Partial evaluation of inter-language wrappers

Nathanael Schärli and Franz Achermann

{`schaerli`|`acherman`}`@iam.unibe.ch`
Software Composition Group, Schützenmattstr 14, CH-3012 Bern
University of Berne

Abstract. Wrapping external components by scripts can be a performance bottleneck if inter-language bridging is frequent. Piccola is a pure composition language that wraps components according to a specific composition style. This wrapping must be efficient, since even arithmetic operations are done by external components. In this paper we present how to use partial evaluation to overcome much of the overhead associated with the wrapping. It turns out that Piccola scripts can be highly optimized since form expression exhibit the right kind of information to separate side effects from services and resolve internal dependencies.

1 Introduction

Piccola [1] is a pure composition language that has a generic mechanism to use external components. It allows the user to adapt these components according to a specific composition style using wrappers. We argue that a composition language must allow the user to specify this glue code in a flexible and highlevel way. On the other hand, efficient execution of the glue code is an important issue for a composition language or inter-language bridging in general.

We propose the use of partial evaluation to optimize the wrapping abstractions. Piccola is well-suited for partial evaluation due to its simple syntax and semantics based on forms. Forms are extensible records and have light-weight introspection facilities. Piccola has no built-in datatypes or objects that would complicate reasoning.

In this paper we present a partial evaluation technique that allows lazy evaluation. We represent the result of wrapping — and of service invocations in general — by lazy forms. Lazy forms have the advantage that only necessary expressions are evaluated. The technical contribution of this paper is the separation of side-effects from services to make them referentially transparent.

The paper does not give an introduction to Piccola itself. An overview of Piccola and its use of forms can be found in [1] and in the forthcoming thesis of the second author. We explain the necessary syntax and semantics of Piccola when discussing the code examples.

In Section 2, we present a wrapper for Piccola components and examine the introduced performance loss. In Section 3, we present the concept of lazy forms and illustrate the partial evaluation strategy. Section 4 contains a set of examples, Section 5 concludes the paper and addresses related and future work. The partial evaluator for core Piccola is in the appendix.

2 Profiling a SPiccola wrapper

```
addComparison X:
    _==_ Y: asBoolean(X._=_ Y)
    _!=_ Y: asBoolean(X._~=_ Y)      # Squeak uses ~= for inequality
    _<_ Y: asBoolean(X._<_ Y)
    _>_ Y: asBoolean(X._>_ Y)
    _<=_ Y: asBoolean(X._<=_ Y)
    _>=_ Y: asBoolean(X._>=_ Y)

def asNumber X:
    peer = X.peer
    addComparison X                  # add all comparison operator bindings
    -_: asNumber X.negated()         # arithmetic operators...
    _+_ Y: asNumber(X._+_ Y)         # plus
    _-_ Y: asNumber(X._-_ Y)
    _*_ Y: asNumber(X._*_ Y)
    _/_ Y: asNumber(X._/_ Y)
    abs: if (_<_ 0)                  # if this is smaller than 0
        then: -_()                   # return -X
        else: asNumber X             # else return self
    trunc: asNumber X.truncated()
```

Fig. 1. The `asNumber` wrapper of SPiccola

SPiccola is the implementation of Piccola on top of Squeak. In Piccola, infix operators can be defined as services in one component. `A + B` invokes the service `_+_` bound in `A` with the argument of `B`. The wrapper `asNumber` in figure 1 receives a form `X` containing a peer that refers to the native Squeak number object. It returns a form of service bindings according to the composition style for numbers. It does so by building a form that contains the necessary operators and services.

Some service like arithmetic plus do have a corresponding Squeak method. The implementation of the plus operator (`_+_`) takes the right-hand side operand `Y`, calls the native Squeak plus (available through `X._+_`) and wraps the result using the number wrapper. Other services like `abs()` are specified using previously defined bindings. Last but not least, the comparison operators are factored out. The invocation `addComparison X` adds the operators `==, !=, ...` to the resulting number form.

Let us consider when this wrapper gets triggered. The evaluation of an expression `a = 1 + 2` causes the following invocations of the wrapper:

1. The external object representing the number 1 gets wrapped by `asNumber`.
2. A projection on the label `_+_` of the wrapped form is performed.
3. The external object representing the number 2 gets wrapped.

4. The wrapped form is passed as an argument to the `_+_` service. Executing this service sends the Squeak + message to the object 1 with a projection on the label `peer`[1] object of 2. The result (the number 3) is again wrapped.

The wrapping service is invoked three times and each time constructs a form with 14 bindings. But for the forms built in step 1 and 3, only one of these bindings (`_+_` or `peer`) is used while all the other bindings are discarded.

Examination of Piccola scripts show that in the average, we only use about 10% of the bindings created by the Piccola wrappers. Assuming 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 could be reduced by 90% if we only created the bindings that are actually used.

In a reference implementation of Piccola the introduction of this number wrapper leads to approximately six times slower performance. It should be noted that our current requirements for number specifies only a small number of bindings per number. In contrast, the Squeak v2.9 `SmallInteger` object understands more than 400 messages. If the wrapper consists of so many bindings, the performance penalty would be much worse.

3 Lazy Forms

Lazy evaluation is used in certain functional languages like Haskell. It means that beta reduction gets only performed when a value is actually used. In Piccola, evaluation is in general strict, since the language has side effects. In order to avoid the evaluation of expressions in unused bindings, we can split up service invocation into the creation of a lazy form and its usage. A lazy form defers evaluation of its bindings and only evaluates the bindings that are effectively needed. As forms are immutable, it is possible to cache the projected value.

At *invocation time*, we only execute the side effect specified by the service and return a lazy form. When a binding of the lazy form is *used*, only the bound expression gets evaluated. The bound expression may refer to the argument of the lazy form.

There are two critical requirements for services to invoke them lazily: The services need to be separated into a functional and side-effect part and the functional part may not contain internal dependencies.

Separated side-effect. It is necessary to execute the side effect part on invocation time. Consider the following service:

```
chfact channel:
    value = channel.receive()     # wait for a channel
    factorial = fact value        # heavy calculation without sideeffect
```

---

[1] The projection on `peer` does not appear in the SPiccola code since it is part of the generic inter-language bridge.

It contains a side effect receiving a value from a channel and an invocation of a referentially transparent function `fact` to calculate the factorial. The side effect part can be represented by the following anonymous service:

```
\channel: s1 = channel.receive()
```

and the functional part by:

```
\channel Sideeffect:
    value = Sideeffect.s1
    factorial = fact value
```

The functional part is a higher order abstraction taking the result of the side effect as an additional argument. Note that a service can contain different side effects which are bound by different fresh labels, like `s1`. The original service `chfact` can is the functional composition of the above two anonymous services.

Independent bindings. In order to evaluate only a single binding when a projection occurs, it is necessary that it does not refer to other bindings in the same lazy form. In our example, the `factorial` binding refers to `value`. This dependency can be resolved by replacing the use of `value` by its definition. Note that substitution of referentially transparent bindings does not introduce side effects.

```
\channel Sideeffect:
    value = Sideeffect.s1
    factorial = fact Sideeffect.s1        # No reference to value
```

The reader may want to notice the following implementation aspect. Although we present de-serialization by substitution, it is common to use references in order to avoid multiple expensive computations. This technique is used to implement call-by-need evaluation.

In the next section, we give some more examples of the separation. The formal translation of core Piccola is given in appendix A. Due to lack of space, we only present the core of Piccola omitting syntactical sugar like user defined operators or dynamic namespaces.

4   Transformation examples

Lazy closures are a triple $\mathbf{Cl}(x, P, S)$ where x is the formal argument, P is the purely functional part and S is the side effect. Below we present lazy services in tables.

A referentially transparent service. As a first example, consider the invocation of `addComparison` service presented in section 2. The service `addComparison` is transformed into a lazy closure:

service	functional part	side effect
addComparison(X)	`_==_ Y: asBoolean(X._=_ Y)`	ϵ
	`_!=_ Y: asBoolean(X._~=_ Y)`	
	`_<_ Y: asBoolean(X._<_ Y)`	
	...	

Since it does not contain side effects, we can inline its body into the code. Thus `asNumber` becomes:

```
def asNumber X:
    peer = X.peer
    _==_ Y: asBoolean(X._=_ Y)       # in-lined addComparison
    _!=_ Y: asBoolean(X._~=_ Y)
    _<_ Y: asBoolean(X._<_ Y)
    ...                              # rest as in Figure 1
```

Optimize compound service. As a more involved example consider the service `f` which contains an invocation of `chfact` defined in the previous example. Written in tabular form the service `chfact` is

service	functional part	side effect
chfact(ch)	`value = ` *eff* `.s1`	`s1 = ch.receive()`
	`factorial = fact ` *eff* `.s1`	

where *eff* denotes the result of evaluation the side effects. We only write the body of the anonymous abstractions and use the identifier *eff* as placeholder for the argument. Now assume a service `f` invoking `chfact`:

```
f ch:
    result = chfact ch
```

The service `f` is split up into functional and side-effect part as:

service	functional part	side effect
f(ch)	`result = `	`s1 = ch.receive()`
	`    value = ` *eff* `.s1`	
	`    factorial = fact ` *eff* `.s1`	

Default Arguments. The last example shows the use of wrappers to define default arguments. The glue code adding default values is optimized away by partial evaluation. Assume a service `newBox` creating a new peer box with default arguments and a client service `c` as follows:

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

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

In Piccola, a quoted expression `E is syntacic sugar for $\mathbf{root} = (\mathbf{root}, E)$. Thus a quoted expression modifies only the local namespace denoted by the keyword $\mathbf{root}$ and does appear in the resulting form. Consequently, double quoted expressions extend the namespace with the empty form and denote pure side effects.

These services `newBox` and `c` get split into lazy closures as:

service	functional part	side effect

`newBox(X)`	eff `.s2`	`s1 = println "Width: " +`
		`    (width = 10, X).width`
		`s2 = newPeerBox`
		`    w = (width = 10, X).width`
		`    h = (height = 15, X).height`

`c ()`	`box = ` eff `.s2`	`s1 = println "Width: " + 10` `s2 = newPeerBox(w = 10, h = 20)`

Observe that the client takes the default width 10 from the wrapper and specifies its own height 20 for the peer box. The `newBox` wrapper does not add any performance overhead.

5 Related and future work

This work mainly uses partial evaluation to transform Piccola scripts, (e.g. [3]) In addition to partial evaluation we also translate services to isolate the side effect part. 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 [6].

In order to script external components, they have to imported into the composition environment first. The tool SWIG [2] automatically generates wrappers in C to script components written in C and C++ in Perl, Tcl, or Python. Making components available in Piccola can be done generically since both implementation languages of Piccola (Squeak and Java) provide run-time introspection. We are more concerned with the other side of the coin, namely how the scripting language can raise the level of abstraction without adding to much runtime overhead. Jones et al. [4] use Haskell to script COM components making use of higher-order functions. They use the type system to detect certain composition errors at compile time. However, a lazy language makes it hard to reason about concurrency, although that is not in the scope of this paper.

It is interesting to compare partial evaluation with aspect oriented programming [5]. We *weave* the composition wrappers into existing scripts, thereby making use of the formal foundation of Piccola for reasoning.

We are working on an integrated composition environment in Piccola. The information derived by the translation knows *current value* of an identifiers. This might be helpful in order to detect unwanted corruption of the root namespace. For instance:

```
'size = newVar(17)      # a Piccola variable
'myservice()            # call some other services, extending root
a = size.get()          # return the value stored in size
```

One can think of tool-tips like information when the user selects the identifier size in the last line. If the service call returns a binding size we would unexpectedly return the contents of the wrong binding. Type information would also help to improve the static analyze and vice-verca. Due to lack of space, we omited in the present version the treatment of errors. Certain type errors like using a form without a service as functor can be detected by the analyzer.

The appropriate treatment of fixed-points is still a point of debate. **Def** is syntactic sugar for fixed points. The fixed point is encoded by a channel and as such using the fixed point is not a referentially transparent serivce. However, for mutual recursive services a lazy fixed point combinator would do the job and it would allow us to use the fixed point without sideeffect.

In this extended abstract we introduced a way to split up Piccola services into a side effect and a referentially transparent part. This split enables the execution of wrappers in constant time and not in linear time with respect to the number of overridden bindings.

Acknowledgement

References

1. F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2001. to appear.
2. D. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.
3. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings POPL'93*, pages 493–501. ACM, January 1993.
4. S. Jones, E. Meijer, and D. Leijen. Scripting COM components in haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.
5. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, June 1997.
6. N. Sample, D. Beringer, L. Melloul, and G. Wiederhold. CLAM: Composition language for autonomous megamodules. In P. Ciancarini and A. Wolf, editors, *Proceedings of Coordination'99, LNCS 1594*, pages 291–306, 1999.
7. Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, 2001. to appear.

A The translation

This appendix contains the formal translation of Piccola services into a functional and side-effect part. The translation is explained in more detail in [7]. Core forms expressions in Piccola E evaluate to forms F. Identifiers are ranged over by x. *Lazy forms* F^* are a modification of form expressions:

$$E ::= \epsilon \mid x \mid \mathbf{root} \mid E \cdot E \mid E.x \mid E\,E \mid \backslash x : E \mid x = E, E \mid \mathbf{root} = E, E$$

$$F ::= \epsilon \mid x = F \mid F \cdot F \mid \mathbf{Cl}(x, F, E)$$

$$F^* ::= \epsilon \mid x \mid x = F^* \mid F^* \cdot F^* \mid \mathbf{Cl}(x, F^*, F^*) \mid$$
$$\quad \mathbf{Pr}(F^*, x) \mid \mathbf{Se}(F^*) \mid \mathbf{Ap}(F^*, F^*) \mid \mathbf{Re}(F^*, F^*)$$

We use P, Q, R, S to range over lazy forms. For convenience, S denotes a lazy forms containing a side effect and P a form that specify a pure function. Lazy forms are: The empty form; an identifier; a singleton binding; an extension; a closure $\mathbf{Cl}(x, P, S)$; an unevaluated projection; an unevaluated service selection; an application $\mathbf{Ap}(R_1, R_2)$; or a substitution $\mathbf{Re}(R_1, R_2)$ where R_2 substitutes identifiers in R_1. We can express substitution by a form, since essentially, a form represents a name-value binding.

The translation $[\![E]\!]_R$ defines a tuple (P, S) of lazy forms where P is the functional part and S is the side effect part of evaluating E in the environment R. In the end, we are only interested in P and in-line S wherever possible. $[\![E]\!]_R$ is defined inductively on the grammar for expressions as follows:

$$[\![\epsilon]\!]_R \stackrel{\text{def}}{=} (\epsilon, \epsilon) \tag{empty}$$

$$[\![x]\!]_R \stackrel{\text{def}}{=} (project(R, x), \epsilon) \tag{ident}$$

$$[\![\mathbf{root}]\!]_R \stackrel{\text{def}}{=} (R, \epsilon) \tag{root}$$

$$[\![E_1 \cdot E_2]\!]_R \stackrel{\text{def}}{=} (P_1 \cdot P_2, S_1 \cdot S_2) \qquad (P_1, S_1) = [\![E_1]\!]_R \tag{extend}$$
$$(P_2, S_2) = [\![E_2]\!]_R$$

$$[\![E.x]\!]_R \stackrel{\text{def}}{=} (project(P, x), S) \qquad (P, S) = [\![E]\!]_R \tag{project}$$

$$[\![\backslash x : E]\!]_R \stackrel{\text{def}}{=} (\mathbf{Cl}(x, P, S), \epsilon) \qquad (P, S) = [\![E]\!]_{R \cdot (x = x)} \tag{abs}$$

$$[\![x = E_1, E_2]\!]_R \stackrel{\text{def}}{=} ((x = P_1) \cdot P_2, S_1 \cdot S_2) \qquad (P_1, S_1) = [\![E_1]\!]_R \tag{assign}$$
$$(P_2, S_2) = [\![E_2]\!]_{R \cdot (x = P_1)}$$

$$[\![\mathbf{root} = E_1, E_2]\!]_R \stackrel{\text{def}}{=} (P_2, S_1 \cdot S_2) \qquad (P_1, S_1) = [\![E_1]\!]_R \tag{sandbox}$$
$$(P_2, S_2) = [\![E_2]\!]_{P_1}$$

Due to lack of space, we only explain a few tranformations. The empty form returns the empty form and has no side effect. An identifier returns the projection within the current root context and has no side effect. The keyword **root** returns the current root form and has no side effect. Polymorphic extension returns the extension of its subexpression, and sequentially composes their side

effects. Abstraction returns a closure and has no side effect. The difficut case is invocation:

$$\llbracket E_1 \ E_2 \rrbracket_R \overset{\text{def}}{=} \begin{cases} (\mathbf{Re}(Q, x{=}P_2), & \text{if } P_1' = \mathbf{Cl}(x, Q, \epsilon) \\ \quad S_1 \cdot S_2) & \\ (\mathbf{Re}(Q, (x{=}P_2) \cdot (\mathit{eff}{=}y)), & \text{if } P_1' = \mathbf{Cl}(x, Q, Q') \\ \quad S_1 \cdot S_2 \cdot (y{=}\mathbf{Re}(Q', x{=}P_2))) & \quad \text{and } Q' \neq \epsilon \\ (\mathbf{Pr}(\mathit{eff}, y), & \text{otherwise} \\ \quad S_1 \cdot S_2 \cdot (y{=}\mathbf{Ap}(P_1', P_2))) & \end{cases} \quad \text{(invoke)}$$

where $(P_1, S_1) = \llbracket E_1 \rrbracket_R$, $(P_2, S_2) = \llbracket E_2 \rrbracket_R$, $P_1' = service(P_1)$ and y denotes a fresh identifier. The first case applies when the invoked service does not contain any side effects. In this case, we return a substitution of the formal argument x by the functional part of the argument P_2 in the body Q of the closure. The side effect of the result consist in the sequential composition of the side-effects of the functor and the argument. The second case applies when the service has a side-effect Q'. In this case, the substitution of the first case is extended with a fresh identifier y to contain the *result* of the side effect invocation. Finally, if we do not have enough static information, we return an lazy application. We have evidence that this translation is behaviour preserving, although we have not carried out a formal proof yet.

The meta-function $project(F^*, x)$ returns a projection onto x in the lazy form F^*, and $service(F^*)$ returns the service bound in the lazy forms. In the worst case, for instance when F^* is an identifier these functions denote the lazy projection $\mathbf{Pr}(F^*, x)$, or the lazy service lookup $\mathbf{Se}(F^*)$ respectively.