$ is done by applying the third case of the rule *apply* in Table 5.8.

$split(k = e\ u;\ a) = split(k = a.e\ a.u;\ \varepsilon) = (k = y1;\ y1 = a.e\ a.u)$
$split(l = k\ v;\ a \cdot k = y_1) = split(l = y_1\ a.v) = (l = y_2;\ y_2 = y_1\ a.v)$

Altogether, we get the following result for the transformation of s:

$split(s;\ \varepsilon) = (\backslash a\text{: } (k = y_1 \cdot l = y_2;\ y_1 = a.e\ a.u,\ y_2 = y_1\ a.v);\ \varepsilon)$

If we invoke this service with an argument form $e = \boldsymbol{primitive_1}$, $u = 1$, $v = 2$, we first evaluate the side effect expression with the interpreter *eval*. Note that the second binding $y_2 = y_1\ a.v$ of the side effect refers to the first one and that this reference is properly resolved:

$eval(y_1 = a.e\ a.u,\ y_2 = y_1\ a.v;\ a = (e = \boldsymbol{primitive_1},\ u = 1,\ v = 2))$
$= \ldots = (y_1 = F_1 \cdot y_2 = F_2;\ \boldsymbol{effect_1}, \boldsymbol{effect_2})$ *where* $(F_1;\ \boldsymbol{effect_1}) = eval(\boldsymbol{primitive_1}\ 1)$ *and* $(F_2;\ \boldsymbol{effect_2}) = eval(F_1\ 2)$

Now, evaluating the functional part of the service is trivial and yields the resulting form $k = F_1 \cdot l = F_2$.

5.5 How to prove correctness

A prototype implementation in SPiccola (cf. Appendix A) gives us evidence that our lazy evaluation strategy is behavior preserving, but we have not carried out a formal proof yet. In this section, we are discussing what it means to formally prove it and how such a proof could look like.

Considering the diagram in Figure 5.3, we see that there are two ways of evaluating a Piccola service $\backslash x: E$ with an argument form F_a. On the one hand, we can apply the standard Piccola evaluation *eval* (respectively *apply*), which invokes the service and yields a resulting form F and some side effects SE. On the other hand, we can evaluate the service using our lazy evaluation strategy. This means that at compile time, we use the partial evaluation function *split* to transform the service $\backslash x: E$ into a lazy service $\backslash x: (P; S)$ that consists of a functional expression P and a side effect expression S. At runtime, we use the extended evaluation function *eval* to evaluate the side effect expression S, which yields a form F_s and some side effects SE'. In addition, we build a lazy form F^* that consists of the functional expression P, the side effect form F_s and the argument form F_a. Finally, when the resulting form is actually needed, we again apply the function *eval* to evaluate the lazy form F^* and get the resulting form F'.

In order to show that our lazy evaluation is behavior preserving, we basically have to prove that the form F' and the side effects SE' that result from the lazy evaluation are *equivalent* to the form F and the side effects SE resulting from the standard Piccola evaluation. Equivalence on the domain of forms and side effects can be formally defined by using the Piccola calculus, which is based on the polyadic π-calculus [9]. Since we do not treat the Piccola calculus in this thesis, we use the following informal definition: Two side effects SE and SE' are considered equivalent, if they contain the same atomic side effects in the same order. (Empty side effects ε do not matter and can be ignored). Two forms F and F' are considered equivalent, if they represent equivalent services[1] and contain the same set of labels L and projections on each of these labels $x \in L$ yield equivalent forms for both F and F'.

In Section 5.4.5, we have defined a semantic function *lazyEval*, which combines the three steps involved in our lazy evaluation. Assuming that the binary operator $\equiv$ stands for equivalence of forms respectively side effects, we can now say that our

[1] Two forms represent equivalent services if either both forms do not contain a service or if the services of both forms yield equivalent results (form and side effects) when invoked with an arbitrary argument form F.

lazy evaluation strategy is behavior preserving if for all the services $\backslash x: E$ and all the forms F it is true that:

$$F \equiv F' \quad and \quad SE \equiv SE' \qquad \begin{array}{r} where\ (F; SE) = eval((\backslash x: E)\ F) \\ and\ (F'; SE') = lazyApply(\backslash x: E; F) \end{array}$$

We believe that we can inductively prove this equivalence on the definition of the form expressions E in the body of the service $\backslash x: E$. Note that if the resulting forms contain services, we apply the induction assumption to show that they are equivalent.

5.6 The SPiccola based prototype implementation

The lazy strategy described in this chapter has been implemented as a prototype based on SPiccola. In this thesis, we do not treat this implementation in detail, and we just mention a few important issues.

Avoiding code duplication and multiple evaluations of expressions

The heart of our lazy evaluation strategy is the partial evaluation algorithm *split* presented in Section 5.4.3. This algorithm resolves literal dependencies by inlining the expressions that are referenced by identifiers. Since a naive inlining approach leads to code duplication and multiple evaluations of the same expressions, we use a more sophisticated technique that is based on pointers (respectively object references in Squeak). This means that the algorithm *split* resolves literal references to functional expressions by using pointers to the original expressions. This avoids unnecessary duplication of code. When a functional expression with multiple references is evaluated for the first time, it gets replaced by its value, which is directly returned for subsequent evaluations. This ensures, that every expression is only evaluated once.

It is important to understand that this technique does not only avoid multiple evaluations of expressions that appear in more than one binding of a service, but it also avoids multiple evaluations of certain expressions in all but the first invocation of a service. In fact, the functional expressions that are independent of the concrete argument (and the dynamic namespace) are evaluated only when the service is executed for the first time and are then replaced by their value. As an example, we consider the following script, which defines a service `f` that is invoked with different arguments.

```
f X:
  print X + fact(10) + 5

f 200
f 300
```

Because the expression `fact(10) + 5` is independent of the function argument (respectively the dynamic namespace), it is only evaluated in the first invocation and then replaced by its result (`3628805`). Thus, for all the subsequent invocations, the service `f` looks as follows:

```
f X:
  print X + 3628805
```

Note that all these optimizations do not affect the semantics of a service because they are only used within functional expressions, which are referentially transparent.

Using pragmas to indicate referential transparent services

Although Piccola exhibits information that allow us to iteratively determine the expressions causing side effects, we cannot find out which of the external services effectively cause side effects. Therefore, our prototype implementation assumes that external services have side effects by default. The programmer can indicate a referentially transparent external service by means of pragmas.

Using pragmas to transform services at definition time

Because there is no compiler for SPiccola yet, executable SPiccola code is represented as parse trees. By default, we apply the partial evaluation *split* at the time when the parse tree of a service is built. On the one hand, this has the advantage, that the transformation has to be executed only once and that it has no negative impact on runtime efficiency at all. On the other hand, applying the transformation at compile time (respectively parse time) has the disadvantage that we have no information about the execution context. This means that we do not know the definitions of the services available in the environment (i.e. the static namespace) and cannot determine whether they cause side effects. As a consequence, the code resulting from the partial evaluation is less efficient than it could be.

This problem can be solved if we apply our partial evaluation algorithm at the time when a service is defined (i.e. when the expression that defines the service is executed). At this time, all the entries in the static namespace are available, and in particular, we can access the structure of the used services and inline their definitions.

In our prototype implementation, we use pragmas to indicate that a service should be transformed at definition time instead of compile time (respectively parse time).

Although this makes execution of the service definition slower, it usually increases the performance of service invocations and may therefore lead to a better overall performance. Obviously, the performance gain gets bigger if the service is invoked more often.

5.7 Application examples

In this section, we present a few application examples for the lazy evaluation technique developed in this chapter. We apply the transformation *split* to the example services and represent the resulting lazy services *\x: (P, S)* as tables.

Example 1 — A referentially transparent service

Table 5.10 shows the transformation of the referentially transparent service `asNumber` introduced in Figure 5.1. This service uses the referentially transparent service `addComparison`, which is represented by a lazy service without a side effect. Therefore, the invocation of `addComparison` in the service `asNumber` is directly replaced by a substitution that replaces the formal argument of `addComparison` with the concrete invocation argument. Since both the formal and the concrete argument are the identifier `X`, this substitution is trivial and we essentially inline the body of the service `addComparison`.

Service	Functional part	Side effect

`addComparison(X)`	`_==_ Y: dynamic.wrapper.asBoolean X._=_(Y)`
	`_!=_ Y: dynamic.wrapper.asBoolean X._~=_(Y)`
	`_<_ Y: dynamic.wrapper.asBoolean X._<_(Y)`
	`_>_ Y: dynamic.wrapper.asBoolean X._>_(Y)`
	`_<=_ Y: dynamic.wrapper.asBoolean X._<=_(Y)`
	`_>=_ Y: dynamic.wrapper.asBoolean X._>=_(Y)`
		ε

`asNumber(X)`	`peer = X.peer`
	`_==_ Y: dynamic.wrapper.asBoolean X._=_(Y)`
	`_!=_ Y: dynamic.wrapper.asBoolean X._~=_(Y)`
	`_<_ Y: dynamic.wrapper.asBoolean X._<_(Y)`
	`_>_ Y: dynamic.wrapper.asBoolean X._>_(Y)`
	`_<=_ Y: dynamic.wrapper.asBoolean X._<=_(Y)`
	`_>=_ Y: dynamic.wrapper.asBoolean X._>=_(Y)`
	`-_: asNumber X.negated()`
	`_+_ Y: asNumber X._+_(Y)`
	`...`
		ε

Table 5.10: Transformation of referentially transparent services

Example 2 — A compound service

As a more involved example, we consider the service `f`, which invokes the service `chFact` introduced in Section 5.3.1.

```
chFact Ch:
   value = Ch.receive()          # A blocking read operation on the channel
   factorial = fact value        # Calculate factorial
   status = factorial < 256

f Ch:
   result = chFact Ch
```

The transformation of these services is shown in Table 5.11. Since reading on a communication channel is not referentially transparent, it is performed in the side effect part. Note that in the functional part of both functions `chFact` and `f` the application of `fact y1` (respectively `fact y1`) appears multiple times, but in a real implementation, multiple evaluations of the same expressions is avoided and pointers (references) are used to decrease the code size (cf. Section 5.6).

Service	Functional part	Side effect

`chFact(Ch)`	`value = y1`
	`factorial = fact y1`
	`status = (fact y1) < 256`
		`y1 = Ch.receive()`

`f(Ch)`	`result =`
	`  value = y2.y1`
	`  factorial = fact y2.y1`
	`  status = fact y2.y1 < 256`
		`y2 =`
		`  y1 = Ch.receive()`

Table 5.11: Transformation of compound services

Example 3 — Default arguments

The last example illustrates how our partial evaluation technique eliminates the performance overhead introduced by specifying default arguments in a Piccola service.

The service `newBox` defines default arguments for both `width` and `height` of a new box that is returned as the result. The client service `g` invokes `newBox` with an argument that specifies a specific value for `height`.

```
newBox X:
   '(with = 10, height = 15, X)
   'println "Width: " + with
   newPeerBox (w = width, h = height)

g: box = newBox(height = 20)
```

Table 5.12 shows how these services are transformed. The functional part of the service `newBox` refers to the result of the peer service that creates the new box. The side effect part of `newBox` contains the invocation of `println` and `newPeerBox`. Note that the quoted expressions are inlined and do not appear in the transformed service. The functional part of the service `g` consists of a binding that assigns the box created within the side effect part to the label `box`. In the side effect part, the invocations of `println` and `newPeerBox` are statically simplified, and therefore they directly contain the literal arguments.

Service	Functional part	Side effect

`newBox(X)`	*y2*	`y1 = println "Width " +`
		`  (width = 10, X).width`
		`y2 = newPeerBox`
		`  w = (width = 10, X).width`
		`  h = (height = 15, X).height`

`g()`	`Box = y3.y2`	`y3 =`
		`  y1 = println "Width: " + 10`
		`  y2 = newPeerBox (w = 10, h = 20)`

Table 5.12: Optimization of default arguments

Chapter 6

Conclusion

The research described in this thesis is focused on two important issues of scripting language design and implementation. The first issue is inter-language bridging, and we developed a technique to use external objects in a flexible and higher-level way that can be dynamically configured within the scripting language. The second issue is performance. We show how we can use lazy evaluation to optimize the additional layer of flexibility introduced by the inter-language bridge. In particular, we present a partial evaluation technique that separates the non-transparent part of a service and therefore allows us to apply lazy evaluation in the presence of side effects. Because this approach is entirely generic, it is not limited to the inter-language bridge and can be used to improve the performance of our language in general. Although the presented solutions are specific to the language Piccola, we believe that similar techniques can also be applied to other programming languages and in particular to other scripting and composition languages.

Inter-language bridging

After an overview of the language Piccola, we explain the problems caused by a lack of abstraction for accessing external objects, and we show that these problems prevent a composition language like Piccola from dealing with components in a higher-level and independent manner. Analysis of these problems leads to the conclusion that it is not possible to achieve the needed flexibility with a generic bridging strategy that is hardcoded in the virtual machine. Thus, our solution introduces a modified strategy for inter-language bridging that reduces the activities in the virtual machine to a technical conversion and performs the higher-level configuration in an abstraction layer located on Piccola's meta-level. This allows the programmer to use the full expressive power of Piccola to adapt and configure the external objects according to the demands of the application and the used compositional (architectural) style.

Using the bridging strategy presented in this thesis, we were able to develop the Piccola 3 standard, which is independent of the underlying host language. Since Piccola is a pure composition language, the standard specifies standard components such as numbers, collections and URLs that are used very frequently. Depending on the specific requirements of an application, the programmer may dynamically reconfigure these components or replace them with more appropriate ones. The

standard components are available in the latest versions of JPiccola and SPiccola, and they allow us to write host-independent Piccola scripts.

Optimization using lazy evaluation

Moving the variable part of the inter-language bridge onto Piccola's meta-level leads to a great flexibility in specifying external components, but at the same time, it is far less efficient than performing the bridging operations inside the virtual machine. Analyzing and profiling typical scripts lead to the conclusion that most of the interface abstractions built by the wrappers of the bridging framework are never used. Therefore, we introduce a lazy evaluation strategy that only executes the effectively needed expressions of a service. In order to use lazy evaluation for all Piccola services, we develop a partial evaluation algorithm that separates the side effect of a service and resolves internal dependencies. It turns out that this can be done very effectively in Piccola. Form expressions exhibit the right kind of information to statically analyze a script, and there is a direct and natural mapping between the syntactical form expressions and the actual forms, which provide lightweight introspection facilities. Furthermore, Piccola has no built-in datatypes that would complicate reasoning. Nevertheless, we believe that the presented partial evaluation technique can also be applied to other languages.

Using this optimization technique, only the effectively used bridging code is executed. If the wrapping services can be statically determined (i.e. they are not defined in the dynamic namespace), the bridging code is directly *weaved* into the script and is executed extremely efficiently. It is interesting to compare this effect with aspect-oriented programming [35].

Due to its generic nature, this technique can also be used to optimize many other aspects of Piccola. As an example, the prototype implementation in SPiccola ensures full laziness, which means that in a function body, the subexpressions that do not depend on the arguments are only evaluated once [19]. Resolving literal references avoids the lookup of identifiers in the static namespace and makes service definitions independent of the static namespace at the time when they were defined. This independence will be very useful to efficiently send a Piccola service over the network in a distributed scenario. (Piccola does not support distribution yet). We also use this partial evaluation technique to detect errors such as undefined identifiers already at compile time.

Chapter 7

Related and future work

The main contributions of this thesis are in the field of inter-language bridging and partial evaluation respectively lazy evaluation in presence of side effects. In this chapter, we present some of the related and future work.

Related work

In this thesis, we present a flexible bridging layer that is located inside the language Piccola and facilitates using external components by means of wrappers. Agora [15] is a prototype-based object-oriented programming language that is entirely based on message passing. However, on the level of the Agora implementation, other concepts such as delegation, encapsulation, cloning and object concatenation can be found as explicit operations on objects. By making Agora reflective, these implementation operators become visible and accessible to the programmer. In Agora, there are built-in objects that are wrapped versions of their corresponding implementation language objects. The wrapping is performed by sending the *up* message to an implementation level object. (This means that the object is passed from the implementation language *up* to the Agora language). On the Agora level, all the wrapped objects understand the messages *send* and *down*, which are used to send a message to the implementation level object and to retrieve the corresponding implementation level object, respectively. Using these three messages (*up*, *down*, and *send*), an Agora programmer can make use implementation level objects and their methods.

Accessing external structures is an important feature of any scripting and composition language, but usually, they do not allow the user to configure the external objects in the scripting language itself. In Python, C/C++ libraries are accessible as extension modules. An extension module needs to register wrappers with Python. These wrappers serve as a glue layer between the languages and are responsible for converting function arguments to C and for returning Python friendly return values [21][22][23]. Other than with Piccola, the programmer has to write the wrappers in C/C++, and although there are extension-building tools such as SWIG [23][24] or GRAD available, this is far less high-level and ad-hoc than the Piccola approach. Ruby [25] has an extension API that is similar to the one used by Python. Also here, extensions have to be initialized (registered) with a C/C++ function that associates methods with object types. Also Tcl [26] and Perl [27] use a similar technique to access external structures.

In Piccola, accessing external components can be done generically since both implementation languages of Piccola (Java and Squeak) provide run-time introspection. We are more interested in the other side of the coin, namely how the scripting language can raise the level of abstraction without adding too much runtime overhead. Jones et al. [28] use Haskell to script COM components and make use of higher-order functions. They also use the type system to detect certain composition errors at compile time.

IBM's System Object Model (SOM) [38] is another approach that allows a programmer to use components that are written in another language. SOM is designed specifically to overcome the main obstacles to the pervasive use of object class libraries. System objects can be distributed in binary form. In addition, they can be used and subclassed across different languages. This means that it is possible to implement a system object using one language, subclass it using another language and use it to build an application in yet a third language. SOM is based on an advanced object model and an object-oriented runtime engine that supports this model. SOM supports the concepts and mechanisms that are normally associated with object-oriented systems including inheritance, encapsulation, and polymorphism. Furthermore, there are some advanced object mechanisms such as metaclasses, different types of method dispatch (static and dynamic), dynamic class creation, and user intercept of method dispatch.

In the fifth Chapter of this thesis, we present a partial evaluation strategy to transform Piccola scripts into equivalent scripts that can be evaluated lazily. Consel and Danvy survey the field of partial evaluation and present a critical assessment of the state of the art [18].

Reasoning about side effects is a necessary precondition to apply our technique to other languages. Sample et al. argue that even more information should be used for composition, like cost, associated network delay, or security requirements [29]. Side effects are a very critical issue for lazy evaluation in general. Our solution is based on locating the side effects as accurately as possible and separating them from the rest of the service. Gifford and others proposed an effect typing discipline to delimit the scope of computational effects within a program [30], while Moggi proposed monads for much the same purpose [31][32]. Wadler shows how to combine these two approaches [33].

Future work

In parallel to the work described in this thesis, we have also been working on distributed Piccola, and our goal is to implement a distribution layer directly in Piccola. In particular for distribution between heterogeneous Piccola hosts, we need a flexible technique to abstract away from the host language. In addition, we can use our partial evaluation algorithm to make Piccola services independent from the static

namespace at the time when they were defined. This facilitates sending services efficiently over the network.

The SPiccola based prototype implementation of our lazy evaluation strategy is still in a very early phase and we have no significant benchmark data yet. Future work will show the performance benefits for different type of Piccola scripts and wrapping frameworks. In particular, we have to find out in which cases it is better to apply the partial evaluation algorithm at definition time instead of applying it at compile time.

The basic concept of our lazy evaluation strategy is to separate the side effects of a service from the referentially transparent part. In our prototype implementation, we use this separation to avoid multiple evaluations of (referentially transparent) expressions that are independent of the service argument (respectively the dynamic namespace). It would be interesting to examine other optimizations that can be applied to the referentially transparent expressions of a service. As an example, we could cache the values of the referentially transparent part of a service for often-used arguments.

We are also working on an integrated composition environment in Piccola. The information provided by partial evaluation can be used to determine identifier values and possible runtime errors already when the user writes the source code. In addition, the partial evaluation algorithm can be used to simplify source expressions statically. One can think of tool-tips like information when the user selects an identifier or an expression in the source code. Type information would help to improve the static analysis and would make the partial evaluation even more effective.

Bibliography

[1] Franz Achermann and Oscar Nierstrasz. „*Applications = Components + Scripts – A Tour of Piccola*". Software Architectures and Component Technology, Mehmet Aksit (Ed.), Kluwer, 2001.

[2] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz. „*Piccola – a Small Composition Language*". Formal Methods for Distributed Processing, an Object Oriented Approach, Howard Bowman and John Derrick. (Eds.), Cambridge University Press, 2001.

[3] Jean-Guy Schneider and Oscar Nierstrasz. „*Components, Scripts and Glue*". Software Architectures — Advances and Applications, Leonor Barroca, Jon Hall and Patrick Hall (Eds.), pp. 13-25, Springer, 1999.

[4] Markus Lumpe, Franz Achermann and Oscar Nierstrasz. „*A Formal Language for Composition*". Foundations of Component Based Systems, Gary Leavens and Murali Sitaraman (Eds.), pp. 69—90, Cambridge University Press, 2000.

[5] Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz. „*Scripting Coordination Styles*". Coordination Languages and Models, António Porto and Gruia-Catalin Roman (Eds.), LNCS 1906, Limassol, Cyprus, September 2000, pp. 19—35.

[6] Oscar Nierstrasz and Franz Achermann. „*Separation of Concerns through Unification of Concepts*". ECOOP 2000 Workshop on Aspects & Dimensions of Concerns, 2000.

[7] Franz Achermann and Oscar Nierstrasz. „*Explicit Namespaces*". Modular Programming Languages, Jürg Gutknecht and Wolfgang Weck (Eds.), LNCS 1897, Zurich, Switzerland, September 2000, pp. 77—89.

[8] Jean-Guy Schneider. „*Components, Scripts, and Glue: A conceptual framework for software composition*". Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[9] Franz Achermann. „*Piccola: A Language to Script Composition Styles*". Ph.D. thesis, University of Bern, Institut of Computer Science and Applied Mathematics, 2001, to appear.

[10] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Key. „*Back to the future: The story of Squeak, A practical Smalltalk written in itself*". In Proceedings OOPSLA '97, ACM SIGPLAN Notices, volume 21, November 1997.

[11] John K. Ousterhout. *„Scripting: Higher Level Programming for the 21st Century"*. IEEE Computer magazine, March 1998.

[12] David Ungar and Randall B. Smith. *„Self: The Power of Simplicity"*. OOPSLA '87 Conference Proceedings, pp. 227-241, Orlando, FL, October 1987.

[13] Adele Goldberg, David Robson. *„Smalltalk 80: The Language"*. Addison-Wesley Pub Co, ISBN: 0201136880, June 1989.

[14] Adele Goldberg, David Robson. *„Blue Book - Smalltalk-80: The Language and Its Implementation"*. Out of print.

[15] Wolfgang De Meuter. *„Agora: The Story of the simplest MOP in the World - or - The Scheme of Object-Orientation"*. Programming Techology Lab, Department of Computer Science, Vrije Universiteit Brussel.

[16] Gregor Kiczales, Jim des Rivière and Daniel G. Bobrov. *„The Art of the Metaobject Protocol"*. MIT Press, 1991.

[17] John Launchbury. *„Lazy imperative programming"*. In Proceedings of the ACM SIGPLAN Workshop on State in Functional Languages, Copenhagen, Denmark, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[18] C. Consel, O. Danvy. *„Tutorial Notes on Partial Evaluation"*. In 20th ACM Symposium on Principles of Programming Languages. Charleston, South Carolina, pp.493-501, ACM Press 1993.

[19] Philip Wadler. *„Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time"*. Proceedings of the 1984 ACM Conference on LISP and Functional Programming, pp. 45-52, August 5-8, 1984, Austin, Texas, USA. ACM

[20] Mark Lutz. *„Programming Python (2nd Edition)"*. O'Reilly & Associates, ISBN: 0596000855, March 2001.

[21] Guido van Rossum, Fred L. Drake, Jr. *„Extending and Embedding the Python Interpreter"*. PythonLabs, Release 2.1, April 15, 2001

[22] Guido van Rossum, Fred L. Drake, Jr. *„Python/C API Reference Manual"*. PythonLabs, Release 2.1, April 15, 2001

[23] David M. Beazley. *„Interfacing C/C++ and Python with SWIG"*. 7th International Python Conference, SWIG Tutorial, 1998

[24] David M. Beazley. *„SWIG: an easy to use tool for integrating scripting languages with C and C++"*. In proceedings of the 4th USENIX Tcl/Tk Workshop pp. 129-139, July 1996.

[25] Yukio Matsumoto, Yukihiro Matsumoto. *„The Ruby Programming Language"*. Addison Wesley Professional, ISBN: 020171096X, February 2002, to appear.

[26] John K. Ousterhout. *„Tcl and the Tk Toolkit“*. Addison Wesley Professional, ISBN: 020163337X, May 1994.

[27] Larry Wall, Tom Christiansen, Jon Orwant. *„Programming Perl (3rd Edition)“*. O'Reilly & Associates, ISBN: 0596000278, July 2000.

[28] Simon L. Peyton-Jones, Erik Meijer and Daan Leijen. *„Scripting COM components in Haskell“*. Fifth International Conference on Software Reuse (ICSR5), Victoria, Canada, 1998.

[30] D. K. Gifford and J. M. Lucassen. *„Integrating functional and imperative programming“*. ACM Conference on Lisp and Functional Programming, Cambridge, Massachusetts, August 1986.

[31] E. Moggi. *„Computational lambda calculus and monads“*. IEEE Symposium on Logic in Computer Science, Asilomar, California, June 1989.

[32] E. Moggi. *„Notions of computation and monads, Information and Computation“*. 93(1), 1991.

[33] Philip Wadler. *„The marriage of effects and monads“*. Proceedings of the third ACM SIGPLAN international conference on Functional programming, September 26 - 29, 1998, Baltimore, MD USA.

[34] McDowell, C., and Helmbold, D. *„Debugging Concurrent Programs“*. ACM Computing Surveys 21, 4 (Dec. 1989), 593 - 622.

[35] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *„Aspect-oriented programming“*. In Proceedings of the European Conference on ObjectOriented Programming (ECOOP), number 1241 in Lecture Notes in Computer Science. Springer-Verlag, June 1997.

[36] Herb Schildt. *„Java 2: The Complete Reference (Osborne Complete Reference Series)“*. McGraw-Hill Professional Publishing, ISBN: 0072130849, December 2000.

[37] Bertrand Meyer. *„Eiffel: The Language“*. Prentice Hall PTR, ISBN 0132479257, March 2001.

[38] Christina Lau. *„Object-Oriented Programming Using SOM and DSOM“*. John Wiley & Sons, ISBN 0471131237, March 1995.

Appendix A

SPiccola and its debugger

Within the scope of this thesis, we implemented SPiccola, which is a Piccola implementation on top of Squeak. Squeak is an open, highly portable Smalltalk-80 implementation whose virtual machine is written entirely in Smalltalk [10]. Having a second Piccola implementation has been essential to gain practical experiences about inter-language bridging and host-independence. It was also interesting to compare the strongly typed Java language to the type-less Squeak language with respect to using them as a Piccola host. As expected, it turned out that the type-less approach makes implementation of the inter-language bridge easier and more natural.

SPiccola consists of a parser, a virtual machine and an integrated Piccola development environment. In addition, there are several tools for thread-aware debugging that are well integrated into the SPiccola development environment. As explained in Section 2.1, concurrency is a primitive concept of the Piccola language and therefore, concurrency is an important issue for debugging tools in Piccola. Unfortunately, the classical debugging techniques used for sequential programs are only of limited use for debugging concurrent programs. The fact that several threads may be active at a time adds a new dimension of complexity and a program cannot be considered a totally ordered sequence of operations anymore. In fact, concurrent programs do not always have deterministic and reproducible behavior and even when they are run with the same inputs, their results may be radically different. This is caused by races, which occur whenever two or more concurrently executing threads make use of the same memory location, where at least one of the threads modifies the contents at this location, and the accesses are not ordered by synchronization.

This non-determinism has major negative impacts on debugging and testing concurrent programs. In particular, debugging of a concurrent program often fails because the undesirable behavior may not appear when the program is re-executed. Especially if this behavior occurs with a low probability, the programmer may never be able to recreate the error situation. In fact, any attempt to gain more information about the program may prevent the programmer from reproducing the erroneous behavior. This effect has been referred to as the *Heisenberg uncertainty principle applied to software* or the *Probe Effect* [34].

In SPiccola, we tackle these problems by two main concepts: First, we make the complete debugger interface accessible within the Piccola language. This allows the

programmer to specify the debugging operations without additional user interaction at runtime, and it significantly decreases the problems caused by the Probe Effect. Second, we provide SPiccola with event-based debugging techniques [34]. This means that the debugger supports logging of specific events at runtime, which allows a reconstruction of the actions performed by the script after some or all of its threads are already terminated.

In the following, we give a short overview of SPiccola's main debugging features and especially focus on the aspects related to concurrency:

Association between parse tree nodes and source code. As most of the other debuggers, the SPiccola debugger associates the executable code (i.e. the parse tree nodes) with the source code. Whenever the programmer looks at a certain position of the executable code, these associations are used to show the programmer the corresponding location in the source code.

Thread-aware cyclical debugging. Cyclical debugging is the classical debugging approach where a program is repeatedly stopped during execution in order to examine its state [34]. SPiccola provides all the commonly used features to specify breakpoints and step through a script. In addition, the cyclical debugger in SPiccola is completely thread-aware. It is possible to simultaneously attach debugger windows to multiple threads and to control them individually. When a thread with an attached debugger forks a new agent, another debugger window is opened and attached to the thread of the newly created agent.

Debugger interface is accessible within Piccola. The complete debugger interface is accessible within the Piccola language. As an example, this allows us to define breakpoints by using the according debugging commands. In addition, also the meta-information about the threads is available in Piccola, and there is a notion of *groups of threads*. Combined with the debugger interface, this can be used to atomically apply debugging operations to a group of threads. As an example, a programmer may open a debugger for each thread in a specific group.

Thread animation. SPiccola can animate the threads that are executing a Piccola script. While the programmer is watching the threads walking through the source code, he can also influence them. As an example, he can change the active thread (i.e. the thread owning the CPU) or influence the execution speed of an individual thread.

Runtime history and form lifecycles. SPiccola is able to log specific actions during the execution of a script. After the execution is terminated, the logged information is available in the *runtime history*. Using this history, the programmer can inspect the runtime behavior without forcing the Probe Effect. In addition, SPiccola also supports *form lifecycles*. This means that the debugger logs every usage of a certain form and allows the programmer to browse them afterwards.

As an example, the form lifecycle of a channel may be used to easily examine communication or synchronization of threads.