PINOCCHIO: Bringing Reflection to Life with First-Class Interpreters

Toon Verwaest Camillo Bruni David Gurtner Adrian Lienhard Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland

http://scg.unibe.ch

Abstract

To support development tools like debuggers, runtime systems need to provide a meta-programming interface to alter their semantics and access internal data. Reflective capabilities are typically fixed by the Virtual Machine (VM). Unanticipated reflective features must either be simulated by complex program transformations, or they require the development of a specially tailored VM. We propose a novel approach to behavioral reflection that eliminates the barrier between applications and the VM by manipulating an explicit tower of first-class interpreters. PINOCCHIO is a proof-of-concept implementation of our approach which enables radical changes to the interpretation of programs by explicitly instantiating subclasses of the base interpreter. We illustrate the design of PINOCCHIO through non-trivial examples that extend runtime semantics to support debugging, parallel debugging, and back-in-time object-flow debugging. Although performance is not yet addressed, we also discuss numerous opportunities for optimization, which we believe will lead to a practical approach to behavioral reflection.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Interpreters, Runtime environments; D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

General Terms Reflection, Virtual Machines

Keywords Smalltalk, Behavioral Reflection, Metacircularity, Virtual Machines, Debugging, Object-Flow Analysis

1. Introduction

Debuggers, profilers, sandboxing, support for memory barriers, transactions, the addition of reflective capabilities, and many other language extensions, are all common examples of useful tools and techniques that are conveniently provided by modifying the runtime of a language.

> *[Modifying the interpreter] presents the advantage of having direct access to the internal structure of the interpreter and therefore provides greater flexibility and expressiveness for supporting dynamic adaptation. The major disadvantages are the loss of compatibility with standard environments, which often results in particular tools becoming obsolete quickly, and the complexity of the implementation. Indeed, modifying a production virtual machine is not an easy task, and it is subsequently difficult to keep up-to-date with new versions and technologies ... [39]*

Consider the example of the object-flow debugger [27], a back-in-time debugger that keeps track of the flow of objects in an object-oriented runtime. Although this is a very effective debugging tool for object-oriented languages, the original Object Flow VM prototype quickly became out of sync with the rapidly changing Pharo VM that it was based on [3]. The problem is not related to the choice of modifying the interpreter definition, but rather to the fact this interpreter definition *cannot be changed from within the language itself*. If the semantics of the interpreter could be changed from within, the code would be kept in sync with the standard VM, just like normal applications.

Although high-level languages traditionally support reflective programming, they provide only a limited set of capabilities for introspection and intercession. Normal applications cannot extend the reflective interface. Hence, applications that need features that are not supported either cannot be implemented, must be simulated in a roundabout way, or have to resort to modifying the VM.

Early research on reflection, rooted in Lisp, proposed a different approach to reflection in which the interpreter itself is modeled as a first-class entity [36]. Although such an approach is extremely powerful, it has not been made practical and is not implemented in modern VMs.

In this paper we present PINOCCHIO, a runtime that supports first-class interpreters. PINOCCHIO is based on Small-

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

talk, but uses an AST instead of bytecode interpretation. Applications freely flow from interpreter to interpreter depending on the required semantics. Applications specify their own interpreters *inside* the runtime as subclasses of the default `Interpreter` class, a reification of the core interpreter.

PINOCCHIO absorbs and exploits three features from Smalltalk to effectively extend interpreters:

- *the object model:* PINOCCHIO adopts the object model of Smalltalk— since specialized interpreters typically make only modest changes to their base-level semantics, objects can often flow freely between levels;
- *recursion:* interpreters are defined as recursive AST visitors instead of as bytecode machines with explicit stacks — first-class continuations can be used to implement non-local flow of control;
- *garbage collection:* interpreters can rely on the garbage collection provided by their meta-level.

The contributions of this paper include (i) an analysis of the limits of behavioral reflection as practiced today; (ii) a novel approach to behavioral reflection inspired by Refci's first-class interpreters [36]; (iii) the design and implementation of PINOCCHIO, a proof-of-concept prototype of the approach; (iv) the presentation of three non-trivial case studies demonstrating how PINOCCHIO's design facilitates behavioral reflection.

This paper is organized as follows: Section 2 reviews standard approaches to behavioral reflection and summarizes the practical challenges facing these approaches. In Section 3 we present PINOCCHIO in a nutshell, it explains how PINOCCHIO bootstraps itself with the help of first-class interpreters and presents PINOCCHIO's metaobject protocol. Section 4 illustrates how a first-class interpreter can easily be extended to support debugging, and then motivates the design of multiple interpreters with the help of an example of a specialized interpreter to support object-flow analysis for back-in-time debugging and a parallel debugger. Section 5 compares PINOCCHIO to other dynamic languages in terms of performance, and outlines the further steps required to turn PINOCCHIO into a fast approach to behavioral reflection. Section 6 compares PINOCCHIO with related work. Section 7 concludes with a brief summary of the results.

2. Practical Limits of Behavioral Reflection

We first review basic terminology and history before considering the challenges and limits of behavioral reflection.

Computational reflection refers to the ability of computer programs to reason about their own structure and behavior at runtime [29, 37]. Reflective systems distinguish the *base* (application) level from the *meta* (semantic) level. Reflection entails the *reification* of meta-level entities to the base level, that is, semantic entities are reified as ordinary application entities. (If we ask a Java object for its class we obtain an ordinary Java object representing that class.)

Structural reflection is concerned with reification of the structure of the program, *i.e.*, its data and code. *Behavioral reflection* is concerned with reification of the behavior of the program, *i.e.*, its *interpretation*. Reflection can be further refined into *introspection* and *intercession*. Introspection is purely concerned with reifying meta-level concepts to reason about them at the base level. For example, we may ask an object what fields it has so we can print them all. Intercession, on the other hand, allows us to manipulate reified meta-level entities and *reflect* changes back to the meta-level. In Smalltalk one can change the class of an object at runtime, immediately causing that object's behavior to change. Such changes effect a *causal connection* between the reified entities and the meta-level entities they represent [29].

Some typical uses of reflection are found in (i) debugging tools; (ii) GUIs for object structures; (iii) code instrumentation and analysis tools; (iv) dynamic code generation; and (v) language extensions.

In mainstream high-level languages support for reflection is dictated by whatever is provided by the VM. Short of building a tailored VM, users of the reflective API are strictly limited to the capabilities that have been built into the VM. As a consequence, practical, mainstream uses of reflection are mostly limited to (structural) introspection, for example, as offered by the Java reflection API. These systems provide no means for extending them with reflective capabilities that were not anticipated. Since extending and shipping a modified runtime usually is not an option, innovative extensions are not possible.

Several different approaches to behavioral reflection have been realized over the years, but each suffers from its own practical limitations. Smith introduced the notion of computational reflection and he illustrated his model through the implementation of a reflective dialect of Lisp, called 3-Lisp [37]. 3-Lisp applications can contain special *reifier* functions that take reifications of aspects of the interpreter as arguments: the current expression, the environment in which the expression is being executed and the continuation of the application. In Smith's model these reifiers conceptually run in the scope of the interpreter since they operate on the application from the point of view of the interpreter. Adding support for reifiers to a language thus adds the ability to *add lines* to the code of the interpreter from within the application context. It also makes us believe that below every interpreter there is another interpreter that evaluates the interpreter and all reifications requested by the application on top. Reflection is therefore supported by an infinite *tower of interpreters*.

Starting with Smith's Tower of Interpreters we will review the practical challenges faced by the different approaches to behavioral reflection.

2.1 Continuous behavioral reflection

Behavioral reflection in most systems is *discrete*, which means that reflective computations are initiated at a discrete point by calling a reflective procedure and only lasting until this procedure returns [30]. For example, a method wrapper that makes a method asynchronous only affects the particular wrapped method not all the methods in the whole system. Discrete reflection is not well suited for use cases requiring continuous changes that span the execution of multiple methods or even of a whole application. Continuous behavioral reflection refers to reflective computations that are explicitly inserted in the meta-interpreters, thus having a continuous effect on the base level computation [30]. Software transactional memory, for example, entails a continuous change to the semantics of a language which can benefit from invasive changes to the runtime [19].

Smith's Tower of Interpreters (ToI) approach is fundamentally discrete since the behavior of interpreters can only be extended, not modified. Existing meta-behavior cannot be mutated strictly limiting custom reflective behavior to the base-level code explicitly triggering this custom reflective behavior. Simmons *et al*. extended the discrete reflective tower to a continuous model of reflection by introducing *first-class interpreters* in Refci (reflective extension by first-class interpreters) [36]. In Refci, changes can be applied to the interpreter by explicitly wrapping around the meta-interpreter. These wrappers are then used for the interpretation of all code to which the modified interpreter is applied, thus having a continuous effect on its interpretation.

Refci extends the interface of reifiers with reifications of the interpreter itself in the form of a *preliminary* and a *dispatch* procedure. The preliminary is a function that transforms a dispatch onto another dispatch, and is executed before executing the actual dispatch. The dispatch itself evaluates expressions and as such is a reification of the actual interpreter. By obtaining first-class access to these procedures the programmer can extend the semantics of the interpreter for the duration of the evaluation of the passed expression.

To our knowledge, no further research has taken place to make first-class interpreters practical. The key research problem is how to implement extensible first-class interpreters in a VM in a clean and fast way.

2.2 High-level reflective API

Reflection enabled by mechanisms such as method wrappers [7], proxies [15], or overriding exception handling methods [15] is used in practice only in limited, idiomatic ways depending on the host programming language. To enable widespread use of reflection, a safe and practical reflective API is needed.

Bracha and Ungar claim as a fundamental design principle for reflection that *"meta-level facilities must encapsulate their implementation"* [6]. McAffer justifies this principle as follows:

> *the metalevel has been thought of as a place for making small changes requiring small amounts of code and interaction. We believe that the metalevel should be viewed as any other potentially large and complex application — it is in great need of management mechanisms. [32]*

In an effort to make it feasible to develop libraries and applications that rely on reflection, Kiczales *et al*. proposed the use of *metaobject protocols* (MOPs) to implement discrete reflection.

> *What reflection on its own does not provide, however, is flexibility, incrementality, or ease of use. This is where object-oriented techniques come into their own [25].*

Since its definition, all reflective object-oriented languages have resorted to metaobject protocols to provide discrete reflection. By providing a clear interface to the language, metaobject protocols give the user the ability to incrementally customize the language's behavior and implementation. Metaobject protocols provide discrete reflection since one has to install custom metaobjects wherever non-standard behavior is required.

Open implementations [24] provide a general design principle that moves the black box boundary of objects so that part of their internal implementation strategy becomes open and customizable to the user. For example a `Set` class could allow the user to specify what kind of operations are most common for a particular instance so that instance can be optimized towards that use-case. Open implementations provide discrete customizations, affecting only particular instances rather than the system as a whole.

2.3 Separation of base and meta-level

Typically, discrete reflection is implemented by manipulating base-level code (*e.g*., through source or bytecode transformation). This technique brings forth a whole new set of problems (the same that optimizers introduce for debugging). Most importantly, the application that uses reflection has to keep track of the meta-level on which it is being evaluated to avoid endless recursion [8, 12]. For instance, the code that logs a method execution must avoid itself triggering the logging meta behavior to avoid infinite meta recursion. This problem arises from the lack of a clean separation between base and meta behavior.

Bracha and Ungar argue that "meta-level facilities must be separated from base-level functionality" [6]. McAffer [31] argues that "The implementation of an object must be explicitly exposed and clearly distinguished from the object's domain-specific behaviour description."

While towers of interpreters clearly separate the base-level and meta-level computations, metaobject protocols generally lack this clear distinction, leading to confusion between the two levels [8, 12].

2.4 Summary

Behavioral reflection is most usable in practice when encapsulated through a well-designed MOP. *"All reflective systems therefore provide fixed MOPs: as flexible as they may be, they impose the actual interfaces of metaobjects."* [39] Despite their practical advantages, MOPs inherently offer discrete reflection, lead to meta-level confusion, and are thus limited in their support for extension. These limitations are also inherent to today's VM approaches which establish a rigid barrier between what one may reflect on, and what is hidden in the guts of the runtime.

The following table summarizes the shortcomings of the various approaches.

	ToI	MOP	Refci
Discrete reflection	✓	✓	✓
Continuous reflection			✓
High-level reflective API		✓	
Separation of levels	✓		✓

3. PINOCCHIO in a Nutshell

PINOCCHIO eliminates the barrier between applications and the VM by using first-class interpreters to realize behavioral reflection. The VM core implements a dialect of Smalltalk-80 [17]. PINOCCHIO supports the runtime implementation and use of interpreter extensions.

Instead of interpreting bytecodes, PINOCCHIO directly interprets abstract syntax trees (ASTs) that more faithfully represent Smalltalk-80 code. The core interpreter is implemented in C, and is reified in the runtime as a first-class interpreter. The interpreter provides a basic MOP to support structural reflection. Unlike most interpreters that are based on the assumption that the VM is a black box isolated from the runtime system, PINOCCHIO supports behavioral reflection by opening the interpretation of code to the runtime. Behavioral reflection is supported by explicitly instantiating first-class interpreters that subclass the reified core interpreter. Extending interpreters is facilitated since AST nodes are semantically closer to the original source code than bytecode [11, 13].

To construct a new variant of the PINOCCHIO interpreter it suffices to subclass the `Interpreter` class and override a part of its interface (see Figure 1). The `Interpreter` class defines a meta-circular interpreter implemented as an AST visitor that manages its own environment but relies on recursion to automatically manage the runtime stack. The meta-circular interpreter reifies the core interpreter written in C, so its methods are actually implemented as native functions that hook into the underlying C interpreter code. From user's point of view the `Interpreter` is fully written in PINOCCHIO itself.

Application code is evaluated by a new interpreter by sending the `interpret:` message to the desired interpreter class with a closure representing the code as its argument. For example, the expression

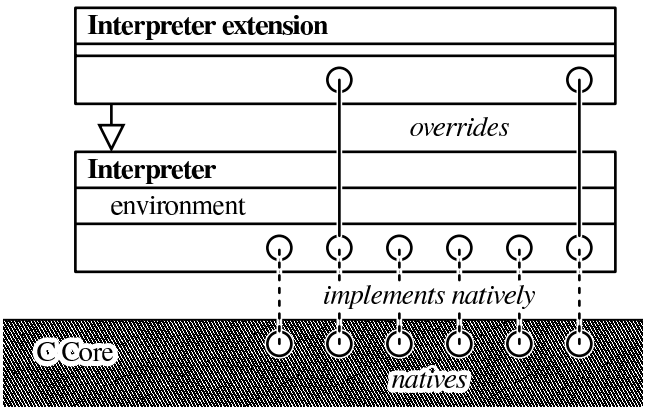

Figure 1. Native methods in the `Interpreter` and interpreter extension through subclassing

```
Debugger interpret: [ self runApplication ].
```

will cause the closure `[ self runApplication ]` to be evaluated by the `Debugger` interpreter.

As usual, closures encapsulate an *environment* and an *expression* object. When starting up a specialized interpreter the continuation of the interpreted application is empty. The interpreter installs the enclosed environment and starts evaluating the expression in this environment. Since the passed expression for the default interpreter is a closure the evaluation is done by sending the message `value` to the closure on top of the interpreter:

```
interpret: aClosure
    ↑ self send: (Message new selector: #value)
           to: aClosure.
```

Although it might seem correct to directly evaluate the closure by invoking `aClosure value`, this is incorrect as the closure would be evaluated at the wrong level of interpretation. It would run at the level of the interpreter (the meta-level from the application's point-of-view) rather than at the application level as desired.

The open design of the meta-circular interpreter lets programmers extend the runtime with very little effort. More importantly, the extensions to the interpreter are implemented within the language provided by the interpreter itself. As such they can be implemented using any of the existing tools for the language, including development environments, debuggers, test-runners and versioning systems.

3.1 Bootstrapping PINOCCHIO

The bootstrapping C interpreter is the first interpreter used in the runtime. It is an instance of the `MainInterpreter` class, a subclass of the `Interpreter` class. The main interpreter does not override the interpretation implementation but provides a prelude that decides how to start interpretation. It uses the command line arguments passed to the executable to decide whether to directly interpret an input file or to start a REPL. As shown in Figure 2, all code passed to the

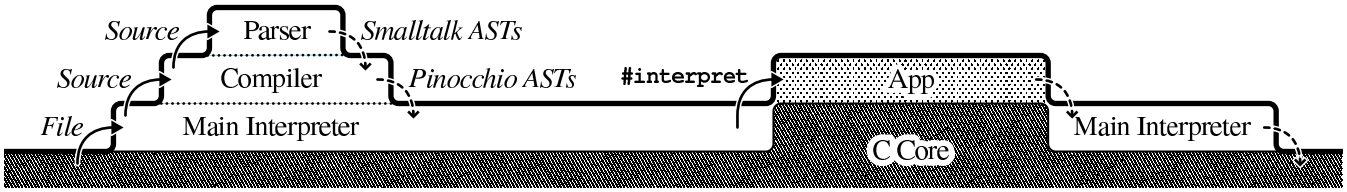

Figure 2. Bootstrapping PINOCCHIO

interpreter in either case is parsed, compiled and interpreted on top of this interpreter instance.

Source code is transformed into the AST using a compiler that runs on top of our runtime, just like the self-hosted Smalltalk compilers. This compiler is completely written in standard Smalltalk code hosted on Pharo [3]. The compiler is then bootstrapped to PINOCCHIO with the help of a bridge that performs the following steps:

- a Pharo class is translated to a Pharo object representing a PINOCCHIO class, and the compiler is used to generate PINOCCHIO ASTs for the methods of the class;
- the bridge then transforms these Pharo objects to C code that uses the internal API of the PINOCCHIO runtime;
- the generated C code is statically compiled into the interpreter;
- when the runtime boots, the generated C code builds the PINOCCHIO version of the original class and its associated ASTs.

The compiler first uses a Parsing Expression Grammar (PEG) [16] that translates the input source code to a full Smalltalk AST. We use a PEG since they provide a simple way of combining parsers into bigger parsers and are well suited for extensions to existing languages [33]. This parsing step is followed by one step of simple semantic analysis and translation to the PINOCCHIO AST.

Once the input source code is loaded it is passed to the superclass of the `MainInterpreter`. Since the main interpreter does not specialize any of the interpretation methods, this comes down to executing the code directly on the C interpreter. Its function is merely to provide a location to put the prelude of an interactive interpreter.

3.2 Minimizing the Interpreter Stack

Our approach of starting new interpreters on top of other interpreters is similar to, albeit the inverse of, the tower of first-class interpreters in Refci [36]. This has the advantage that we can use the same approach to minimize the height of the tower that is actually running at each point in time. For example, since the `MainInterpreter` does not alter the interpretation behavior of the standard interpreter, the evaluation of the application can happen fully at the C level, dropping the `MainInterpreter` from the active interpreter stack. It is important to never run on a stack bigger than necessary, since each extra level of interpretation has a steep price in terms of performance.

In PINOCCHIO the height of the tower is pragmatically minimized by making the whole definition of the standard interpreter available as a fine-grained set of natives installed on the `Interpreter` class (see Figure 1). Only the extensions to the interpreter are evaluated meta-circularly.

Since most custom interpreters will only partly alter the semantics of existing native methods, the default implementation in charge of invoking natives, `invokeNative`, allows interpreters to rely on meta-meta-level implementations of natives to provide the behavior to the base-level. In other words, whenever an application ends up in code that invokes a native, the interpreter can ask its meta-interpreter to perform the actual invocation of the native. This temporarily drops the interpreter from the active stack of interpreters.

Since natives are able to send messages back to the application level, every call to `invokeNative` stores the interpreter that triggered the actual native. To ensure that the application always runs on the proper level of interpretation, when the native wants to send a message back to the application it first has to restore the stack of interpreters that was active before invoking the native. An example of such a case is the native implementation of the `at:` method installed on dictionaries. This method needs to be able to request the `hash` value of a key, and later compare it with the keys in the dictionary using the `=` message. Both methods are within the control flow of the native evaluation of the `at:` method. To evaluate both methods at the right level of interpretation, the stack of interpreters that was active before the `at:` was invoked needs to be reconstructed before their evaluation is started.

3.3 PINOCCHIO Metaobject Protocol

Aside from providing access to user-definable and runtime instantiatable first-class interpreters, PINOCCHIO provides a default metaobject protocol that is sufficient for many reflective use cases. From the point of view of the programmer, the main interpreter is written as a meta-circular AST visitor. New semantics can easily be added to the language by replacing standard application constructs such as methods with custom metaobjects following the same metaobject protocol. The following extension points are noteworthy.

First-class AST nodes. New nodes can be defined by following the visitor protocol. The new nodes could be gener-

ated by extending the default parser and compiler. This could for example be used to provide mutable AST nodes or *link* objects for partial behavioral reflection (see subsection 6.2).

Does not understand. Following Smalltalk-80, our core interpreter sends the `doesNotUnderstand:` message to any object that does not implement a method corresponding to the selector of a message sent to it. This is an important feature to make PINOCCHIO compatible with existing Smalltalk code.

First-class slots. Unlike most Smalltalk systems, which rely on *magic numbers* to encode the layout of instances, PINOCCHIO's class layouts are described using *layout* metaobjects. These metaobjects further rely on *slot* metaobjects that define the semantics of instance variables. Whenever a reference to an instance variable is made in the application's source code, the compiler directly inserts the corresponding slot metaobject into the resulting method's AST.

Every slot metaobject can override the AST node evaluation protocol to provide custom semantics for retrieving the instance variable. Custom semantics for the assignment to instance variables are implemented by overriding the protocol provided by the interpretation of the `Assign` AST node.

By providing explicit layout and slot metaobjects, applications can easily decide what kind of layout and accessing semantics to attach to specific classes. Special behavior such as first-class relationships or singletons can cleanly be factored out into slot libraries to avoid cluttering of the code referring to the slots.

4. Implementing Custom Interpreters

In this section we present three different customized interpreters implemented in PINOCCHIO. We first introduce a simple debugger that absorbs garbage collection and the object model, and relies on straightforward recursion to manage control flow. The alias interpreter shows how interpreters with custom object models are implemented. Finally we outline how interpreters relying on access to the runtime stack are supported in PINOCCHIO through modifiable interpreters and the availability of first-class continuations. The full sources of these use cases are available under `http://scg.unibe.ch/download/pinocchio/pinocchio_svn1397_mc196.zip`.

4.1 A Simple Debugger

To show how extensions to existing interpreters are implemented and used we first describe the implementation of a simple debugger. It executes a program while allowing the user to pause evaluation at the level of message sends. In order to start evaluating code using a debugger the user passes the code to the debugger in the form of a closure:

```
Debugger interpret: [ self runApplication ].
```

The debugger takes control over the evaluation of the block. At each message send it allows the user to decide to step to the next message send, to inspect the current receiver, to step over the evaluation of the message send or to evaluate PINOCCHIO statements. This is a typical subset of actions available in any debugger.

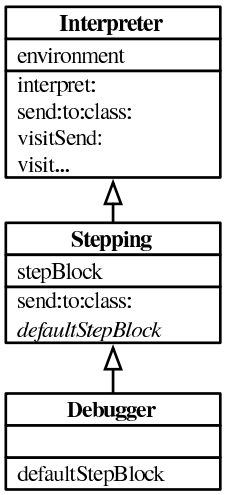

Figure 3. Specializing a Pinocchio interpreter

As shown in Figure 3, to implement the debugger in PINOCCHIO, we start by creating the stepping interpreter class as a subclass of the standard `Interpreter`. The stepping interpreter overrides the methods in charge of evaluating message sends. Rather than directly executing a send, the stepping interpreter delays this behavior and first gives control to a `stepBlock` installed on the interpreter instance:

```
send: message to: receiver class: class
    ↑ stepBlock value: receiver
                value: class
                value: message
                value: [ super
                            send: message
                            to: receiver
                            class: class ].
```

The `stepBlock` of the interpreter can be used to flexibly modify the message send semantics of the running interpreter. Subclasses of the stepping interpreter can define a custom default `stepBlock` and replace the `stepBlock` at runtime.

The debugger itself is implemented by providing different kinds of blocks to the stepping interpreter. The default `stepBlock` of the debugger is implemented as follows:

```
defaultStepBlock
    ↑ [ :receiver :class :message :action |
        self print: receiver class name, '>>', message.
        self debugShellWithAction: action ].
```

It first displays information about the current message send by printing out the receiver's class and the message including the selector and arguments. Then the debug shell is launched a simple *read-eval-print*-loop (REPL) that accepts certain debug actions, as well as PINOCCHIO statements as input.

Since this REPL runs within the execution context of the interpreter, the current execution of the application is temporarily halted until the REPL eventually returns and decides to evaluate the `action`.

Other types of debug actions can be implemented using different stepping blocks. The follow method implements the *step over* behavior. It lets the debugger execute an entire application-level recursive call without prompting the user about its evaluation.

```
stepOver: overAction
    |result previousBlock|
    previousBlock := stepBlock.
    stepBlock := [ :receiver :class :message :action |
                    action value ].
    result := overAction value.
    stepBlock := previousBlock.
    ↑ result
```

It locally stores the previous stepping strategy and installs a block that skips all steps. Whenever the application finishes the recursive call that triggered the current step the control flow will automatically end up back in this method restoring the block to the previous version and continuing.

Evaluation This way of implementing a debugger is straightforward and only requires very little code to add new flexible features. The whole implementation of the debugger adds around 50 lines of code to the stepping interpreter, which adds another 30 to the default interpreter. Since the debugger is just another interpreter it can be passed in at any level of interpretation. As such it can be used not only debug a user program, but also the interpreter running it. Naturally this allows for the debugger to debug itself.

4.2 Alias Interpreter

As second use case we show how to implement Object Flow Analysis [27] in PINOCCHIO. Object Flow Analysis is a dynamic analysis that tracks the transfer of object references at runtime. It has been employed for various reverse engineering approaches and for the implementation of an efficient back-in-time debugger [28].

The problem tackled by Object Flow Analysis is the fact that in code with assignments it is hard to track where a certain value comes from. A debugger only shows the current call stack and hence often does not reveal the context in which a field was assigned. While execution traces show exactly how the interpreter goes through the code they do not show how the values are stored and retrieved. For example, to understand where a certain value of an instance variable comes from, we need to look at all the source code that might have triggered a store. In an alias interpreter (the back-end used by Object Flow Analysis) object references are represented by real objects on the heap. These objects, referred to as *aliases*, keep track of the origin of each reference in memory.

To know where each value comes from the alias interpreter alters the semantics of the interactions so that it generates aliases for:

- allocation of objects and their instance variables,
- reading and writing of fields,
- passing of arguments,
- returning of return values, and
- evaluation of literals (constants).

Rather than directly passing around actual values, in the interpreter objects are wrapped into alias objects.

We chose Object Flow Analysis as second use case because it requires deep changes in the interpreter and its object model. This case lets us evaluate how flexible our approach is for extending low-level details of the runtime and how much less effort is required to realise these changes compared to the original implementation.

An Alias Example Suppose we have a class `Person` with one instance variable `name` and simple accessors for `name`, consider for example the following code:

```
testMethod
    ↑ AliasInterpreter interpret: [ |person|
        person := Person new.
        person name: 'John'.
        person name: 'Doe'.
        person ].
```

In this excerpt, the block is evaluated in the context of an alias interpreter. All the values used by the alias interpreter are aliased. When the result is returned from the alias interpreter it is not unwrapped so we can inspect the aliasing in the instance. The resulting alias graph, as shown in Figure 4, contains the following information:

```
return2 := self testMethod.

self assert: (return2 isKindOf: ReturnAlias).
self assert: (return2 environment selector = #testMethod).

person := return2 value.
self assert: (person isKindOf: Person).

return1 := return2 origin.
self assert: (return1 isKindOf: ReturnAlias).
self assert: (return1 environment selector = #new).

fieldWrite2 := person name.
self assert: (fieldWrite2 isKindOf: FieldWriteAlias).
self assert: (fieldWrite2 value = 'Doe').

fieldWrite1 := fieldWrite2 predecessor.
self assert: (fieldWrite1 isKindOf: FieldWriteAlias).
self assert: (fieldWrite1 value = 'John').

allocation1 := fieldWrite1 predecessor.
self assert: (allocation1 isKindOf: AllocationAlias).

parameter1 := fieldWrite1 origin.
self assert: (parameter1 isKindOf: ParameterAlias).
self assert: (parameter1 value = 'John').
```

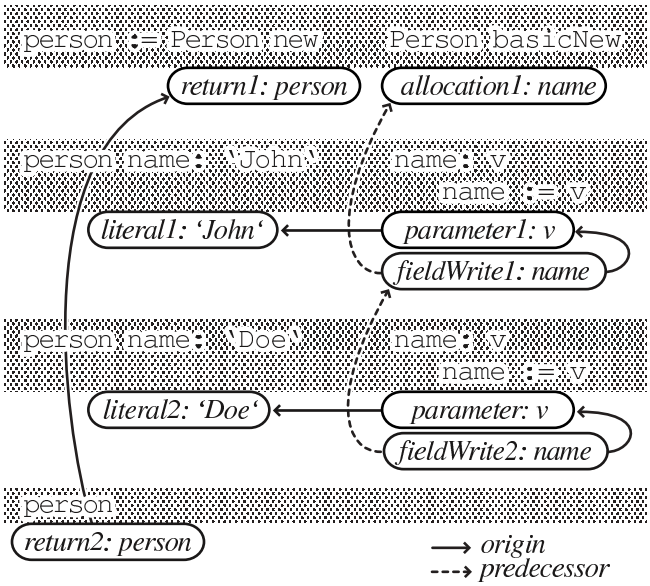

Figure 4. Alias Graph (*origin* denotes where an alias comes from, *predecessor* of a field write alias is the alias that was previously stored in this field)

```
literal1 := parameter1 origin.
self assert: (literal1 isKindOf: LiteralAlias).
self assert: (literal1 value = 'John').

parameter2 := fieldWrite2 origin.
self assert: (parameter2 isKindOf: ParameterAlias).
self assert: (parameter2 value = 'Doe').

literal2 := parameter2 origin.
self assert: (literal2 isKindOf: LiteralAlias).
self assert: (literal2 value = 'Doe').
```

All the gathered information can be used by a debugger to provide means to navigate through the tracked flow of objects. This can easily be used to track for example where null-pointers come from, since all objects are accounted for by aliases.

Linguistic Symbiosis To track aliasing the interpreter wraps all objects into alias objects. This makes the object model of the alias interpreter differ significantly from the default interpreter.

PINOCCHIO's object model provides structural reflection similar to that of Smalltalk. This feature is a requirement for *symbiotic reflection* [18, 43]: applications have to be able to start a new interpreter and pass themselves as applications. The new interpreter starts by running at the base-level of the application, but as the application passes itself to the new interpreter it becomes part of the meta-level of the application. The new interpreter makes use of base-level structural reflection to interpret the code of the application.

Symbiotic reflection is typically used when the language of the meta-level differs from that of the base-level, for example, when Java is used to interpret a dynamic language. Objects from the meta-level (*e.g.*, Java) typically need to be wrapped before they can be used at the base-level, and unwrapped to be manipulated at the meta-level. This process is known, respectively, as *upping* and *downing*.

In PINOCCHIO the base- and the meta-languages generally differ only in limited ways at the meta-level leaving most of the base-level semantics unaltered. This allows many of the user-defined interpreters to let objects flow freely from the meta-level to the base-level and back, transferring or sharing ownership of the same object without any wrapping or unwrapping. This is the case for the debugger in the previous section.

In case that the base- and meta-levels of a PINOCCHIO interpreter diverge significantly however, it is entirely up to the interpreter to correctly realize the required upping and downing. The alias interpreter is such an example. Rather than directly passing objects from the meta-level to the base-level, all objects passed around in the alias interpreter have to be wrapped into alias objects. When the base-level application performs native actions on aliased objects they first need to be unwrapped by the interpreter.

***Aliasing in* PINOCCHIO** The implementation of an alias interpreter using PINOCCHIO is fairly straightforward. First all interpreter methods that are related to one of the tracked actions (object allocation, reading and writing of fields, passing as argument, evaluation of literals (constants) and returning from method) are overridden to generate the aliases. For example, the method that interprets methods is overridden so that it returns a `ReturnAlias` instance wrapped around the result:

```
interpretMethod: aMethod
    | result |
    result := super interpretMethod: aMethod.
    ↑ (ReturnAlias alias: result)
        environment: environment.
```

Notice that the actual semantics of the interpretation of methods is just inherited from the standard interpreter.

All methods that need the actual values inside the aliases are overridden to first unwrap the aliases. Aside from methods related to the evaluation of natives, all interpreter methods only need the value of the current `self`. This is shown in the following method that is invoked by the interpreter whenever it evaluates an assignment to a slot. It assigns the actual value to the slot of the current `self` by first unwrapping the aliased `self` and then wrapping the value in a `FieldWriteAlias`:

```
assignSlot: aSlot to: anAlias
    | alias unwrappedSelf |
    unwrappedSelf := self currentSelf value.
    alias := (FieldWriteAlias alias: anAlias)
              environment: environment.
    alias predecessor: (aSlot readFrom: unwrappedSelf).
    ↑ aSlot assign: alias on: unwrappedSelf
```

The alias interpreter uses a different object model for the base-level than for the meta-level. As explained in the previ-

ous paragraph this requires the alias interpreter to realize the upping and downing by itself. Every time an object moves from the meta-level to the base-level it needs to be wrapped to look like the other objects in the runtime (like aliases, in this case).

There are two places where objects potentially flow from the meta-level to the base-level. The first is the initial closure passed to the interpreter. The closure is linked to an environment that contains objects possibly referred to by the code of the closure. Rather than directly sending `value` to the closure, we first have to pre-process it:

```
interpret: aClosure
    ↑ self send: (Message new selector: #value)
          to: aClosure asAliased
```

The `asAliased` message will deep-clone the closure, and wrap the closure as well as all the values referred to by its environment into allocation aliases. We use allocation aliases since we are unsure of the origin of the objects. As it is the initial state of the alias interpreter, from the perspective of the alias interpreter it is as if the objects were allocated at that point in time. This indicates that users of interpreters that rely on a modified object model have to be careful not to pass a closure along that has references to huge object graphs or ensure that deep-cloning is not required by the interpreter.

The second place where objects flow from the base-level to the meta-level and back is the evaluation of natives. To support the interpretation of natives, the original alias interpreter overrides most of the supported methods and performs the correct action. Since not all natives have the same semantics, no single implementation can properly support the evaluation of all of them. To complete our implementation we would also have to provide new implementations for the subset of operations we support. For this experiment, however, we limited ourselves to the general solution that works for most examples. Whenever a native is called the receiver as well as the arguments are downed and passed to the implementation of the meta-interpreter. The result returned from this native is upped by wrapping it into an allocation alias and passed to the application.

Evaluation The original Object Flow Analysis has been implemented by directly extending the Pharo VM [3]. It required changes of a large amount of the VM code and took several weeks to implement. The PINOCCHIO version on the other hand was implemented in less than one day. It is spread over 20 methods and 12 alias data classes.

One of the main ideas behind the alias interpreter is that it allocates the aliases on the heap so they are automatically garbage collected when their state becomes irrelevant to the state of the application. Because the extension is at the VM level new objects have to be manually instantiated at that level. Since referring to specific Smalltalk classes is cumbersome from within the interpreter, the original interpreter just provides one class fitting all types of aliases. This class has a special field designated to indicate the actual alias type. In PINOCCHIO, the alias interpreter is implemented in a standard Smalltalk environment. This allows the programmer to rely on the full expressiveness of the language: all aliases are instances of classes representing their specific type. Since PINOCCHIO interpreters absorb garbage collection from the main runtime it is also automatically used for collecting the aliases.

In contrast to the original alias debugger the PINOCCHIO version is fully hosted within the language itself. This allows us to use the standard tools for implementing, and more importantly, for debugging the alias interpreter. Now that the alias interpreter is functional new tools or even alias interpreters can be debugged using the current alias interpreter. This is not possible in the original Smalltalk version, since their alias interpreter extensions are written in C, and are thus not subject to the (modified) Smalltalk interpreter.

4.3 Recursive Interpreters

Behavioral reflection in Smalltalk entails manipulation of the (reified) runtime stack. In Smalltalk-80, the state of the computation is fully captured by the runtime stack. As a consequence, the Smalltalk bytecode interpreter does not need to keep track of any control flow itself and automatically adapts to reflective changes to the runtime stack. The disadvantage of this approach is that to be able to adapt to such reflective changes easily, the evaluation of the interpreter is completely decoupled from the evaluation of the application. Not only must bytecodes explicitly manipulate the stack, but code that passes control from the meta-level to the base-level must be "ripped" into event handlers that can run to completion without blocking [2]. Since base-level code in Smalltalk-80 can make arbitrary changes to the runtime stack, the bytecode interpreter must be prepared to resume execution in any arbitrary context. If the interpreter wants to keep track of state related to the evaluation of the application other than what is available in the standard stack frames it also has to manually keep track of this data and keep it in sync with the application's execution.

PINOCCHIO's use of first-class interpreters with *automatic stack management* [2] greatly simplifies the expression of behavioral reflection. Code that passes control from the meta-level to the base-level can be straightforwardly implemented by relying on recursive calls. PINOCCHIO interpreters can simply rely on recursion to keep track of any state related to the application's control flow just like any other Smalltalk application.

The disadvantage of this approach is that it becomes impossible to directly perform operations normally requiring explicit stack manipulations since there is no explicit stack. We identify two types of direct stack manipulation in Smalltalk: applications need to be able to (i) capture a certain state of the stack and later restore it, and (ii) capture a stack and pass it to another program, a meta-circular interpreter, for reflective evaluation of the application. In this section we show that PINOCCHIO supports these two requirements re-

spectively through first-class continuations [9] and modifiable first-class interpreters.

Parallel Debugging An example of a situation where a user would like to capture and restore the state of a stack is a *parallel debugger*. Unlike the normal debugger, which only evaluates one block at a time this special kind of debugger takes two blocks and interprets them in parallel comparing the state of evaluation at each step.

Consider the following failing test case that we encountered during the development of PINOCCHIO:

```
dict := SetBucket new.
dict at: #key put: 'value'.

self assert: (dict includes: #key).
self assert: (dict includes: 'key').
```

The second assertion (last line) fails. This test was documenting a bug that we had difficulties to track down. Symbols and strings are considered equal (`#key = 'key'`) in Smalltalk and hence the second assertion should pass too.

Using the basic debugger described in subsection 4.1 to find the difference in execution of the two assertions is cumbersome. The manual approach would be to launch a separate debugger for each of the assertions and step through the code until the states of the tests differ.

Since we had difficulties tracking down the root cause of this bug we implemented a specialized debugger that we call *parallel debugger*. The use of the parallel debugger for the previously mentioned test case looks as follows:

```
ParallelDebugger interpret:
    (Array
        with: [ dict includes: #key ]
        with: [ dict includes: 'key' ])
```

The debugger runs the given blocks in parallel up to the point where the executions start to differ:

```
SetBucket>>#includes:
   SetBucket>>#do:
      SmallInt(1)>>#to:do:
         BlockClosure>>#whileTrue:
            SmallInt(1)>>#<=
              SmallInt(1)>>#>
              --> false
              false>>#not
              --> true
            true>>#ifTrue:
              SetBucket>>#at:
              --> #'key'
              Symbol(#'key')>>#==
              1) (#'key')--> true
              2) ('key') --> false
```

Listing 1. Parallel debugger trace

Looking at this trace immediately reveals that both traces differ upon a strict equality check on a symbol. In the first case the comparison returns `true`, in the second case `false`. SetBucket incorrectly uses `==` (pointer equality) rather than `=` to compare keys, rendering strings and symbols distinct. The parallel debugger provides the minimal

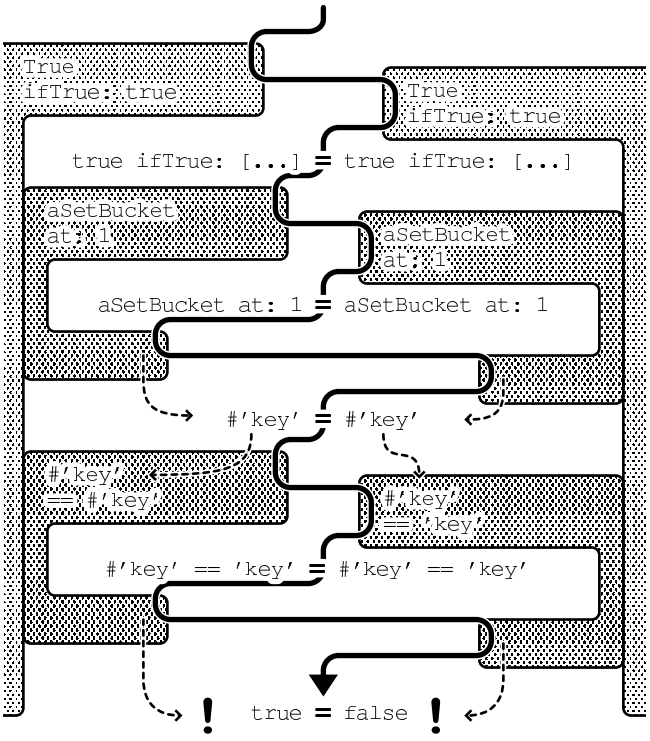

Figure 5. Thread-based parallel execution of two code parts in the parallel debugger.

output needed to quickly identify the root cause of the problem.

To implement the parallel debugger we need to be able to evaluate multiple closures at once. In an interpreter with manual stack management this is straightforward. Rather than interpreting the code of one interpreter in a loop, one lets all interpreters do one step of evaluation before comparing their states. PINOCCHIO however relies on automatic stack management and thus relies on recursion to evaluate the closures. This implies that the parallel debugger needs to be able to jump out of, and back into a certain recursion state. This problem is similar to implementing coroutines in a recursive language. The only difference is that the parallel debugger itself handles thread-switching before and after each message send. Coroutines and threads can easily be accommodated in recursive languages through the use of first-class continuations [20].

Just like the debugger described in subsection 4.1, the parallel debugger is built as a subclass of the stepping interpreter. The main difference is that the stepping block is not used to control a single execution trace but to handle the interleaved execution of the given number of closures. Before and after each message send we store the state of the current green thread by capturing its continuation and the application's environment, and resume the next thread by restoring its application's environment continuing its con-

tinuation. Whenever we resume the first thread we compare the state of all the routines and continue with the first thread.

The parallel debugger, like the serial debugger presented in Section 3, directly reuses the object model of the underlying base-level interpreter. As a consequence, no upping or downing is required, and objects can freely flow between the base- and meta-levels.

Even though interpreters are defined recursively as AST visitors, this poses no problem for expressing non-local flow of control. Threads are easily simulated by capturing the needed continuations and explicitly transferring control when needed. The parallel debugger is only possible due to the support for continuations in PINOCCHIO. Without continuations we could not switch between the execution of multiple closures. It would only be possible to continue the execution of the next closure from inside the current one.

Runtime Modifiable Interpreters Meta-circular interpreters such as the Smalltalk debugger are a second type of application that require direct access to an explicit stack. The Smalltalk interpreter can pass control over the evaluation of an application to a debugger by passing its runtime stack to the debugger. The Smalltalk debugger however is fully meta-circular and has to manually manage this runtime stack for the evaluation of the application. The advantage of this approach is that since the meta-circular as well as the core interpreter are both decoupled from the state of the application's stack the meta-circular interpreter is given full control over the evaluation of the application. While PINOCCHIO's first-class interpreters only have continuous behavioral impact on the evaluation until the closure finishes, meta-circular interpreters with manual stack management can evaluate the program beyond the continuation where the interpreter was started. This is a useful feature as is obvious from the Smalltalk debugger. Whenever an error occurs, the Smalltalk debugger can take over the evaluation as if it had been running the application all along, although the core interpreter has mostly executed the program up to that point.

If in PINOCCHIO we would like to start a new interpreter to change the interpretation semantics, the change would be limited to the scope of the control flow in which they were activated. This is undesirable for debugging purposes since we would not be able to step through more of the program than the recursion of the message send that caused the error. We rather want to change the semantics of an interpreter while it is running.

In PINOCCHIO modifiable interpreters are accommodated by letting the user specify which parts of an interpreter are mutable. The stepping interpreter discussed in subsection 4.1 is an example of an interpreter whose semantics can partly be modified while it is running. Its `stepBlock` influencing the semantics of message sends is orthogonal to the control flow of the application, leaving it unaffected by the interpreter exiting the control flow where the `stepBlock` was installed. The semantics of the `stepBlock` is only bound to the scope in which its host interpreter is active.

The following example is an extension method to the debugger described in subsection 4.1. It temporarily replaces the current stepping semantics for the duration of a variable number of message sends. It allows a user to specify a specific number of steps to be skipped.

```
skipBlock: count
    |skips previousBlock|
    skips := 0.
    previousBlock := self stepBlock.
    ↑ [ :receiver :class :message :action |
        skips := skips + 1.
        (skips >= count)
            ifTrue: [ self stepBlock: previousBlock ].
        self executeAction: action ].
```

For the duration of the given number of message sends the user is not prompted concerning the evaluation. Once the steps are over, control is returned to the previously active stepping style.

The `skipBlock:` method is a clear example of how the `stepBlock` can be used to apply changes that potentially surpass the control flow in which they were activated. Even if the number of skipped steps is larger than the number of steps needed for the application-level recursive call before which the block was installed, the block will stay active until the requested number of steps are over.

5. Performance

The current implementation of the PINOCCHIO interpreter[1] is only slightly optimized, but most optimization opportunities are left open. The interpreter only implements the evaluation of constants, variables, instance variables, assignment, closures and message sends with monomorphic inline caches [21]. All boolean operations for example are implemented in high-level Smalltalk style as message sends to boolean objects with closures as arguments. None of these messages are currently optimized away by the compiler. Figure 6 compares PINOCCHIO to other high-level language VMs in terms of message sends by running a Fibonacci benchmark. The results are presented relative to the speed of the standard implementation of Fibonacci in Pharo 3.10-3. Since most of the other VMs have dedicated support for conditionals we created a second Fibonacci test (marked with *all sends* in Figure 6) enforcing message sends even for conditionals. Since Python prohibits the addition of new behavior to built-in types we were unable to modify the benchmark accordingly. With this test we provide a fair comparison to the current implementation of PINOCCHIO which does not yet feature any of these optimizations. Based on this test PINOCCHIO is slightly faster than Ruby 1.9 (with all message sends) and magnitudes faster than the older Ruby 1.8.7.

[1] Revision `r1397`, `http://pinocchio.unibe.ch/svn/pinocchio`

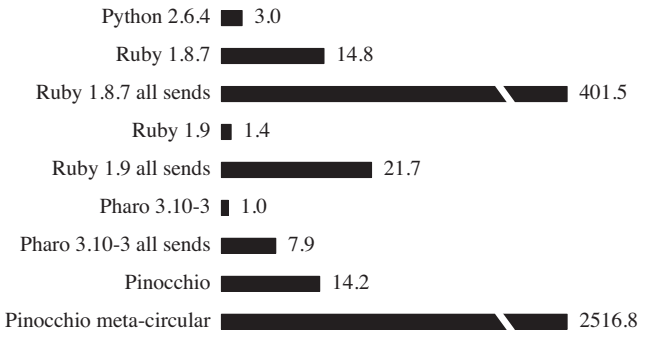

Figure 6. Fibonacci benchmark testing the speed of message sends. Values are given relative to Pharo 3.10-3.

Every added level of interpretation in PINOCCHIO causes a constant overhead. Our `Interpreter` class should have the same speed as the C interpreter since it is a direct reification of the C interpreter without any changed semantics. At this time, however, this interpreter is still mostly meta-circular making it around 160 times slower than the C interpreter since all application-level actions including constant evaluation require at least 3 message sends. Where the C interpreter performs 89 messages to calculate the sixth Fibonacci number, the meta-circular interpreter internally performs $9,665$ message sends to perform those 89 sends, that is, around 108 times more message sends.

5.1 Inline Caching

The overhead of the meta-circular interpreter can be amortized by linking all interpretation methods back to the C code. This would make the meta-circular interpreter exactly as fast as the C interpreter. To let custom interpreters benefit from the same speedup the C-level interpretation code should only meta-circularly evaluate the custom extensions to the interpreter. The C-level interpreter can be made aware of such extensions by using C-level polymorphic inline caches at reified functions that map classes of interpreters to their concrete implementation of the reified method. If the method is not overridden, the C-level interpreter continues evaluation at the C-level. Only when a custom implementation is provided is the interpreter popped from the meta-continuation and reactivated. This limits the interpretation overhead of custom interpreters to their custom features.

The stepping interpreter for example needs $10,995$ message sends to evaluate the sixth Fibonacci number. This is $1,330$ sends more than the normal meta-circular interpreter, or 15 message sends per application-level message send.

The number of extra message sends is not surprising given that even all boolean operations are currently implemented as message sends. General language optimizations such as avoiding message sends whenever possible dramatically decreases the number of sends on both the application level as well as the interpreter level. The message `ifTrue:ifFalse:` in combination with `value` sent to the closure accounts for 30 of the 89 application-level messages. In most Smalltalk implementations these messages are optimized away, thus already removing 34% of the steps. The same is true for the messages used in the implementation of the stepping interpreter itself.

5.2 JIT Compilation

Unlike the core interpreter, PINOCCHIO's interpreter extensions are interpreted. It is interesting to look into dynamic optimization techniques that can combine compiled versions of the extensions with the native version of the interpreter into one fast customized interpreter.

One can imagine running a JIT compiling interpreter that is hosting another instance of the same class. This second interpreter then runs the application. The first interpreter will turn all extension methods of the second into native methods that are embedded into the native interpreter definition. This makes the second interpreter a native JIT compiling interpreter. This interpreter would then evaluate the actual application and JIT compile the relevant parts.

Customized interpreters would however not be able to directly benefit from JIT compilation of the application. All the custom interpreters would have to provide their own JIT compiler plug-ins so that the embedded semantics are correct. This would call for another type of JIT strategy. Bolz *et al*. propose JIT compilation of the meta-level [5]. In their approach, the meta-interpreter applies a tracing JIT to the interpreter between looping constructs of the interpreter definition. This automatically embeds the semantics of the interpreter into the application, since the meta-interpreter is evaluating both at once.

6. Related Work

We summarize several of the key approaches to reflection and elaborate the differences to PINOCCHIO.

6.1 AOP

Aspect-Oriented Programming (AOP) [26] targets modularization of crosscutting concerns. It consists of two main parts, a set of *advices* that change the behavior of programs and a *pointcut language* that declaratively defines where the aspect system has to *weave* in the advices.

The power of AOP mainly comes from the pointcut language that makes the way shadow points are selected for local modifications easier and more understandable by being declarative.

Since reflection provides support for changing the structure and behavior of programs, AOP can be seen as a principled, structured, and language-supported way of doing meta- and reflective programming [34]. AOP can be implemented by providing a pointcut language as a declarative front-end to reflection [38].

The main difference between PINOCCHIO and AOP is that AOP is designed towards discrete reflection while PINOCCHIO handles continuous reflection.

6.2 Partial Behavioral Reflection

During a workshop on reflection Smith mentioned that in the wide spectrum of reflective applications most applications only need a fragment of the information that can be provided by the interpreter [22]. Since reification of information is expensive due to wrapping into special objects, partial behavioral reflection tries to limit the number of reified objects and message sends needed during the execution of a program.

Partial behavioral reflection provides a reflective model that enables local extensions to the code by attaching metaobjects to operations through links. A link conditionally lets the metaobject decide over the evaluation of the operation that activated the link [40]. In the original model links are installed in the code at class-loading time. Unanticipated partial behavioral reflection extended the model by allowing dynamic installation and retraction of links [35]. The model was also further refined to hook into the high-level AST representations of the code rather than low-level bytecodes [10].

Both partial behavioral reflection and AOP essentially apply structural intercession to the base-level code: they do not alter the evaluation of the base-level code but rather change the code to incorporate the new behavior. The developers of AspectJ even take pains to distinguish the implementation of AspectJ from classical computational reflection [26].

As explained in subsection 3.3, PINOCCHIO's first-class interpreters and partial behavioral reflection are not mutually exclusive, on the contrary. Whenever a reflective change only locally affects code the change should only have a local overhead. PINOCCHIO does however cleanly solve problems that arise in pure partial behavioral reflection by avoiding meta-confusion in the first place and by providing a model for global modifications.

6.3 Generating Interpreters

Douance *et al*. [14] state it is useful to build new interpreters that embed new reflective capabilities in an effort to optimize the amount of information that is actually reified. They propose to implement specific changes by modifying a meta-circular interpreter that is compiled to a new interpreter for each specific metaobject protocol.

The idea of generating real-world VMs from meta-circular definitions has been explored to a greater extent in several projects [4, 23, 41]. They split their energy into building a meta-circular VM and building a compiler toolchain that can optimize the high-level meta-circular interpreter to be at least as fast as (and potentially faster than) a manually written interpreter. The difference in the various projects lies in the language being implemented by the VM, and more importantly the expressiveness of the actual subset of the language used in the definition of the meta-circular VM.

While this approach provides programmers with the ability to change a high-level version of the interpreter to incorporate new language features, these changes have to be applied at compile-time. The semantics of the resulting interpreter cannot be changed anymore from within applications running on them. This again results in multiple interpreter sources and applications are unable to use different versions of the interpreter for different subparts of the applications. The expressiveness of the language accepted by the compiler has a great impact on the ease of development.

Since also for PINOCCHIO a first interpreter that reifies itself needs to be bootstrapped, the technique of compiling meta-circular interpreters to standalone optimized interpreters could be reused. In PINOCCHIO however all extensions to the interpreter are applied at runtime, avoiding the problems exhibited by the compile-time modification approaches. Modifications are easily implemented and tested through use of the complete expressiveness of the host-language. The modifications can be used in combination with other custom interpreters since they live in different subparts of the same runtime. This setup does have implications on the direct performance of the extensions, which are hopefully alleviated by the optimizations discussed in Section 5.

6.4 Meta-circular Interpreters

Meta-circular interpreters [1] such as the Smalltalk debugger and all uncompiled versions of the interpreters described in subsection 6.3 are a way of easily allowing changes to the semantics of a language at runtime from within the language itself. They generally reify a subset of features from the host-language, and absorb the complementary set of features.

The first problem with meta-circular interpreters is that to modify their semantics one generally needs to directly modify the source code. No extension mechanisms are provided by the interpreters themselves. PINOCCHIO on the other hand provides a clear protocol for extension through subclassing.

The second problem is that meta-circular interpreters impose a large runtime overhead in comparison with standard interpreters. This problems is partly solved by compilation of the full interpreter, as discussed in subsection 6.3, but this results in a compile-time rather than runtime change, resulting in immutable interpretation semantics at runtime.

While in the current implementation of PINOCCHIO the `Interpreter` class is still mostly meta-circular a clear strategy of how performance can be greatly increased has been outlined. We explained how to rely as much as possible on the existing C-level interpreter, essentially removing the meta-circular layer for all code except for the custom interpreter extensions. Then we explained how these extensions to the interpreter can also be optimized at runtime through JIT compilation.

6.5 Tower Approach to First-Class Interpreters

Of all reflective interpreters, PINOCCHIO is most similar to the model proposed by Simmons *et al*. in their prototype Refci [36]. Just like PINOCCHIO, Refci provides access to

first-class extensible interpreters. Rather than starting new extended interpreters the model proposed by Refci follows the interpreter-model of reflection. In Refci interpreters are extended at runtime with new features.

The interface provided by PINOCCHIO for the extension of interpreters is much more fine-grained and practical. Our interpreters follow the standard object-oriented extension strategy of subclassing. An advantage of a more fine-grained extension protocol is that smaller changes to the semantics can be more easily scoped without impacting the performance of the natively provided system. In Refci all code has to go through all extension points before ending up in the native code that handles it, even if no extensions to the semantics of that particular node were planned by the interpreter extension in question.

Unlike Refci interpreters, PINOCCHIO's interpreters are not tail-recursive continuation passing interpreters (by default). Instead they are normal recursive interpreters that rely on the continuation of the interpreter below to maintain its continuation. This greatly simplifies the final definition of the actual interpreter since control flow is handled implicitly. It however does not restrict the power of the interpreter since continuations can be captured and restored. In Refci such an implementation would be unpractical since tail-recursion is used to ensure that the theoretically infinite tower of interpreters can be cut off to a finite stack of actually running interpreters and an infinite meta-continuation of waiting interpreters.

In Section 4.3 we showed how PINOCCHIO interpreters can be made modifiable just like Refci's interpreters. In PINOCCHIO the first-class interpreters themselves can decide what is made modifiable while in the case of Refci the extensions themselves are in control. The modifications applied to an interpreter in PINOCCHIO can have further extent than the recursion in which it was created. In Refci this notion of extended continuations was merely noted as future work.

Refci provides no model to share extensions between different interpreters in the stack. Duplicate changes need to be installed manually in the levels where they are required. In PINOCCHIO subclassing takes care of the sharing of code. Since in PINOCCHIO interpreters are manually stacked these interpreters can be instances of the same interpreter thus automatically sharing extensions.

No effort was made to outline how Refci can be made into a practical runtime. Since all interpreters in Refci are meta-circular it thus suffers from the problems explained in subsection 6.4.

6.6 Dealing with Infinity

Brown [42] is an extension of 3-Lisp that first introduced the *meta-continuation* as an explicit representation of the infinite tower of interpreters. As shown in Figure 7[2], the theoretically infinite tower of interpreters can be implemented as a finite stack of interpreters running on top of a *level-shifting processor* — a non-reflective processor that is able to *shift up* a level whenever a reification occurs in the application. To stay efficient the processor is also able to *shift down* whenever a level of interpretation is not needed anymore. An application should ultimately never run at a level higher than is necessary. Shifting up is implemented in Brown by popping an interpreter from the meta-continuation, a lazy infinite stack of interpreters. Shifting down pushes the unneeded interpreter back onto the meta-continuation.

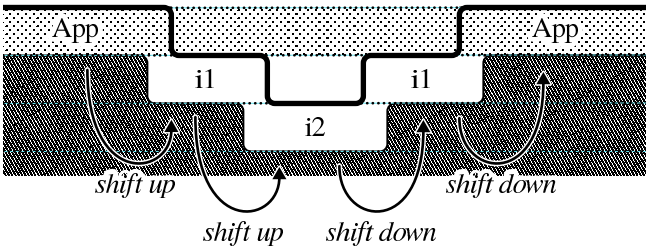

Figure 7. Finite Scope of the Infinite Tower of Interpreters

As explained in subsection 3.2, PINOCCHIO uses this technique to minimize its finite stack of interpreters.

7. Conclusion

PINOCCHIO is a runtime for Smalltalk based on AST interpreters rather than bytecode interpretation. PINOCCHIO eliminates the barrier between VM and runtime by fully reifying the core interpreter. Instead of providing only a MOP for structural reflection, PINOCCHIO provides behavioral reflection by specializing first-class interpreters.

PINOCCHIO inherits its object model from Smalltalk except for "compiled" methods which are represented as ASTs rather than bytecode. Objects can freely flow between base- and meta-levels provided their structural representation remains the same. Otherwise the interpreter is responsible for upping and downing objects between levels. Interpreters are defined recursively, but have a fine degree of control over program flow due to their ability to capture and transfer control to first-class continuations. Garbage collection, and other native features provided by the core interpreter can be simply reused by specialized interpreters.

We have shown through several examples how behavioral reflection provided by PINOCCHIO allows sophisticated behavioral adaptations to be easily implemented. In addition to a serial and a parallel debugger we demonstrated how an alias interpreter, which tracks object flow for back-in-time debugging, can be easily implemented by a specialized interpreter. This stands in contrast to a conventional approach

[2] We turned the tower upside-down since it better matches the mental model that we have of applications and interpreters: applications run on top of interpreters, not the other way around.

in Smalltalk which required invasive changes to create a specialized VM.

PINOCCHIO is presently a proof-of-concept prototype. Although the core interpreter has acceptable performance, no serious optimization effort has been undertaken yet. We have outlined a number of promising tracks that we believe will significantly reduce the overhead introduced by specializing an interpreter.

Acknowledgments

We would like to thank Lukas Renggli and Marcus Denker for kindly reviewing earlier drafts of our paper.

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT electrical engineering and computer science series. McGraw-Hill, 1991. ISBN 0-262-01077-1.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

[3] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0.

[4] C. F. Bolz and A. Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565827.

[6] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[7] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[8] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In K. Futatsugi and S. Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996. ISBN 3-540-60954-7.

[9] C. T. H. Daniel P. Friedman and E. Kohlbecker. Programming with continuations. Technical Report 151, Indiana University, Nov. 1984.

[10] M. Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[11] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, Oct. 2007.

[12] M. Denker, M. Suen, and S. Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008. doi: 10.1007/978-3-540-69824-1_13.

[13] T. D'Hondt. Are bytecodes an atavism? In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*, pages 140–155. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8. doi: 10.1007/978-3-540-89275-5_8.

[14] R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *Higher Order Symbol. Comput.*, 14(1):7–34, 2001. ISSN 1388-3690. doi: 10.1023/A:1011549115358.

[15] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[16] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011.

[17] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0.

[18] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection — a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006. doi: 10.1016/j.cl.2005.10.003.

[19] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. ACM Press, New York, NY, USA, Oct. 2003. doi: 10.1145/949305.949340.

[20] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802046.

[21] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[22] M. H. Ibrahim. Reflection and metalevel architectures in object-oriented programming (workshop session). In *OOPSLA/ECOOP '90: Proceedings of the European conference*

on Object-oriented programming addendum: systems, languages, and applications, pages 73–80, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-443-0.

[23] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997. doi: 10.1145/263700.263754.

[24] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, Jan. 1996.

[25] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0-262-11158-6.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[27] A. Lienhard. *Dynamic Object Flow Analysis*. Phd thesis, University of Bern, Dec. 2008.

[28] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_25. ECOOP distinguished paper award.

[29] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, Dec. 1987.

[30] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection*, pages 1–20, 1996.

[31] J. McAffer. *A Meta-level Architecture for Prototyping Object Systems*. Ph.D. thesis, University of Tokyo, Sept. 1995.

[32] J. McAffer. Engineering the meta level. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, USA, Apr. 1996.

[33] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding languages without breaking tools. In T. D'Hondt, editor, *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *LNCS*, pages 380–404. Springer-Verlag, 2010. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_19.

[34] L. Rodríguez, É. Tanter, and J. Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, Nov. 2004. IEEE.

[35] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 47–65. Springer, 2007. ISBN 978-3-540-71835-2. doi: 10.1007/978-3-540-71836-9_3.

[36] J. W. Simmons, S. Jefferson, and D. P. Friedman. Language extension via first-class interpreters. Technical Report 362, Indiana University Computer Science Department, Sept. 1992.

[37] B. C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982.

[38] A. Strauss. Dynamic aspects — an AOP implementation for Squeak. Master's thesis, University of Bern, Nov. 2008.

[39] É. Tanter. Reflection and open implementations. Technical Report TR/DCC-2009-13, University of Chile, Nov. 2009.

[40] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[41] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. doi: 10.1145/1094855.1094865.

[42] M. Wand and D. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In P. M. North-Holland and D. Nardi, editors, *Meta-level Architectures and Reflection*, pages 111–134, 1988.

[43] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.