Bachelor Thesis

Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

Implementing Pinocchio
a VM-less metacircular runtime library for dynamic languages

Kill Gepetto, Cut the Strings and Run

Olivier Flückiger
December 2011

Supervision:
Toon Verwaest

Abstract

This thesis describes the conceptual ideas and technical solutions behind Pinocchio, a VM-less Smalltalk dialect implemented using a self-supporting runtime library. Pinocchio conceptually unifies the meta-level (the language semantics, the *behind the scenes* part of the language) with the base-level (e.g. a user written application). Both the base-level as well as the meta-level provided by a runtime library are written in plain Smalltalk. The Smalltalk code is compiled directly to machine code in a standard compliant binary format. All objects communicate through a standard protocol that is an extension of the OS application binary interface (ABI) calling conventions. Message sends are compiled to function calls to a meta-level *lookup* ∘ *apply*. After a successful lookup those calls are overwritten with inline-cached calls to the native code of the method. Meta-regression is avoided by type hinting the meta-level code and pre-filling their inline caches at compile-time. We will present the main challenges to support this basic idea of language-construction.

Acknowledgments

I especially thank Toon Verwaest for the opportunity to work on this project, for the encouraging and steady involvement and all the fun. It was amazing to work with you. I also thank Oscar Nierstrasz for supporting me and letting me promote at the Software Composition Group.

Contents

Chapter 1

Introduction

Pinocchio is a Smalltalk dialect, implemented using a self-supporting runtime library written in Smalltalk. Instead of relying on a traditional VM architecture, Pinocchio programs are compiled to native code and execute in a runtime environment similar to the one of a C program. But instead of having the language-semantics statically woven in by the compiler, they are provided by a open and metacircular runtime library. Even the core of the language is implemented in normal, unrestricted Smalltalk. Pinocchio objects send messages by relying on unified calling conventions and a meta-level *lookup* ∘ *apply*.

In Pinocchio the functionality of the VM is fully provided by a meta-object protocol[5]. It consists of Smalltalk objects that implement meta-level tasks such as performing message sends. Since we unify the base- and meta-level, any normal Pinocchio object can be used to fulfill VM-level tasks. The objects flow freely between the meta- and the base-levels. The meta-object library lives next to the user level code, as shown in figure 1.1. Since only the *calling conventions* are fixed, Pinocchio is designed to support different languages in the same environment by multiple meta-level implementations used by different objects.

In a traditional VM the core implementation of the language semantics is statically compiled and conceptually on a different level than user-level code. Even meta-circular implementations of dynamic languages, like Squeak Smalltalk and the Klein VM for Self[9], rely on a static core. Many assumptions about the implemented language are therefore early-bound. They can be made available to an application developer by exposing reflective capabilities. But this approach is limited to the partial reflective hooks introduced beforehand by the language implementors — which will always be a subset of the features a user might need[8]. Additionally, the approach of introducing hooks does not represent a coherent solution, since it results in ad hoc features being introduced for every new entry point.

The core challenge in building a meta-circular environment is self-reference. Consider a *lookup* ∘ *apply* Smalltalk method that handles message sends. To accomplish this task, the method itself is required to send messages to other ob-

jects such as the method dictionary. If this regression is not addressed somehow, the method will end up in an infinite loop calling itself over and over again. This problem is circumvented in existing systems by implementing the core functionality of the system in a restricted early-bound language (e.g. without dynamic dispatch). Pinocchio does not have such a static core with restricted semantics. Instead we provide *static anchors* by reusing inline caches to lay out default execution paths. Since inline caches can fail, and in that case bail out to a fully dynamic message send, even the core of Pinocchio behaves polymorphically and can handle custom user defined objects as replacements for native meta-objects.

This thesis will present the technical solutions we developed in Pinocchio to:

- Compile a dynamic language and its runtime library to native code.
- Create a self-supporting runtime by solving the meta-regression problem without relying on an early-bound core.
- Unify the base- and the meta-level to create a reconfigurable runtime.

This thesis will follow along the implementation of Pinocchio and explain the technical challenges and solutions in building such an environment. We will start by showing how the meta-circular runtime can be constructed and what conceptual and technical challenges needed to be resolved there. Then we describe the compiler and its components and we explain how Smalltalk semantics can be mapped onto the architecture provided by the CPU and OS. Also we will take time to explain some implementation details. The discussion will end with core examples that let us evaluate and use our new found freedom in implementation.

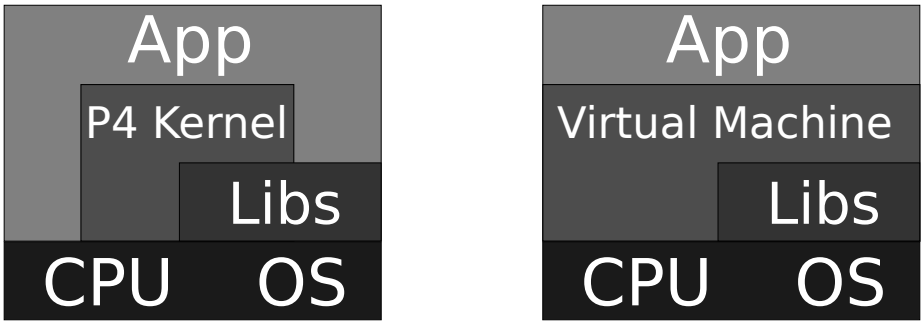

Figure 1.1: In the Pinocchio runtime architecture (left) user- and meta level live next to each other. As opposed to a traditional VM architecture (right) where all features used by an application have to be explicitly exposed through the VM layer.

Chapter 2

The Runtime

Pinocchio features a metacircular, virtual machine-replacing runtime. All core language features are provided by Smalltalk meta objects which have exactly the same format as user level objects.

Objects and code from the user- and meta-levels are unified. Objects can flow between the two and user-level objects can replace any meta-level object. Thus all parts of the support layer — where all the structural and behavioral assumptions of the language are captured — are composed out of accessible and open meta objects.

In Pinocchio since every meta-object is replaceable and message sends are handled by a behavioral meta-object, behavioral reflection is available as a side-effect of the uniformity of the system.

2.1 Message sends

Sending messages is enabled by a *lookup* ∘ *apply* method. The implementation in normal Smalltalk is provided by *invoke:* as listed in figure 2.1.

```
invoke: selector
  <invoke>
  |method behavior|
  behavior := self behavior.
  method := behavior lookup: selector for: self
  ↑ method perform: selector on: self.
```

The actual lookup of the method through the class hierarchy is implemented by a behavior object. By default this is SmalltalkBehavior but a user can provide his own implementation and extend or alter the lookup semantics.

2.1.1 The low-level view

At the assembler level a message send is compiled to a series of instructions that implement the Pinocchio calling conventions. The invocation of a method is compatible to the operating systems ABI. The arguments are passed in argument registers or via the stack.

For example, the following Smalltalk code:

```
receiver selector: argument
```

is compiled into the these X86-64 assembler instructions:

```
mov receiver, %rdi
mov argument, %rsi
// Read selector from literal frame using PIC.
mov $selector(%rip), %rax
// Call to the meta-level invoke function.
call invoke
```

The listing uses *$rdi* as the first and *$rsi* as the second argument register, in accordance to System V AMD64 ABI calling conventions.

2.1.2 The high-level view

The standard Smalltalk method lookup can be expressed in just a few Pinocchio source code lines, as shown in figure 2.1.2.

```
lookup: selector
    |class dictionary|
    class := self class.
    [ class == nil ] whileFalse: [
        dictionary := class methodDictionary.
        (dictionary at: selector)
            ifNotNil: [ :method | ↑ method ].
        class := class superclass ].
    ↑ self doesNotUnderstand
```

Figure 2.1: A meta-level lookup method that browses a class hierarchy and searches for a matching method.

It is important to stress that all messages sent in this meta-level example behave like normal Smalltalk messages. For example the listing shows how the method dictionary of a class is asked for a particular method. Normally this method dictionary object will be a default identity dictionary that contains the methods as value and their selectors as keys. But a user can also construct a class that features other means of storing the implemented methods. As long as the object in question understands the *at:* protocol everything will work as expected. The message send in detail is shown in figure 2.2.

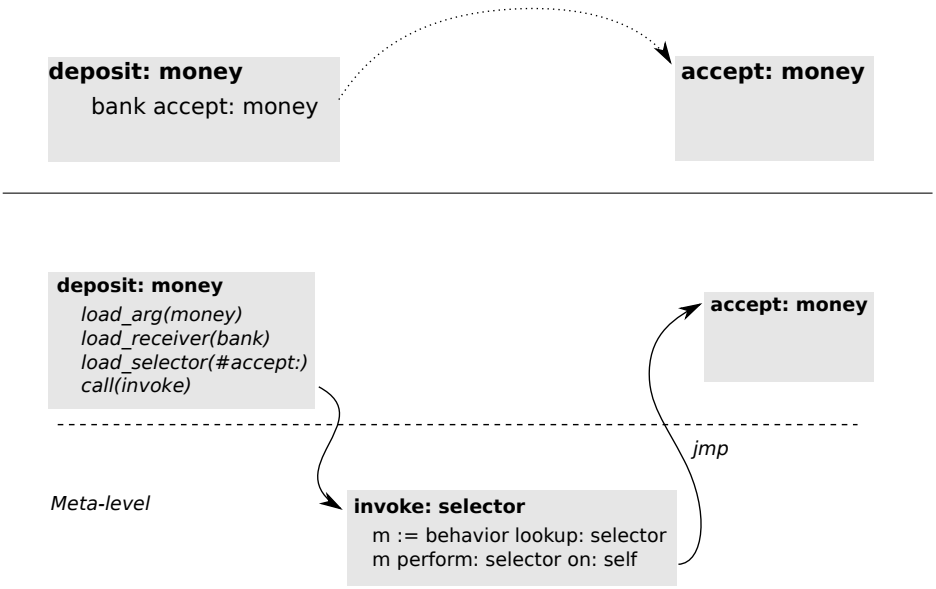

Figure 2.2: *Top:* The message *accept: money* is sent to the bank object. *Bottom:* The lookup ∘ apply mechanism in detail. We retrieve the matching method (on the meta-level, below the dashed line) and jump to its code.

The `<invoke>` annotation in figure 2.1 informs the compiler that this is a special meta–level method that is activated to support a base-level method invocation. Invoking the method is equivalent to crossing the dashed meta-boundary line in figure 2.2. It simply tells the runtime to preserve all volatile argument registers before executing the method body[1]. Secondly it tells the compiler that the selector is passed to the method via the `%rax` register rather than a normal argument register.

2.2 Meta-regressions

The meta-level code shown so far is self-referencing. We send various messages in the basic support mechanism for sending messages. This leads to the meta-regression shown in figure 2.3.

To solve this problem, we differentiate between formal and actual binding time. This notion was introduced by Malenfant et al.:

> *"A **formal binding** time is the latest moment at which a binding can occur in general, while an **actual binding time** is the actual moment at which the binding occurs for a particular element of a particular program."[6]*

[1]This is similar to a *syscall* in UNIX, where another execution level is entered and the calling environment has to be preserved.

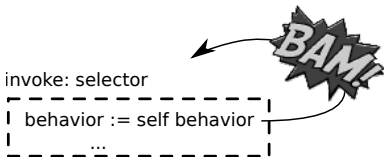

Figure 2.3: Self-referencing invoke implementation. Sending messages within `invoke` will result in an endless loop.

The semantics of Pinocchio are realised only with late-bound message sends. While in some cases we shift the actual binding of a message send to compile time, the formal binding stays at runtime. For example, our compiler inlines `ifTrue:ifFalse:` for `true` and `false` by performing local jumps. However, rather than implementing it as an if/else branch, there is a third case where the receiver is neither `true` nor `false`. In that case a message is sent to the actual receiver. This provides the illusion that the message is actually implemented as a message send, thus maintaining the formal binding time while improving performance through inlining.

2.3 Inline caches as static anchors

Our runtime respects the following invariants:

- Everything (except the object header) is formally resolved at runtime.
- Where this would end up in a infinite recursion we give the compiler a type hint. This grants that the default execution path is accessible without the recursion.
- The type hint mechanism does not affect formal binding time. If an unexpected type is encountered the execution will follow the alternative execution path provided by `invoke:`.

The type hints are referred to as *static anchors* and are just manually placed annotations for self-referencing meta-level methods. The annotations tie a local variable to a default type. This allows the compiler to look up the message at compile time. The annotations are visible inside the meta code and therefore explicitly document the default control flow. They are not external assumptions but specified from within the runtime. The Pinocchio MOP is therefore *anchored* by annotated default meta objects. Static anchors in the Pinocchio runtime are merely boundary conditions for recursions. They only provide an early bound native execution flow without ever having an influence on the formal binding time. They can be overloaded with custom code at any level of meta recursion. The static anchors preserve the semantics of message sends.

The actual implementation of static anchors is the same as the implementation of inline caches: When a method is looked up the first time the result of the

lookup will be stored for later executions. In fact the *invoke* instruction inside the caller code will be replaced by a direct call to the correct method found by the lookup part.

But when the caller-method is activated a second time there is always the possibility of a cache miss: The receiver of the message could have changed since the last execution. Therefore the type has to be verified on every call. In the case of Pinocchio every method has a preamble that does a type check[2].

Consider for example a method `length` that has been inlined for a String object. On the next execution of the same code the receiver is now an Symbol. The preamble of the `length` object will detect a type mismatch between the expected (String) and actual (Symbol) type of the receiver. Therefore instead of executing the method body it will jump to the `invoke:` function.

The only difference between *static anchors* and *inline caches* is that the former are *prefilled and immutable*. It is therefore possible to reuse inline caching mechanisms to resolve the self referencing problem. This provides us with a universal method of avoiding infinite loops. But since inline caching doesn't affect formal binding time we can avoid all problems that come with hard-coded static assumptions. Also since the static anchors are unified with inline caches we have a simple way of unifying user- and meta-level code.

[2]It is better to check the type on the callee side, since this avoids branch target mispredictions.

Chapter 3

The Compiler

The Pinocchio compiler is fully written in Smalltalk. It currently supports X86-64 as a target architecture. We have support for two binary formats: ELF and Mach-O. The compiler has a fairly traditional architecture: methods are parsed by a *parsing expression grammar*[2] into an abstract syntax tree (AST). We apply semantic analysis on variables while transforming the AST into three address code (TAC). After applying a *linear scan register allocator*[7] and some easy peephole optimizations, the assembler generates the X86-64 binary instructions. The full compile chain is shown in figure 3.1

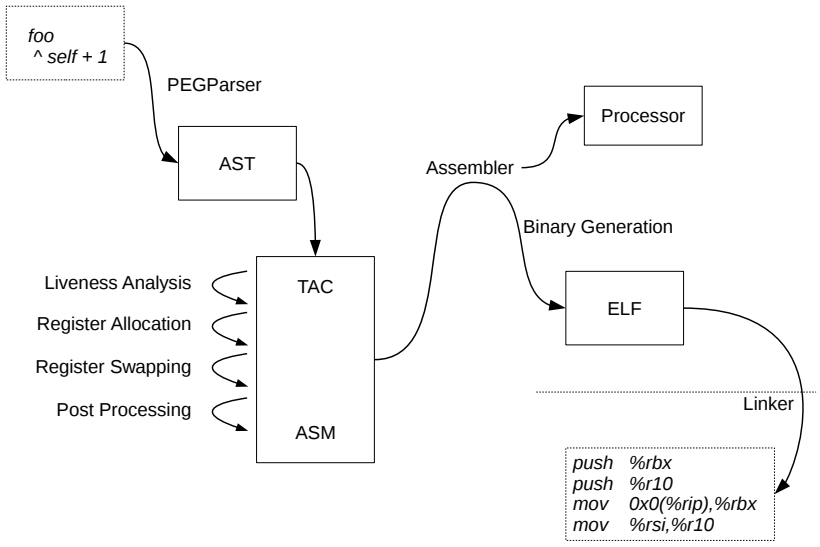

Figure 3.1: The full compiler chain implemented by Pinocchio.

Mapping Smalltalk code to assembler is straightforward for most of the semantics. We have variables and messages sends. Variables are mapped directly to registers or stack positions. Every object lives in a garbage-collected heap space and is addressed by a pointer. To assign an object to a variable we just move the pointer into the register in question.

Message sends relate to call instructions: They also have arguments but instead of a call target address they have a message selector. For an early experiment on how to map message sends to the C-stack in a direct threaded VM see appendix B. An assembler call instruction has the format `call name` where *name* corresponds to a label defined somewhere else in the code. The assembler will translate the label to a byte offset to the current instruction pointer ($ip) position. This instruction will cause the cpu to push the $ip to the stack (to be able to return from the call) and then add the offset thereby letting the $ip point to the called piece of code. The call target address is early-bound, therefore this cannot directly be mapped to a message send.

As described in the previous chapter, a message send in Pinocchio first pushes the arguments to the stack or, in accordance to System V ABI, loads them into the call-argument registers (thus being binary compatible to e.g. code compiled by a C compiler). The first argument is the receiver of the message. Thus in the preamble of every method we can fetch *self* from the first argument register. Secondly we store the name of the message, i.e. the selector symbol into the return value register ($rax). Most compilers use this register to pass the number of arguments to functions with a variable number of arguments. Since all Smalltalk messages have a fixed number of arguments we can use this register for our purposes. Finally we `call invoke` which is the meta-level function that will handle the late bound lookup of the correct method. Invoke will go through the class hierarchy to search for a method with the selector name. If found, it jumps directly to this method, thus staying in the same call-frame. This allows the invoked method to return directly to the caller.

3.1 Parser

Smalltalk code is parsed into an abstract syntax tree by a parsing expression grammar (PEG). The parser was reused from an existing, previous Pinocchio implementation.

3.2 Three Address Code

We generate a three address code (TAC) from the AST in multiple passes. In a first pass the AST is flattened out and all nested expressions are translated by subsequently storing the intermediate results in temporary variables. In this step mainly move and call instructions are produced. The resulting TAC is already quite close to the actual assembler code. Closures are replaced by an intermediate capture-closure pseudo-instruction. The closure code is compiled

and stored after the method code. Additionally volatile fences are inserted in all sections where the volatile registers could potentially be overwritten. They prevent the register allocator from assigning a volatile register to a variable whose liveness interval includes such a fence. This is necessary e.g. to prevent an external method from overriding a live variable.

The objects of the TAC can print themselves which provides a convenient way of inspecting them. For example the following code is from the dictionary implementation:

```
do: aBlock
 ↑ self bucketsDo: [ :bucket | bucket do: aBlock ]
```

This translates to TAC which is presented in the inspector as:

```
block 0:
  1: a P4TACBitTest
  2: jumpCarry basicBlock-lookup
  3: cmp arg1[-2], res                          #%rdi[-2], %
   rax
  4: jumpNotEquals basicBlock-lookup

block 1:

block 2:
  5: mov arg1, self                             #%rdi, %n/a
  6: mov arg2, aBlock                           #%rsi, %n/a
  --volatile fence--

block 3:
  8: [a P4TACClosure]
  --volatile fence--
  10: mov self, arg1                            #n/a, %rdi
  11: mov closureTmp0, arg2                     #n/a, %rsi
  12: mov a P4TACMethodObject[1], res           #n/a, %rax
  13: [%rax] := call invoke
  14: mov res, res                              #%rax, %rax
  15: return
```

where `block 0` is the typecheck, `block 1` will backup the non-volatile registers, and `block 2` loads the arguments into locals.

3.3 Compilation Phases

The second pass does a conservative liveness analysis. For every variable the first assignment and the last usage is stored. This represents the *liveness interval* of the variable. There is one special case when inlining the block of a *whileTrue:*-message. Since this block will be evaluated several times all variables which are live in the block need to have their liveness interval extended to the end of the block. This represents the only non-trivial case and it only occurs since we

flatten out the block for performance reasons. After this pass we can inspect `closureTmp0` from the listing above to see:

```
name:         #closureTmp0
register:     nil
firstAssign:  10
lastAssign:   10
lastUse:      13
...
```

The following pass will determine which variables have to be stored as remotes: We search for variables which are accessed both from within a block that could escape the current context, and in the method after the block is captured. Those we will store externally in a *remote array* so we can share the variable between the block and the method. The implementation is described in section 4.1.

In a subsequent step we will replace the intermediate closure-capture pseudo-instruction. We need to create a closure object and store the necessary objects in there. The new expanded instructions will copy into the closure a pointer to the closure code (calculated with the *load effective address* (lea) instruction), a copy of all local values accessed in the closure and the remote array for all variables who can be changed inside the block. This step also introduces TAC to create a *closure return token* which is used for non-local returns. The details are described in section 4.2.

Line eight from the previous listing expands after this step to the following code[1]:

```
block 3:
  mov `5', arg1                                  #n/a, %rdi
  [%rax] := call closureNew
  mov res, closureTmp0                           #%rax, n/a
  lea &a P4TACClosure, intermediateLValue #n/a, %r10
  mov intermediateLValue, closureTmp0[0]  #%r10, n/a
  mov aBlock, closureTmp0[1]                     #n/a, n/a
  --volatile fence--
```

The fourth pass will assign registers to the variables. We use a *linear scan register allocator*. This algorithm can assign registers in one pass in linear time. While scanning the liveness intervals it will subsequently use up all registers. When the number of variables at a given time exceeds the number of registers it will spill the variable with the longest liveness interval to the stack. After this step the size of the *stack frame* is known and the code for storing/restoring the non-volatile registers can be pre-/post-pended to the method. The register allocator will produce TAC which contains too deeply nested indirect addressing. For example there is no valid assembler instruction for directly moving one memory location to another. A double memory-access – a memory access

[1] In the listing the number 5 is actually the number 2 as tagged integer since $5 == (2 << 1) + 1$

where the address is stored in a variable on the stack, which translates to an instruction like *sp[stackPos][i]* – is not a valid instruction either. Therefore a post-processing is needed to flatten out such instructions and store the intermediate values temporarily in an unused register.

The TAC reflects the allocation showing the registers in the right-hand column:

```
block 1:
  push %rbx
  push %rbp
  push %r12

block 2:
  4: mov arg1, self                              #%rdi, %rbx
  5: mov arg2, aBlock                            #%rsi, %rbp
  --volatile fence--

block 3:
  mov `5', arg1                                  #n/a, %rdi
  [%rax] := call closureNew
  mov res, closureTmp0                           #%rax, %r12
  lea &a P4TACClosure, intermediateLValue        #n/a,  %r10
  mov intermediateLValue, closureTmp0[0]         #%r10, %r12[0]
  mov aBlock, closureTmp0[1]                     #%rbp, %r12[1]
  --volatile fence--
  10: mov self, arg1                             #%rbx, %rdi
  11: mov closureTmp0, arg2                      #%r12, %rsi
  12: mov a P4TACMethodObject[1], res            #n/a,  %rax
  13: [%rax] := call invoke
  14: mov res, res                               #%rax, %rax
  15: return
```

The last pass will do simple peephole optimizations. Currently no elaborate optimizations are implemented. We do fix some simple things like removing instructions which just move values back and forth between registers. In the example presented before this step would remove the unnecessary instruction on line 14. The resulting TAC can then be passed to the assembler.

3.4 Assembler

The assembler consists of generic functionality like handling jump labels and architecture specific subclasses that construct the binary instructions. It has a scriptable interface and will directly write the final instructions to a binary stream. Some addresses cannot be resolved directly: Either they correspond to a target of a forward jump in which case they are stored in a dangling references table. As soon as the assembler encounters a label all dangling references to it are resolved. Or the address relates to an external object, like an early-bound call or a global object in which case a label is installed as reference for the

installer. In any case the space for the address is preallocated to simplify the calculations. The downside is that this will cause forward jump offsets to always use the maximal size of 4 bytes, even though they might fit in 2 or 1 byte.

3.5 Binaries

The final binary code is converted into a Method object. Together with all unresolved external references it is processed further by the Method- and ClassInstaller. The Installer could either load the classes directly into memory or serialize it to a binary file. Only the latter is currently implemented. The class installer will collect all methods into classes in memory and then serialize those. The actual writing out will be handled by a different backend depending on the required format. Currently there is an ELF and a Mach-O backend. During serialisation all unresolved references are passed to the backend which will normally translate them into relocatable symbols. Those symbols are part of the normal binary format and can be resolved by a normal linker like the *GNU linker*.

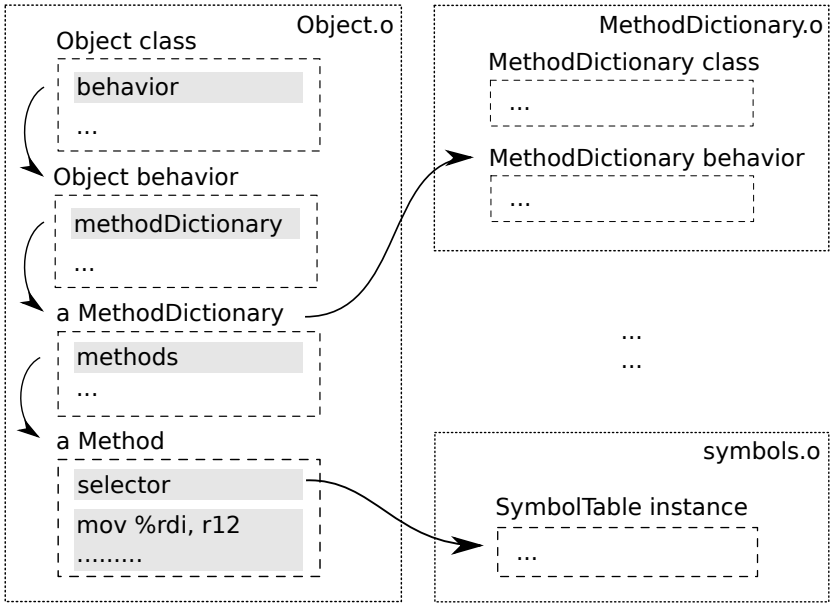

Figure 3.2: Pinocchio binaries layout. Arrows represent linked objects. Every object has a header that links to its class (not all links shown).

3.6 Writing ELF binaries

The *Executable and Linkable Format* (ELF) is an architecture and language agnostic binary format used for executables, object code, shared libraries and core dumps. An ELF file consists of a header and different sections that contain executable, non-executable data, relocation tables and symbols. Note that the symbols used in ELF are used like virtual addresses to instruct the linker on how to piece together the different classes. They are unrelated to the Smalltalk symbols used in Pinocchio. Those are exported by dumping the symboltable object into a relocatable object file.

The relocation table contains all imported and exported symbols of a certain object file. The imported symbols are the ones that need to be resolved by another file, the exported ones are those who can be addressed from within other files. An example of a fully linked Pinocchio binary with all relocatable symbols resolved is shown in figure 3.2.

Pinocchio writes all its binary data into the .pinocchio section. It maintains a relocation table in the .rela.pinocchio section. The relocation table connects unresolved addresses inside the .pinocchio section with relocatable symbols. Those symbols are listed in the .symtab and have certain properties, like a type, a scope and a name. The name is represented by an index into the stringtable (.strtab) which is a section that consists of a sequence of C-strings.

As an example we inspect the relocatable object file for Object, by running *readelf*. The header contains metadata like OS/ABI version and Type:

```
  Class:      ELF64
 OS/ABI:      UNIX - System V
  Type:       REL (Relocatable file)
  ...
```

The next part of the file contains the section table which lists all ELF sections in the file, their name and byte offset from the beginning. The .pinocchio section contains the dump of the Object class. One of its instance variables will be the methodDictionary which will also be written to this section, and therefore also all the methods within it. In an *objdump* the class looks like this:

```
0000000000000010 <Kernel_Object_Object>:
 10:  58 00 00 00 00 00 00     00 00 00 00 00 00 00 00 00
 20:  01 00 00 00 00 00 00     ...
```

The object starts at a byte offset of 0x10[2] (16 bit or two bytes) because it is prepended by a two pointer wide header. The second pointer of the object pointing to zero is an unresolved address. The relocations will instruct the linker on how to fill in the addresses.

[2]Written in hex notation.

The methods will look like:

```
0000000000000890 <Kernel_Object_Object__ifTrue:ifFalse:>:
  890:    0f ba e7 00             bt     $0x0,%edi
  894:    72 0c                   jb     8a2
  896:    48 3b 47 f0             cmp    -0x10(%rdi),%rax
  89a:    0f 85 46 00 00 00       jne    8e6
  8a0:    eb 0c                   jmp    8ae
```

The reason *objdump* can print the method names is that they are exported as function symbols in the relocation table. Those symbols are not directly used by Pinocchio, but they ease debugging since they are added to the final binary and all the standard tools for ELF can therefore recognize Pinocchio methods. The relocation section contains entries of the form:

```
Offset    Type            Symbol Name
48        R_X86_64_32S    MethodDictionary
```

This line states that in the .pinocchio section at the offset of 0x48 bytes a 32 bit pointer should refer to the MethodDictionary symbol. This is the header of the method dictionary object inside Object referring to the class. Note that the name is printed here for convenience but actually it just refers to a symbol and the symbol then refers to a string in the stringtable which will be the name. The symbol itself can be found in the .symtab:

```
Value   Size  Type     Bind    Ndx  Name
00000      0 NOTYPE   GLOBAL UND   MethodDictionary
```

It is a global and undefined notype symbol which means it will be imported from a different object file. The exporting file will provide type information. For example the Object file exports the following symbol:

```
Value  Size  Type    Bind    Ndx         Name
00010     0 OBJECT  GLOBAL  .pinocchio  Kernel_Object_Object
```

Which states that in the .pinocchio section at the offset 0x10 the globally exported ELF object named Kernel_Object_Object can be found. This symbol will let other parts of the system refer to Object, just like the method dictionary refers to MethodDictionary.

The object files generated by Pinocchio can then be linked with the GNU linker, which will resolve all symbols and combine the object files into a single ELF binary. In this binary file all the addresses that were still unresolved in the relocatable format are filled in.

Chapter 4

Implementation Details

This chapter will focus on selected implementation details of Pinocchio. We will present technical solutions and implementation techniques used to face the challenges in compiling Pinocchio.

4.1 Remote Variables

As mentioned in the previous section, there are situations where a local variable is shared between a method and a closure in a way that could lead to inconsistencies if we would just copy the value. This is illustrated in the following code, where a change to a local variable will influence the behavior of a closure:

```
animal := snake.
self delay: [
  animal bite: mouse ].
animal := elephant.
```

To simplify things we use a conservative approximation and state that a local variable which is used inside a closure and accessed or modified by the method after the closure is already defined potentially has to be synchronized between method and closure. In any other case it is safe to just copy the value into the closure which will be performing better since we keep the values to the stack.

All the variables that have this property and need to be synchronized between closures and the method are called *remote variables*. The name refers to the fact that they are stored externally in a heap-allocated array to make sure the values can be accessed by a closure after the stack frame has disappeared. This technique is also used in other implementations like the cog VM[1]. Every method with remote variables has such a remote array. Internally the compiler will create a new local variable called *remote array* and replace all remote vari-

[1] http://www.mirandabanda.org/cogblog/2008/07/22/closures-part-ii-the-bytecodes/

ables by virtual variables which are basically accesses into the remote array. The *remote array*-variable is the shared by the method and all its closures.

4.2 Closure return token

Smalltalk closures can contain return statements. Consider the following source snippet:

```
collection do: [ :i |
  i ifNotNil: [ ↑ i ]]
```

The example features a closure that will return the first item which is not nil. The closure belongs to a certain execution context — the method invocation that sends the *do:* message to the collection. But it will be executed in an other context — inside the invoked *do:* method. The return statement in the closure is a *non-local return*, which has another meaning than a normal return. It means "return from the context where the closure was defined" — as opposed to "return from the current context". A *non-local return* needs to be able to tear down multiple execution frames at once and return from the original invocation context. Another difficulty is visible in the following example:

```
self delayedExecute: [ ↑ self ]
↑ self
```

Assume the closure is stored for later execution but at the time it will be executed the original context is already gone, since the method returned. To prevent unwanted behavior the closure needs to be sure that the context to return from still exists and is available as expected.

In Pinocchio any method that features a closure that could potentially do a non-local return will create a unique *return-token*. This token provides a hand-shake between invocation context and closure to make sure the closure returns from the correct context. In the method preamble a piece of memory is allocated. Since the address of this memory location is unique it can be used as canary value to recognize a context frame. The method preamble will tag the current call-frame by pushing the token-address to the call-stack. If the canary is not present on the stack anymore, the returning closure knows that the context was already destroyed. Additionally the preamble will store the base pointer address at this memory location for the closure to know where to return to. When a closure is invoked the token is copied into the closure-object.

Additionally a method featuring a closure with a return statement will always backup all non-volatile registers to the stack. Otherwise in the case of a non-local return the registers which are stored in a later context would need to be collected from those frames on the stack. By just storing all registers we avoid this.

Once the closure decides to execute a non-local return the following things will happen:

- The stack pointer is set to the content of the return token. Therefore the current context collapses onto the context where the closure was activated.
- The return token address is compared to the first element on the stack. If they are not the same the original context is already gone and an exception is triggered.
- If the token matches we can safely return from this context by restoring all non-volatile registers with the values stored on the stack and then executing a *ret* instruction.

4.3 Inline cache

As described earlier a method invocation is compiled to the following assembler code:

```
mov receiver, %rdi
mov argument, %rsi
mov $selector, %rax
call invoke
```

After the invoke-method has completed the lookup it will modify this piece of code to incorporate the inline cache:

```
mov receiver, %rdi
mov argument, %rsi
mov $expected_type, %rax
call cached_method
```

The inline cache consists of the address of the *cached method* which replaces the address of *invoke*. The literal where the selector was stored is replaced by the *expected type* which will be loaded into the *$rax* register instead.

Every method has a preamble that will compare the type of the receiver with the expected type to detect inline cache misses. Unlike similar systems[3], in Pinocchio the method preamble does not need to be modified at runtime since the caller will provide the expected type as argument to the callee. This has the advantage that multiple call sites can support objects from different classes from an inheritance tree without triggering an unnecessary cache miss.

As an example consider the method *size* on the *Array* class being inline cached. The call site will load *Array* in the *$rax* register. The receiver of a method is always stored in the first argument register. Therefore the *size*-method will start by checking the type of the first argument. The class pointer will be read out and compared to *$rax*. If the test fails it will have to bail out to *invoke*. The argument registers and the stack are already in the same state as if we had called invoke, the only thing missing being the selector. Therefore the method preamble will reload the selector of the message from the literals into the *$rax* register and then jump to the address of *invoke*.

4.4 Method preambles

Since the preamble has to be executed on every invocation we implemented it in as few instructions as possible. Since Pinocchio features tagged integers (represented in the runtime by the SmallInteger class) there are three different preambles that are used depending on the class hosting the method:

SmallInteger The receiver has to be a tagged integer. We do a *bit test* and bail out if the least significant bit is not set.

Superclass of SmallInteger We do a bit test to determine if the receiver is a tagged integer. If it is, the *$rax* must contain the SmallInteger class otherwise we bail out. If it isn't a tagged integer we do a compare between *$rax* and the class and bail out if they don't match.

Otherwise We bail out if the least significant bit is set (because the method can't be installed on SmallInteger) or *$rax* does not match the class (cache miss).

This will generate for any method the smallest preamble possible.

4.5 Method activation

In figure 2.1 we have shown the implementation of the lookup ∘ apply. The actual method is applied by *perform:on:* which is a special message. The compiler will inline some low-level code that does a *jmp* instruction into the provided method. This is possible since it knows about the layout of a Method at compile time. This inlined block has a message sending fallback in case there is a type mismatch — the message being sent in that case is *perform: selector on: receiver with: arguments*. Arguments is an array wrapping all the arguments that were passed within the message. There is a small assembler template inserted (acting as glue code) that wraps the arguments into an array. The details of a custom method object being evaluated are shown in figure 4.1.

The protocol also works the other way round. We can send *perform: selector on: receiver with: arguments* to a native method. In this case the inlined glue code will instead unwrap the arguments and place them in the argument registers and the stack before jumping to the method.

This can be use for example to implement a method proxy that wraps around a native method by using the following implementation:

```
perform: selector on: receiver with: args
 ↑ nativeMethod
    perform: selector
    on: receiver
    with: args
```

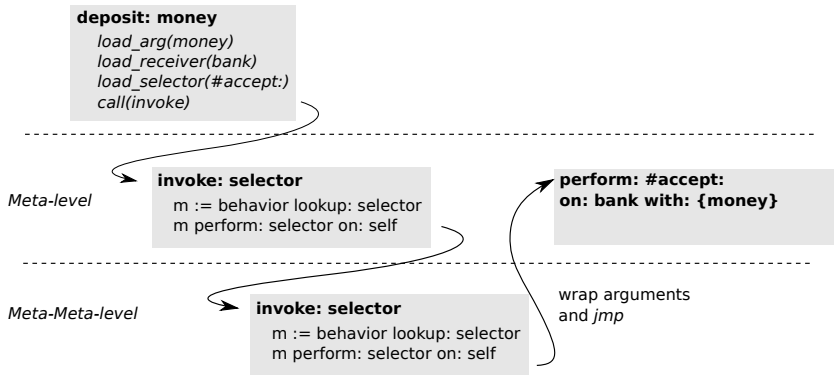

Figure 4.1: The lookup ∘ apply in the case of a custom method object. Invoke will send the message *perform:on:with* to the method instead of directly jumping to the method as in figure 2.2. Note that there are conceptually two meta-levels, but the meta-meta-level is in fact the same as the meta-level since it is self supporting.

4.6 Boehm GC workaround

In Pinocchio objects are referred to by pointers. If a method sends messages to an object or stores an object into a local variable all that is ever stored on the stack or inside the cpu registers are pointers to heap-allocated objects. The pointers point to the first field of the object. Every object also has a header with a pointer to its class, an identity hash, a size and some flags. The header itself lies in memory before the actual object and can therefore be accessed by indexing into the object with a negative index. The advantage is that the header can have a variable size and grow backwards. For example a variable sized object has a bigger header to store the size in it.

Since Pinocchio does not yet have its own garbage collection we use Boehm GC to allocate the memory for our objects. But since Boehm does not know about the headers lying before the actual objects and there are no pointers present in the static roots that would point to the actual beginning of the objects, the garbage collecting algorithm sometimes overwrites the headers when it compacts the memory. Since in future we will implement our own GC a simple workaround was implemented to avoid this problem. When creating a new object we now append a pointer to every object that points to the actual beginning of the memory location. This way Boehm can detect this pointer and is therefore prevented from freeing up the object headers.

4.7 Object Stream

The TAC instructions are stored in a *object stream* data structure. The stream is positionable since in subsequent passes instructions need to be inserted or

replaced. The most important usecases for TAC are:

1. Append an instruction
2. Expand an instruction to multiple instructions

The object stream holds the content in an OrderedCollection therefore the first use case is already implemented by the underlying structure. To make the second use case fast enough we used nested lists. If the stream is positioned to an existing object and new objects are streamed in then the old object will be replaced by a new substream and the new objects will be stored in it. Therefore replacing an object by multiple instructions does not cause the collection to grow nor does it move around any objects from the collection. An iterator can easily iterate over the nested lists and therefore provide a flat representation of the stream content. The datastructure combines all advantages of an array — like the compact representation and fast random access — with the low insert cost of a linked list.

4.8 (Re)Using existing tools

As already mentioned before complying to ELF standards allows us to use default tools like *objdump*, *readelf* or the gdb debugger. We can use the GNU linker to link our object files and even link them to C code. To bootstrap the system it was useful to be able to provide some functionality written in C. In the current system there is still some functionality provided in this way. For example since the size of a variable-sized object is stored in the header we can get it with the following code:

```
#define SIZE(object)  ((long)((tObject*)object)[-3])
#define TAG_INT(v)    (((v) << 1) + 1)

tSmallInteger size(tObject receiver) {
  return TAG_INT(SIZE(receiver));
}
```

As can be seen in this listing we have C structs that relate to certain types of Pinocchio objects. Those types exist so we have metadata available in a debugger session. They do not provide any functionality for the runtime but are rather modeled after the existing Pinocchio objects and ease debugging.

For example in a gdb session we can use them to print out objects:

```
(gdb) break Test_KernelTest__test1
  Breakpoint 2 at 0x611fe4
(gdb) run
  Breakpoint 2, 0x0000000000611fe4 in Test_KernelTest_{
 test1 (
    1: x/4i $pc
   => 0x611fe4 <Test_KernelTest__test1>: bt $0x0,%edi
(gdb) print print_object($rdi)
  printing object: 0x685f18
  size: 0
    header: (base: 2 var: 1 bytes: 0 mutable: 0 gcmark:
  0 hash: 0)
    is a: KernelTest
(gdb) backtrace
  #0  0x0000000000611fe4 in Test_KernelTest__test1
  #1  0x00000000006128d8 in Test_KernelTest__test0
  ...
```

The format of the metadata used at runtime to provide debugging information to gdb is called DWARF which is part of the ELF standard. This format is language-agnostic as well. In a future version of Pinocchio we can focus on generating more such metadata which will ease debugging and make tools like *objdump* and *gdb* even more useful. Then in a next step we could build our custom replacements for those tools providing more functionality and more convenient interfaces. This way we don't have to implement all debugging facilities at once and more importantly we maintain compatibility for debugging in a multi language environment. This approach has been demonstrated to work for Python by Kell [4].

Chapter 5

Evaluation

The current state of Pinocchio took around 4 months to implement for two people working full time on the project. It consists of 17K lines of Smalltalk code, 700 lines of C code (of which around 20% are only used for developing) and 50 lines of assembler code.

The Smalltalk code is distributed over the following parts:

Parser: 3.5K (inherited from previous Pinocchio projects)

Three Address Code generation: 4.5K

Assembler: 1.5K

Class building and linking: 0.5K

ELF/Mach-O Backed: 3.5K

Smalltalk runtime objects: 3K

Examples: 0.5K

This chapter will present some examples where the meta-circularity of Pinocchio can show its full strengths. We conclude the chapter by giving some simple performance metrics.

5.1 Implementing doesNotUnderstand

The protocol for method activation as described in section 4.5 is also used internally. It provides us with a flexible mechanism for handling methods and an interesting application of this protocol is implementing the *doesNotUnderstand:* protocol in a accessible and flexible way. Instead of making this a hard-wired functionality in the VM (like existing Smalltalk implementations handle it) we implement this meta-protocol as follows:

A behavior implementation can whenever the selector could not be resolved just return an instance of a DoesNotUnderstand object instead of a MethodObject. This object implements the perform message in the following way:

```
perform: aSelector on: receiver with: someArguments
  arguments := someArguments.
  selector := aSelector.
  ↑ receiver doesNotUnderstand: self
```

5.2 Behavioral reflection

By replacing a behavior object with another object implementing the *lookup* : protocol, the way messages are looked up can be completely altered. We show this by implementing a Prototype based lookup.

This change can be implemented in a very short time and completely alters the semantics of message sends. Those prototypes still have a class named Prototype but the new lookup doesn't look there anymore. We first instantiate a first prototype that holds a slot dictionary and install the *clone* method on it. We can then switch it to prototype style lookup with the following behavior:

```
lookup: selector for: object
  current := object.
  [current == nil] whileFalse: [
    (current slot: selector)
      ifNotNil: [ :method | ↑ method ].
    current := current parent. ].
```

What we have to consider (as discussed earlier) is that all messages we send to an object with this behavior are going to end up in this lookup method again. This example would not yet work with the above implementation because the messages *slot* : and *parent* would end up in this lookup again.

The problem could be addressed with typeHint annotations as shown before. But we can even resolve the problem without this feature. We simply let our PrototypeBehavior inherit from SmalltalkBehavior and delegate the lookup of those two selectors to the superclass. This means that those two methods would still be installed on *Prototype* whereas all user methods would be stored inside the slots dictionary of the addressed prototype. This example also shows how a user defined behavior modification can easily rely on the existing implementation.

The rest of the system still follows the same semantics but the new behavior can be plugged in an arbitrary class by sending:

```
aClass behavior: PrototypeBehavior new
```

5.3 Performance

To show that a metacircular runtime can compete with the performance of current dynamic language implementations we run three simple benchmarks. We do not provide an extensive performance validation. The benchmarks do not cover a wide range of features. They show the performance of some specific but commonly used features.

We compare Pinocchio with the Ruby interpreter version 1.9.2-p290, Python version 2.6.7 and the Croquet Closure Cog VM version 4.0-2489. Cog is a modern JIT VM for Smalltalk[1]. The benchmarks are:

Fibonacci(35) The time it takes to recursively calculate the 35th Fibonacci number. Since all messages sent can be inline cached this mainly shows how efficient the compiler can compile simple arithmetic, control structure and calls.

PDictionary(1M) The time it takes to store and retrieve 1 million entries in a Pinocchio dictionary. The Pinocchio dictionary was used to have a fair comparison.

NativeDictionary(1M) The time it takes to store and retrieve 1 million entries in a dictionary native to the language implementation. Note that the Ruby and Python hashes are written in C! This shows that it is possible to have metacircular core language features which still perform well enough.

	Pinocchio	Ruby	Python	Cog
Fibonacci(35)	0.22s	1.58s	4.23s	0.17s
PDictionary(1M)	0.96s			1.26s
NativeDictionary(1M)	0.96s	0.86s	0.22s	1.15s

[1] see `http://www.mirandabanda.org/cog/`

Chapter 6

Future Work

We were able to show in this thesis that it is in fact possible to create a VM-less meta-circular language implementation. The implementation is in a proof-of-concept status. We were able to demonstrate that the key features of our Smalltalk implementation are functional. For Pinocchio to be considered to be in alpha status targeting a production release still a lot of work would need to be done.

6.1 Missing key features

From the repertoire of key Smalltalk feature that are not yet implemented in Pinocchio the most important ones are:

Garbage collection For garbage collection we still rely on an external library — the Boehm GC. It is not yet fully clear how a meta-circular garbage collector could be implemented within Pinocchio. For one the garbage collector would need reserved memory space to be able to perform its work even when the available memory for other processes is exhausted. Secondly a GC implementation would need more knowledge about object internals than the other components we currently have in our system. Mainly update pointers, flush caches and update the headers.

Access to the context frame Existing Smalltalk implementations provide access to the current execution context through the pseudo variable *thisContext*. It provides the user with a reification of the call stack and also makes it possible to access its contents, like changing the return target address. In Pinocchio there is no object representing the current context frame. It exists only implicitly since its contents are fully mapped onto the execution stack and the hardware registers. This limitation however could be easily circumvented by providing the user with a mirror-object[1] which will enable him to access all the relevant data. Either by copying out all

values upon activation or even by compiling away the mirror and replacing it by direct accesses.

Exception handling Pinocchio does not yet implement the *ensure:* protocol. But since we already have a mechanism for non-local closure returns we have also shown how an exception handler implementation could look like. A method with exception handling blocks should store its stack pointer in an off-site exception stack. As soon as an exception is raised we need to walk the current exception stack and by some sort of hand-shake mechanism determine whether a specific method can handle the exception. Another possible solution worth exploring would be to write out a *stack unwind table* at compile time which would encode the information needed to unwind the stack for any line of code that can be executed. An example of such an implementation is provided by the DWARF standard: the *.debug* frame provides a debugger with all the information to trace back call frames and variable contents in case the user wants to step through the execution of a program. This frame was redefined in version 4.0 of the standard[1] and extended to form the *.eh_frame* section which enables the operating system to catch runtime exceptions and delegate them to the exception handlers registered to certain stack-frames. The data in this section is statically generated at compile time therefore it provides zero-cost exception handling — meaning there is no runtime overhead when no exceptions occur. This mechanism is used in newer versions of GCC to compile away the C++ exception-handling features.

Object Layout Pinocchio objects currently are just arrays of pointers. The layout itself is implicit, the compiler just indexing into the object pointer to compile instance variable access. Introducing first-class object layouts[10] would expose those implicit contracts and make them accessible and changeable.

6.2 Research directions

Pinocchio provides a very flexible and open environment that makes it possible to experiment with new language features and semantics. The fact that the Pinocchio core implementation is open and accessible enables a user to introduce new concepts not by emulating them but as first-class entities without having to extend a static VM. For example by implementing a custom *lookup*-method we could easily experiment with new access control mechanisms or fine grained encapsulation boundaries.

Additionally Pinocchio could also be used to address very low-level and hardware-near concerns. The compiler is accessible and can be extended to try out new compiler techniques and machine-level ideas. An example would be to prototype new compiler security features, like stack smashing protection.

[1] `http://dwarfstd.org/doc/DWARF4.pdf`

Then the Pinocchio environment itself still leaves research questions open. Future research directions include

Behavior vs. Class Pinocchio conceptually targets at separating the behavioral part and the class part of an object. Currently the class is the bearer of behavior — e.g. if an object should have a different lookup semantic the behavior object of its class needs to be changed. To provide a less class-centric but more agnostic approach we believe that the implementation should actually be the other way. Every object has a behavior object and this behavior object also has the means to retrieve the method-carrying entity — which would be the class for a class-based object and the object itself for a prototype-based object. This would enable a clear separation into two different entities between the meta-behavior (behavior-object) and the programmatic behavior (e.g. the class) of an object. Also the meta-behavior will define the format and behavior of the programmatic behavior.

Meta-circularity The core feature of the Pinocchio implementation — its meta-circularity — is a active research topic. It remains to be determined how far the proposed mechanism can be extended in Pinocchio. For example the implementation of the garbage collection will need more low-level access to objects and memory than any existing object. To offer this kind of direct interface we can rely on machine level instructions to compile to — but still it would mean an extension to the compiler. And it will result in early-bound assumptions given by the machine architecture since there will not be an abstraction layer but just memory access. Other missing features like access to *thisContext* will need similar compiler hooks. To implement all those low-level features in a simple and consistent way a restricted language-subset might need to be introduced — like e.g. the Klein [9] project does. Another possibility worth exploring would be to threat those use cases like a special case of JIT compilation. The low level features will be compiled within type checked sections just like any other type assumption a JIT compiler makes. This would eventually lead to a complete metacircular description of the system — e.g. with a GC that can store the objects it manages within a memory space object that is compiled away for the bottom level to direct memory access.

Quantitative evaluation The project so far showed that it is in fact possible to build a meta-circular VM-less runtime and that the implementation performs quite well under the tested conditions. It remains to show that the approach eases the experimentation with the language and the implementation of applications using the provided reflective flexibility.

Chapter 7

Conclusion

Pinocchio shows how the meta-level of a language can be unified with normal user-level code, and be compiled to machine code. Instead of having an artificial separation between meta- and base-level all objects can freely pass between the two. Pinocchio upholds the semantics of message sends to great extents, thus does not need the notion of an explicit change in meta-level. All features of a language, which are normally fixed assumptions inside a VM, are captured by an open and reconfigurable runtime library. Thus Pinocchio embraces an open feature design principle. For example by replacing a meta-object new behavior or new reflective capabilities can be introduced dynamically. By introducing a new meta-level even a new language can be supported. The shift in meta-levels happens implicitly, following only the normal meta-model of *class based inheritance* and *message-sends between objects*.

We were able to show how this approach enables a user to change language semantics to his needs. Furthermore we could also demonstrate that the performance impact of having metacircular implementations for core language features is not unreasonable.

During the development of Pinocchio we were able to validate many of the technical design decisions that were made by existing virtual machine implementations. And we were able to find a good balance between flexibility and uniformity vs. performance. Pinocchio runs faster than many current dynamic language implementations yet provides a more open, dynamic implementation of the language semantics and less early-bound assumptions. Those are strong indications that a very uniform language implementation can, through its simplicity, outperform an implementation with many specialized performance hacks.

Appendix A

Download and installation

To try out Pinocchio you can download the version described in this thesis from `http://scg.unibe.ch/download/pinocchio/p4.tar.bz2`. You'll need a 64bit operating system. Newer revisions are available at `https://github.com/pinocchio/p` and `http://www.squeaksource.com/P4/`.

You can run the Smalltalk image in the /pharo directory with the Cog VM. The Interface to the ClassInstaller to compile and write some classes to disk is:

```
P4StaticClassInstaller new
  processor: P4X86_64 new
  os: P4Linux new;
  compileClasses: {
    P4StaticKernelTest.
    P4KernelTest.
    P4ExamplesTest }
```

or for OSX

```
P4StaticClassInstaller new
  processor: P4X86_64 new
  os: P4OSX new;
  compileClasses: { ... }
```

After exporting the classes change into the src directory and execute *make*. This will compile all C sources and link the exported classes to one *pinocchio* binary. You can run the binary — by default it will just run all the test methods in the system.

By modifying *main.c* it is possible to select the entry point into the Pinocchio program, by choosing which message to send as the first one.

Appendix B

Early Prototype

An early prototype on how to map message sends to the C-stack using call instructions was still relying on a VM written in C. But the main principles were already present in this experiment. The VM was using direct threaded code which means that the different opcodes are implemented next to each other with jump labels and every instruction in the end will just jump to the instruction relating to the next opcode. This is implemented by first translating the opcodes into an array of addresses and then use those addresses as jump targets:

```
void ** pc = opcode_addresses;
goto **pc;

instuction_A:
    /* do something */
    goto **(pc += 1);

instuction_B:
    /* do something */
    goto **(pc += 1);

...
```

Mapping a message send to a function call is implemented by having all instruction in a designated method which is called recursively on message sends:

```
void interpret(Method method, void* arg[]) {
  void **locals = alloca(method->locals * sizeof(void*));

  ...

  call_method:
    target   = pc[0];
    num_args = pc[1];
    pc += 3;
    interpret( (Method)target,
               &locals[method->locals - num_args] );
    goto **pc

  return_local:
    origin = pc[0];
    return locals[origin];
}
```

As is visible in this listing the stack frame is allocated to a predefined method size. When invoking a method the end of the stack frame will contain the arguments to the called function.

List of Figures

Bibliography

[1] Gilad Bracha and David 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. doi: 10.1145/1028976.1029004. URL `http://bracha.org/mirrors.pdf`.

[2] Bryan 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. URL `http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf`.

[3] Urs Hölzle, Craig Chambers, and David 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. doi: 10.1007/BFb0057013.

[4] Stephen Kell and Conrad Irwin. Virtual machines should be invisible. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 6. ACM, 2011. URL `http://www.cs.iastate.edu/~design/vmil/2011/papers/p02-kell.pdf`.

[5] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0-262-11158-6.

[6] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection*, pages 1–20, 1996. URL `http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf`.

[7] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, sep 1999. ISSN 0164-0925. doi: 10.1145/330249.330250.

[8] David Röthlisberger, Marcus Denker, and Éric 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. URL `http://scg.unibe.ch/archive/papers/Roet07bUPBReflection.pdf`.

[9] David Ungar, Adam Spitz, and Alex 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.

[10] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 959–972, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048138. URL `http://scg.unibe.ch/archive/papers/Verw11bFlexibleObjectLayouts.pdf`.