Design and implementation of Bee Smalltalk Runtime

Javier Pimás
Disarmista S.R.L.
pocho@disarmista.com

Javier Burroni
Disarmista S.R.L
javier@disarmista.com

Gerardo Richarte
Disarmista S.R.L
gera@disarmista.com

Abstract

Bee is a Smalltalk dialect. Its runtime is exceptional in that it is completely written in Smalltalk. Bee includes a minimal kernel with on-demand loaded libraries, a JIT compiler, an FFI interface, an optimizing SSA-based compiler, a garbage collector, and native threading support among other things. Despite being written in Smalltalk, Bee achieves promising performance levels.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code Generation; Compilers; Incremental compilers; Memory management (garbage collection); Run-time environments

General Terms

Keywords runtime, virtual machine, self-hosting, compiler, garbage collector

1. Introduction

Bee runtime is completely written in Smalltalk. This covers kernel features like message dispatching, primitives, just-in-time compiling, threading support and garbage collection, among others.

The implementation of this environment in such a high level language required solving key problems.

Insufficient meta-object semantics. Smalltalk includes a very powerful metacircular class hierarchy. Yet it doesn't reify a key aspect of objects: it is not possible to access object headers. This poses barriers to the implementation of things like primitives, or garbage collectors. We slightly augment the smalltalk semantics by implementing *underprimitives*, extremely small and efficient methods that consist of just a few inline-assembled machine instructions.

Breaking self-sustaining circularity. Bee is self-hosted, which means it doesn't require any external runtime support library or virtual machine. Features like message lookup, primitives and garbage collection are implemented within the language itself. This characteristic means that the code implementing some features often assumes and even requires their existence to work. As an example, message lookup is written in Smalltalk, therefore message lookup naturally sends messages during its own execution. This leads to endless lookup recursion unless a means to cut it is incorporated. We worked around this kind of problems by carefully dissecting *code closures* and by issuing ahead-of-time nativization[1] of dispatch mechanisms.

A key aspect to support our solution is having control of the Smalltalk JIT compiler and machine code assembler, which are both written in Smalltalk. The JIT lets us transform underprimitives into very low level and efficient pieces of code, leveraging low-level actions in an object oriented fashion.

The remaining of this paper is organized as follows. In section 2 we describe the context of this Smalltalk dialect. In section 3 we describe the three most relevant models for the Bee runtime: the Bee metaclass hierarchy, the memory model and the ABI model. The runtime implementation details are described in section 4. We focus on the implementation of method lookup and primitives, but we also describe the modularity in the design, and the mechanism to perform low-level operations from Smalltalk, id est., the underprimitives. Current state of Bee development is described in section 5. An analysis of performance can be found in section 6. We culminate this work with a discussion on related work in section 7 and a conclusion with final remarks in section 8.

ESUG '14, August 18–22, 2014, Cambridge, CB, England.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] We use the term nativization to mean generation of machine instructions. This is to contrast with the word compilation, which we use to refer to bytecode generation; or the word jitting, which we use for just-in-time nativization

2. Context

High level languages usually allow a more dynamic programming style by delaying bytecode and machine code compilation, avoiding static typing and adding automatic garbage collection, among other things.

High level languages require *runtime support* to offer all these functionalities, usually in the form of a Virtual Machine. These functionalities have runtime costs that drag down performance of user programs. Besides, higher level languages discourage or even disallow low level actions like direct access to memory for the sake of program safety. Even if possible, accesses to memory are done through abstractions that try to validate each action, heavily hurting performance.

This combination of characteristics makes it difficult to implement runtime support itself in high-level languages, resorting instead to low-level ones.

Low level languages, on the other hand, usually require static compilation to machine instructions before execution, type specifications throughout the code and manual memory management. In exchange of this, low level languages usually generate highly efficient code.

But implementing runtime support libraries in high level environments can yield a better understanding of the problem's domain [20]. Runtime programmers can take advantage of the environment tools and abstractions. Instead of spending their focus simulating code execution in their heads, they can make use of the plethora of inspectors, browsers and debuggers to give shape to more readable and easier to understand solutions.

Finally, programmers want and should be able to know, understand and improve the implications and limitations of the runtime they are running on [14]. This is eased if the programming language of the runtime is the same than the language they use everyday for writing code.

3. Overview of Bee metaclass and memory model

Before delving into complex Bee topics like machine code generation, lookup and primitives implementation, we give an overview of a small group of Bee details that will help understanding the whole system.

3.1 Bee metaclass hierarchy

Bee follows as a base Smalltalk-80 class hierarchy [9], with some major deviations. The root class in the hierarchy is ProtoObject, whose super class is nil; Object subclassifies ProtoObject.

A big difference between Bee and Smalltalk-80 lays in its metamodel. The metaclass hierarchy, while similar, has been severed to allow dissociating class shape and object protocol. This simplifies the usage of objects of a same class with different behavior. The class Behavior truly refers to object behavior. It is not the superclass of Class and Metaclass. Instead, it is a variable size collection of method dictionaries. During lookup, the object's behavior will be traversed in order until a method is found for the searched selector. Thus, an object's behavior defines how the object responds to messages.

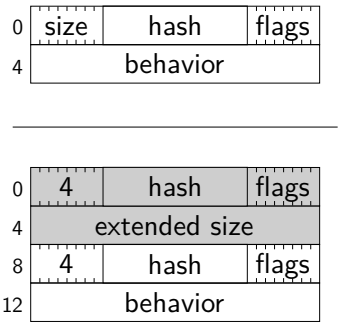


Figure 1. Regular and extended object headers

Both Class and Metaclass subclassify Species. Species includes most of Smalltalk-80 protocol for ClassDescription, which handles class and instance variable names, instance creation and more.

3.2 Memory model

In memory, objects are stored as an array of slots or bytes (depending on whether they are pointer o byte objects) preceded by a header. Slots that don't reference SmallIntegers have the address of the first slot or byte of the referenced objects, just after the header. We call these OOPs (Object Oriented pointers).

Given an object, its header describes many of its properties, like size, hash, shape and behavior. It is composed of various bit fields. It occupies 8 bytes in regular cases, or 16 if it is an extended header. Extended headers are needed for big objects (bigger than 255 elements) and for ephemerons [10]. The first doubleword of a regular header describes size, hash, shape and garbage collection status. The second doubleword is an OOP to the object's Behavior. When extended, the header contains two additional doublewords that are placed immediately before the regular header. The size field of regular header is set to 4 and the isExtended bit is turned on. The first doubleword of the extended header is set as a copy of the first doubleword of the regular header. The second word, on the other hand, is set to the actual size of the object. The size of the object represents the number of slots or bytes in memory of the object.

Notice that objects don't have a direct pointer to their class. Their class is determined by the class field found in the first method dictionary of their Behavior. Figure 1 shows the memory layout of object headers.

SmallIntegers are an exception to these memory layout rules, they are tagged. SmallIntegers are not allocated in the heap. When a slot is stored with a SmallInteger, instead of writing a memory address, we write the numeric value shifted one bit to the left and incremented by one. As objects are aligned in memory to 4 bytes addresses, SmallIntegers can be quickly distinguished from regular objects. This is a common technique that was already present in the 16-bit implementation of Smalltalk-78 [15], and adopted by many other virtual machine implementations later [6, 13].

We extend SmallInteger tagging with the use of *small pointers*. SmallIntegers represent numbers from -2^{30} to $2^{30}-1$, as they fit in a 32-bit word with the least significant bit clamped at 1. When dealing with pointers, we use standard SmallIntegers to do arithmetic calculations. This limits us to pointers with addresses in the 0 to $2^{30}-1$ range, exactly 1GB of memory.

Conversion of pointers to SmallIntegers is done by shifting the pointer to the left 1 bit and adding 1 to the result. But as pointers are always 4 byte aligned, we can convert them to SmallIntegers by just setting their least significant bit to 1, without shifting. The small integer represented by such a doubleword is the memory address divided by two. We call this a small pointer. Small pointers look and behave exactly as SmallIntegers, the programmer is responsible of dealing with conversions when needed. Thanks to small pointers, we are able to support up to 2GB of memory.

3.3 Bee ABI[2]

Bee assembler models a Stack architecture with a group of very specific registers, as described in [1]. These are: R (receiver and result), arg (argument), temp, self, method environment context, stack base and top of the stack. Both R, arg and temp can change between bytecode and bytecode, and are saved by the caller during message send. Self doesn't change from bytecode to bytecode but must be restored before returning from a method. Arguments are passed in the stack, pushed left to right, and are callee cleaned.

Currently, Bee only supports x86-32 bit architecture, and we map R to EAX, arg to EDX, temp to ECX, self to ESI, method environment context to EDI, frame pointer to EBP and top of the stack to ESP[3]. We show a stack frame for this ABI in figure 2. In the example, the method receives two arguments, contains two temporary variables and at least one block closure.

Most Smalltalk methods generate a new stack frame on activation, unless they are very short and don't need one. After pushing the previous frame pointer, they push the receiver and the compiled method. If necessary, they also push the method environment context. On exit, the stack top is set to the frame pointer, the old frame pointer is popped and ret n instruction transfers control back to the caller cleaning n arguments.

[2] The application binary interface defines low level conventions like parameter passing and saved registers across calls

[3] this convention is very similar to Pascal

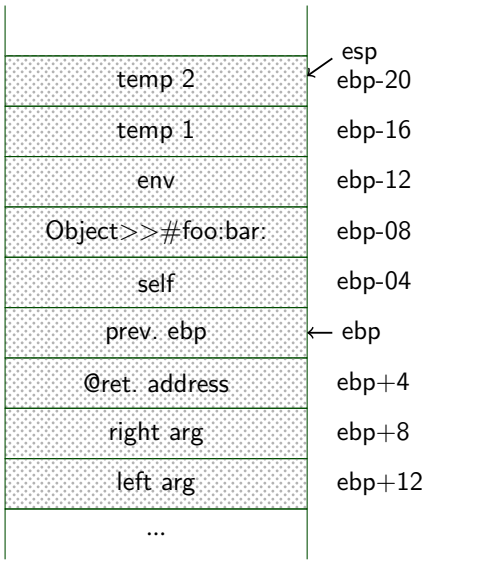


Figure 2. Bee Stack frame layout for Object>>#foo:bar: in memory on x86

4. Bee implementation details

4.1 Bee method nativizer

All methods in Bee are objects of the class CompiledMethod. Methods are compiled to bytecodes first, and then nativized. Among other things, CompiledMethods contain slots pointing to their bytecodes and to their NativeCode, a reification of machine code. NativeCode, in turn, contains a slot that points to the actual machine code, which is stored as a ByteArray. NativeCode also contains an array of references, with their respective offsets inside the machine code ByteArray. In this way, machine code is abstracted, can be easily accessed by the image when needed (i.e. during lookup), and needs little special treatment by the runtime, as shall be seen later.

To generate machine code from a method's bytecodes, an instance of the class BeeMethodNativizer iterates over the bytecodes writing their corresponding assembly.

As in Smalltalk-80, Bee Smalltalk contains special bytecodes for the most common arithmetic and logic operations. Our compiler is also smart enough to transform simple blocks, such as the ones used in #ifTrue:, #ifFalse: or #whileTrue: messages, into equivalent comparison and jump bytecodes. Besides, Bee is also capable of doing special case message send nativization for selectors that are not associated with a special bytecode. We detail both approaches next.

4.1.1 Inline nativization of messages through special bytecodes

The method nativizer generates specific assembly for most cases of special arithmetic and logic selectors. As example, let's consider addition. At compile time, when encountering a #+ message send, the compiler will output a Plus bytecode. At nativization time, the nativizer will assemble a couple of instructions to perform inline addition if possible, and to send the #+ if not. First it will insert a SmallInteger test for both the receiver and the argument. Then it will assemble the addition and a check if the result fits in a SmallInteger. Finally it will assemble the #+ message send. At runtime, if all checks pass, the #+ send will be skipped; if any of the mentioned checks fail, it will fallback to the message send. Even if this is mainly done for performance reasons, it has a deep impact in other parts of the system, easing the implementation of components like lookup and the garbage collector.

4.1.2 Custom nativization of message sends through send inliners

When the nativizer passes through a generic message send bytecode, it delegates the machine code generation to different assembly generators, or *send inliner*s. The method nativizer associates selectors with different send inliners. In the typical case, the associated send inliner will assemble the necessary instructions to call lookup. For some specific selectors, on the other hand, the nativizer will associate a different send inliner and generate different assembly.

Being in control of the bytecode nativizer from Smalltalk is critical for the implementation of these different send inliners, which are essential to leverage the development of Bee runtime in an efficient and object oriented manner. They are also key for system self-sustainability.

Different send inliners include the *assembly send inliner*, which generates special case machine code depending on the selector, and both *lookup send inliner* and *invoke send inliner* which generate machine code to call lookup and invoke respectively.

Underprimitives. These are a minimal set of selectors that are resolved with inline assembly, instead of sending a message. An example of an underprimitive is #_isSmallInteger. When seeing this selector, the assembly send inliner directly inserts assembly to check if the object is tagged.

```
assembleTestSmallInteger
  | integer |
  integer := assembler testAndJumpIfInteger.
  self loadObject: false.
  assembler uncontionalSkip: [
    assembler jumpDestinationFor: integer.
    self loadObject: true]
```

Underprimitives are a convenient abstraction of very low-level operations. They are limited to a maximum of two arguments. They assume the receiver is in R register and that the first and second arguments lay in arg and temp respectively, if present. A few dozen of underprimitives are enough to cover all the low-level actions needed for the implementation of the entire system.

4.1.3 Lookup and invoke

Execution of Smalltalk code involves execution of a message dispatching algorithm. In Bee, this algorithm is written in Smalltalk. For that reason, it is necessary to cut the recursive lookup chain to avoid an infinite recursion.

We shall distinguish two mechanisms when issuing what is generically called lookup: *method lookup* and *method invocation*. The first refers to the action of finding the corresponding compiled method to be later executed. The second one refers to the action of transferring control to the compiled method's native code.

In the next snippet we show the #_lookupAndInvoke entry method. The code is straightforward: #_lookup: fetches the corresponding compiled method for the selector, or nil if none was found, in which case the message to send is #doesNotUnderstand:. The compiled method is prepared for execution and finally control is transferred to the found method's native code. Notice that Bee uses *monomorphic inline caches* [8], so lookup includes call-site patching code.

```
_lookupAndInvoke: aSymbol
  | cm |
  cm := self _lookup: aSymbol.
  cm == nil ifTrue: [
    cm := self _lookup: #doesNotUnderstand:.
    self _transferControlTo:
          cm noClassCheckEntrypoint _asNative].
  cm prepareForExecution; patchClassCheckTo: self behavior.
  self
    _transferControlDiscardingLastArgAndPatchingTo:
          cm noClassCheckEntrypoint _asNative
```

Lookup. The native code generator used for lookup is the same than the one used for any other Smalltalk methods, with a slightly different configuration for message sends. When nativizing the #_lookup: message send, it will generate machine code according to the configured send inliner. If using the lookup send inliner, this code would fall into an infinite recursion. To solve this problem, we calculate a *code closure*. The implementation of #_lookup: is unique to all system, and we know beforehand the compiled method that would be found if #_lookup: were looked up. We can assure, by carefully writing lookup code, that the same happens to all the messages involved in lookup. Then, when nativizing lookup methods, we can set the send inliner to an *invoke send inliner*. An invoke send inliner pushes the unique compiled method for that selector, instead of pushing a generic

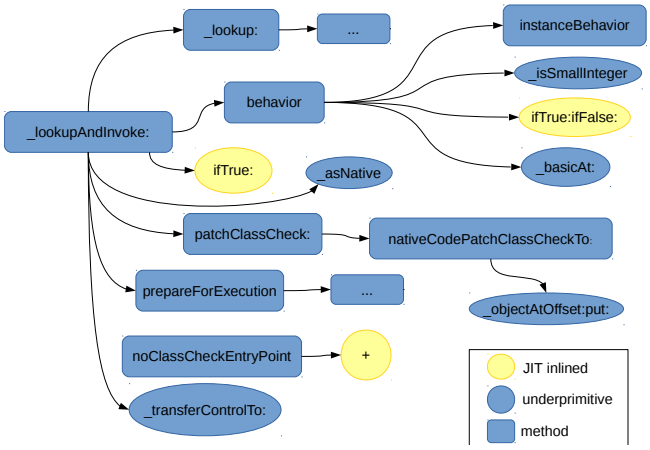


Figure 3. Code closure of #lookupAndInvoke:. Some selectors were summarized for brevity.

selector, and calls an invoke mechanism, instead of lookup. Figure 3 shows the message send graph of #lookupAndInvoke:, for which the method closure is calculated. We manually configure the method nativizer send inliners, so that message sends get nativized as invokes to the methods to be found for the respective selectors.

Invoke. We detail invoke implementation next. It is very similar to the end of #lookupAndInvoke:. The main difference is that invoke patches the call site to point to the native code just after the prologue. This avoids a class check in the next call (which would fail if the next receiver is of a different class).

```
_invoke: aCompiledMethod
  aCompiledMethod prepareForExecution.
  self _transferControlDiscardingLastArgAndPatchingDirectTo:
    aCompiledMethod noClassCheckEntrypoint _asNative
```

There is a last subtlety with invoke. Consider the nativization of #prepareForExecution message send. Of course, if it were nativized with a lookup send inliner, it would cause an infinite recursion at runtime. But if nativized with the very same invoke send inliner, it would also cause an infinite recursion of invokes. The trick to solve this is to nativize #_invoke: with a shortened and inlined version of itself. The following snippet shows the resulting code, where all message sends are actually nativized inline, and can be resolved without sending any real message.

```
_lightweightInvoke: aCompiledMethod
  | nativecode bytes classCheckDisplacement address |
  nativecode := aCompiledMethod _basicAt: 2.
  bytes := nativecode _basicAt: 1.
  classCheckDisplacement := 16rD.
  address := bytes _asSmallInteger +
    classCheckDisplacement.
  self
    _transferControlDiscardingLastArgAndPatchingDirectTo:
      address _asNative
```

To finish this section we explain why _lookupAndInvoke: can be a simple Smalltalk method with normal arguments, and how the control transferring works.

Arguments passing. As can be derived from section 3.3, just before the call to method lookup, all method arguments have been pushed in left-to-right order, and lastly, the selector was pushed and lays in the top of the stack. #_lookupAndInvoke: takes advantage of this fact. Inside #_lookupAndInvoke:, the references to the first and only argument will be transformed by the nativizer to the same address it would do in any case: ESP+8. This works perfectly until the return point.

Transefering control. At the epilogue of #_lookupAndInvoke: and #_invoke:, just after restoring frame pointer and stack top, the stack still has an extra argument, the selector or compiled method, respectively, that needs to be removed. If not, the method to be activated would wrongly see the selector as it rightmost argument. There is an extra complication, because after the selector was pushed, the call to lookup was issued, and the return address was pushed into the stack. _transferControlTo family of underprimitives write the same assembly that is commonly issued on method exit, but also solve this last argument problem by issuing pop [esp], an instruction that pops the top of the stack into its next position. Besides, as last instruction, it assembles a jmp instruction, instead of a ret n, seamlessly transferring control to the actual method, which doesn't need any special stack treatment.

4.2 Access to object headers

Thanks to underprimitives, it is possible to access raw object headers. Yet, these underprimitives are a bit too low-level for common usage. As an example, checking the size of an object requires reading the header bits to determine if it is extended, and then to access the corresponding size field, which might be a byte or a whole slot. Implementing all this with an underprimitive would be overkill, as it would require too much assembly writing. Instead of that, we implemented a set of methods that abstract access to object headers within Smalltalk code. For example, checking if an object is extended can be done with the following line of code:

```
_isExtended
  ^(self _basicFlags bitAnd: IsExtended) = IsExtended
```

Again, these are methods written in Smalltalk, so we can easily do complex actions. An object header can be marked as bytes with:

```
_beBytes
  self _flagsSet: IsBytes.
  self _isExtended ifTrue: [self _extendedFlagsSet: IsBytes]

_extendedFlagsSet: mask
  self _extendedFlags: (self _extendedFlags bitOr: mask)
```

The size of an object can be obtained from its header with:

```
_size
  | total |
  total := self _isExtended
    ifTrue: [self _extendedSize]
    ifFalse: [self _basicSize].
  ^(self _isBytes and: [self _isZeroTerminated])
    ifTrue: [total - 1]
    ifFalse: [total]
```

These methods abstract away most problems of dealing with object headers in a clean, object oriented style. A discussion about their efficiency is done in section 6.3.

4.3 Primitives

Bee doesn't implement primitives in the standard Smalltalk-80 way. The reason for this are undermethods, underprimitives and inline nativization of bytecodes. We begin with the description of the simplest primitives, and finish the section with the most complex ones, showing how most of the code can be implemented in plain Smalltalk, with the help of only a few underprimitives.

It is also important to remark that in current Bee iteration garbage collection has not yet been enabled. This eases implementation of primitives but will need revision when garbage collection is enabled again.

We start explaining this by showing a very small example. Consider the method ProtoObject>>#size. While in other Smalltalks this will need a primitive, in Bee it will be implemented as:

```
ProtoObject >> #size
  ^self _size
```

The implementation takes advantage of the reification of the object header, which can be accessed through undermethods. Other good examples are ProtoObject>>#behavior and ProtoObject>>#class

```
ProtoObject >> #behavior
  ^self _isSmallInteger
    ifTrue: [SmallInteger instanceBehavior]
    ifFalse: [self _basicAt: 0]
```

```
ProtoObject >> #class
  ^self behavior mainClass
```

Notice that Behavior has been reified, so finding the class can be delegated to it. ProtoObject>>#== shows the benefits of inline nativization of bytecodes:

```
ProtoObject>>#== other
  ^self == other
```

This will be nativized as a pointer comparison by the nativizer. If the pointers are equal it will load true, else it will load false. ProtoObject>>#perform: is a good example of the benefits of the reification of lookup.

```
ProtoObject>>#perform: aSymbol
  aSymbol arity = 0 ifFalse: [^self error: 'incorrect arity'].
  ^self lookupAndInvoke: aSymbol
```

Unlike Smalltalk-80 primitives, here there is no special concept of primitive failure. When a wrong arity is detected in normal Smalltalk code and doesn't require a second chance method activation. Careful readers will notice that the sent message is #lookupAndInvoke: and not #_lookupAndInvoke:. The difference is that the former doesn't patch the call site during invocation, which would be wrong in the case of perform. The main advantage of implementing low-level functionality in Smalltalk is that we can rely on existing code. For example, calculating selector arity was already implemented code. This gets an even bigger impact when writing more complex primitives. Consider the implementation of #value

```
BlockClosure>>#value
  self argumentCount = 0 ifFalse: [^self arityError].
  self _transferControlTo: self code
```

Notice how natural this code feels. code returns the address of the block's native code. The only addition to Smalltalk semantics needed was the #_transferControlTo: underprimitive. Other variations with a different amount of arguments are very similar.

The implementation of #become: is very interesting. Remember that #become: should scan all Smalltalk memory looking for references to the receiver, and replacing them with the argument. Besides, the process' stack, which is not inside a GCSpace, should also be visited.

```
ProtoObject>>#become: anotherObject
  Memory current make: self become: anotherObject

Memory>>#make: anObject become: anotherObject
  1 to: spaces size do: [:i | | space |
    space := spaces at: i.
    space make: anObject become: anotherObject].
  ProcessStack current make: anObject become:
    anotherObject
```

#become: is split in two stages. The first stage traverses each existing GCSpace, looking for references to the source object, and replacing them with the target one.

```
GCSpace>>#make: anObject become: anotherObject
  | objectBase object endOop |
  objectBase := self base.
  endOop := self nextFree.
  [objectBase < endOop] whileTrue: [
    object := (objectBase + 8 _asPointer) _asObject.
    object _isExtended
      ifTrue: [
        objectBase := (object _basicSize * 4)
                        _asPointer + objectBase.
        object := objectBase _asObject]
      ifFalse: [
        objectBase := objectBase + 8 _asPointer].
    objectBase := objectBase + object
                        _sizeInBytes _asPointer.
    0 to: object _pointersSize - 1 do: [:i |
      (object _basicAt: i) == anObject
        ifTrue: [object _basicAt: i put: anotherObject]]]
```

After all spaces have been scanned, the stack is traversed to find any remaining slot to change.

```
ProcessStack>>#make: anObject become: anotherObject
  | frame size endMarker nextFrame |
  frame := self _framePointer.
  endMarker := 0 _asObject.
  [
    nextFrame := frame _basicAt: 1.
    nextFrame == endMarker]
    whileFalse: [| first |
      size := nextFrame _asPointer -
                frame _asPointer // 4 _asPointer.
      first := 3.
      self
        make: anObject
        become: anotherObject
        in: frame
        count: size
        startingAt: first.
      frame := nextFrame]
```

GCSpace traversing has an extra subtle difficulty. The implementation avoids using real block closures. Real block closures require an environment, which is nothing more than an array to be allocated in the current GCSpace. This array might reference the source object and get modified during scan. If this happens, the source object might not be recognised any more. Besides #ifTrue:ifFalse: family of messages, both #whileTrue:, #whileFalse: and #on:do: are also inlined by the Smalltalk compiler. For example, when finding a #whileTrue: message send, the compiler inserts a jump-false bytecode at the block guard site, targeting the code after the argument block. It also inserts an unconditional back-jump at the end of the argument block, targeting the beginning of the guard block.

We close this section by showing Bee implementation of the most complex primitives, those related to blocks. We shall first give an overview of block mechanisms in Bee. We already showed the implementation of #value. Here we focus in the most difficult to implement piece, which is related to #ensure:. Consider the code

```
workSafe: aBlock
  [ aBlock value ] ensure: [ resource free ].
```

The meaning of this method is that after aBlock value is run, #free must also be run, no matter what happens in the block. To better understand the problem we can think how the stack looks like just after aBlock value, and how it will evolve. Somebody has called #workSafe: passing a block. To give a view of some of the different possibilities let's just assume it was:

```
sendWorkSafe
  ^self workSafe: [ a == b ifTrue: [^self] ]
```

In the stack we have #sendWorkSafe: frame, followed by #workSafe: frame. Next will be #ensure: frame. We can ignore what it does for now and assume that after a few extra frames aBlock frame will be placed in the top of the stack. The complete stack is shown in figure 4. Now, if a equals b, the ifTrue: branch will be executed, returning from #sendWorkSafe method. In stack frame terms, this means that stack should be unwound until #sendWorkSafe frame is found, and finally that frame should also be popped, returning control to #sendWorkSafe sender. But as there is an ensure in between, unwinding should pause when reaching the ensure stack frame, the ensured block should be executed and only after that unwinding should be continued. In the case that a was not equal to b, aBlock should finish its execution normally, and control should flow back to #ensure: normally, where it will activate the ensured block and return.

To allow the first case, #ensure: marks the stack to indicate an unwind stop point. In the case of premature return from a block, the stack will be traversed to find the returning

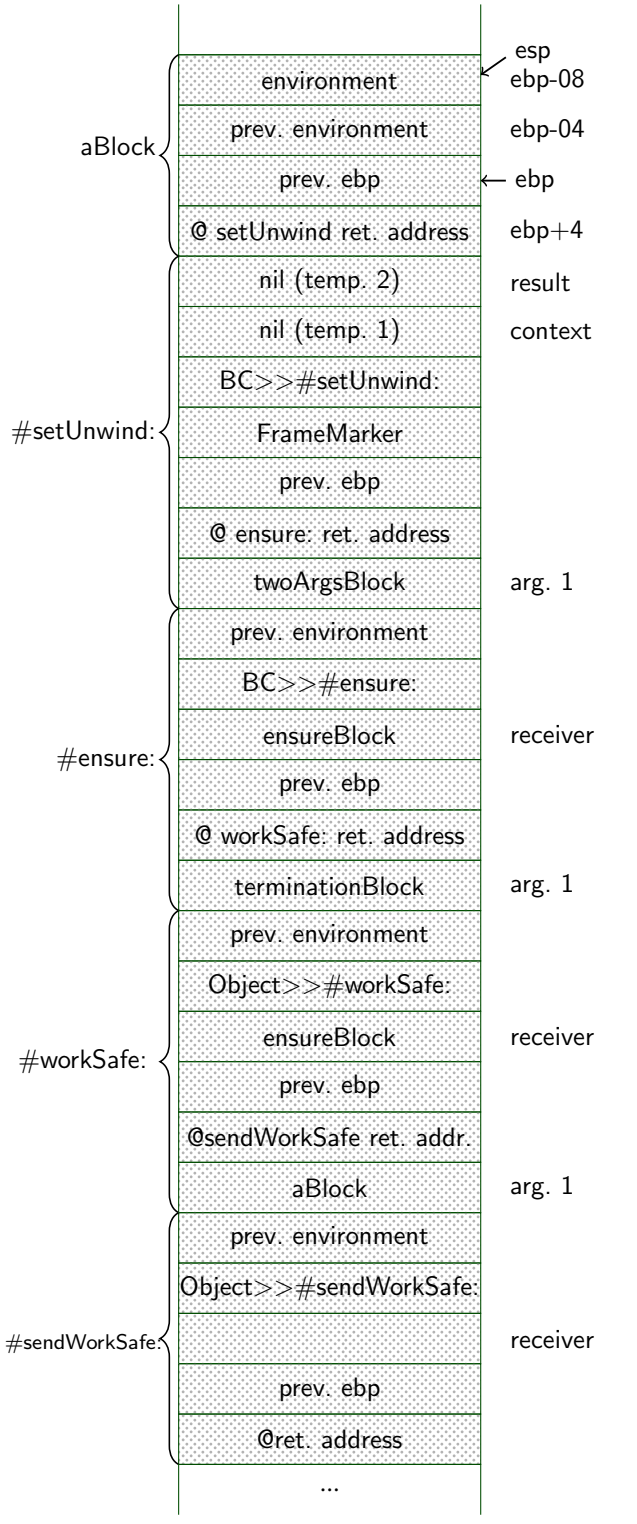


Figure 4. Stack frame layout after aBlock activation

method's frame. If a marker is found before that unwinding will stop. Here we show the code run when returning from a block.

```
BlockClosure>>#return: result
  | home environment saved frame |
  home := self methodEnvironment.
  home == nil ifTrue: [^self sendCantReturn].
  frame := BeeFrame current.
  [
    frame moveNext.
    frame isZero ifTrue: [^self sendCantReturn].
    frame hasBlock ifTrue: [
      saved := frame savedEnvironment.
      environment := frame environment].
    frame hasMarker
      ifTrue: [^self unwindUntil: frame context: saved
     returning: result].
    environment == home]
    whileFalse: [].
  saved == nil ifFalse: [saved _restore].
  ^frame current
    _dropUpperContextsReturning: result
    popping: self method argumentCount _asNative
```

a BeeFrame is a reusable stack frame reification. When initialized to current, it points to the top of the stack. It can be moved next to point to the next frame, and eases a lot the manipulation of stack frames. The #return: method, unwinds frame by frame looking for a marked frame or the returned method's frame. In the latter case it will drop all frames up to that point.[4] In the former, it will unwind just until the marker:

```
BlockClosure>>#unwindUntil: frame context: context
    returning: result
  frame firstTemporary: self; receiver: frame nextInFrame
    receiver.
  context == nil ifFalse: [context _restore].
  frame previous _dropUpperContextsReturning: result
    popping: 0 _asNative
```

The underprimitive used to drop stack frames is the same. This last method also does two modifications to the frame to be activated: changing its receiver and its first temporary. To explain why this is needed, we first show how the stack is marked:

[4] When walking the stack it will eventually find the method's frame, which has pushed an environment that is the same than the block's home

```
BlockClosure>>#setUnwind: twoArgumentBlock
  | context result |
  result := self valueMarked.
  context == nil ifFalse: [twoArgumentBlock value: context
    value: result].
  ^result

BlockClosure>>#valueMarked
  | receiver frame |
  self argumentCount = 0 ifFalse: [^self arityError].
  frame := BeeFrame current moveNext.
  receiver := frame receiver.
  frame receiver: FrameMarker.
  receiver _transferControlTo: self code
```

#setUnwind: method marks the stack. Notice that context, the first temporary is never directly assigned but checked for nil. Now remember the previous snippet of code, when unwinding to the marked stack frame. That code sets the first temporary of the frame, effectively making it not nil. Then, if context is not nil, it means the stack was unwound. If nil, there wasn't any non local return and the result is returned. The two argument block is then a block that is executed on marked stack unwinding. #ensure: uses it accordingly to guarantee that the ensured block is always executed.

```
BlockClosure>>#ensure: terminationBlock
  | result |
  result := self setUnwind: [:context :return |
    terminationBlock value.
    context return: return].
  terminationBlock value.
  ^result
```

Finally, #valueMarked code is similar to #value, but it takes the receiver and overwrites it with the marker. This is fine, as #valueMarked receiver can be restored from the previous stack frame (it is always sent by the same block).

4.4 Modularity

The strongest design principle behind Bee is minimality. Every aspect is split into smaller pieces as much as possible. Bee is divided into a very small kernel library, and a set of other libraries that can be loaded at runtime. Bee is self-hosted. This means it doesn't need to run on top of a Virtual Machine, all its runtime support is written in Smalltalk.

Bee libraries. Bee code is distributed through Smalltalk libraries, which are binary files that contain objects, including compiled methods and their native code.

Libraries are implemented as a heap of objects preceded by a description of the heap. To make loading fast, objects are stored almost as they will lay out in memory after loaded. The kernel includes a library loader, so it knows how to take the objects out of the library and how to plug them to the system. New classes are added to the Smalltalk global. New methods of already existing classes are inserted into their respective method dictionaries. All the needed actions are carried out to ensure that after loading the system stays consistent. Because of being binary based, library load time is small, compared to the time needed for compilation and nativization.

Bee kernel. Bee pushes Smalltalk modularity to new limits. Its kernel doesn't include a Smalltalk compiler, a nativizer, or a garbage collector. All of these functionalities are *optional*, and can be quickly loaded at runtime through libraries.

Bee is distributed as a native executable file. This file contains inside a kernel library with the main Smalltalk objects and code. The kernel library format is the same than the one used for any other library. The only difference is that it is packed inside a windows PE executable and that it contains no references to external objects. This kernel includes the minimal objects needed to be self-hosted. Main classes are placed in kernel, with their main methods. Methods contain their already nativized machine code. This is key for self-hosting. The entrypoint of the PE file is set to point to the machine code of a bootstrapping method. When execution starts, this method performs a basic initialization and then looks at the command line arguments to know what to execute next.

In Bee libraries, methods can be stored with or without their native code. Bee compiler and nativizer are placed in separate libraries, not included in the kernel and loaded on demand. Methods of libraries that include native code can be directly executed without loading the nativizer library. Libraries that don't include native code require loading the nativizer. When the nativizer is plugged, attempts to execute methods that don't contain native code automatically trigger their nativization. Of course, the methods of the nativizer library must be stored with their native code, as the native code of the nativizer is required to nativize methods. If both compiler and nativizer libraries are loaded, Bee will be able to execute arbitrary strings of Smalltalk code. This kind of modularity gives place to interesting possibilities. It is possible to create minimal system that is dynamic and yet doesn't include a compiler nor a nativizer within itself. To allow dynamism, the system could allow remote injection of compiled methods with their native code into the system from the outside world. This may prove useful for hardware platforms were resources are scarce.

5. Current Bee development

Bee is implemented on top of another *host* Smalltalk. This strategy lets us do development within a full blown environment. Many pieces of the system can be developed and tested in this environment. For example, Bee nativizer can be configured to generate machine code compatible with the

host environment. This allows testing most, if not all the nativizer functionality within the host.

In cases where testing within the host is not possible, we still can write the code inside the environment, and generate an executable file containing the kernel image and a library with the tests. Testing is conducted from the host. From it we spawn a Bee process, specifying the name of the test library as a command line argument. Test libraries are constructed to return value of 0 when the tests fail, or 1 if they succeed. Dynamism is transcendent, as changes done in the host environment can usually be tested immediately. Some other changes require the regeneration of the test libraries, which happens in just a few seconds. Only from time to time a change requires writing the kernel bootstrap image, because the system is split in libraries. Even in that case the time required is small, a few dozen of seconds.

As of June 2014, we are not yet able to directly debug Bee when running on itself. When this is needed, we resort to native code debuggers and disassemblers. For typical Smalltalk code, this will be solved after we plug the host's Smalltalk debugger and inspectors. Yet, a Smalltalk-written native code debugger would also be helpful to debug low-level code in a high-level environment.

In the previous iteration of Bee, a handful of garbage collectors were implemented. This includes full space mark and compact, and generational garbage collection. Yet, in the current form of Bee, we haven't finished plugging these collectors to the system. Therefore, there is no garbage collection available at all, until we adapt the old collectors.

6. Performance

Bee has been written with functionality and code quality as main priorities. Even though we haven't focused in performance yet, we still did implement some optimizations to obtain good enough performance for development. The philosophy has been to design the system with no inherent inefficient features, but to leave optimizations for later stages. The flexibility of the system facilitates research in this area.

6.1 Lookup optimizations

Bee is not interpreted, but ahead and just in time nativized. Besides, it utilizes monomorphic inline caches and different send inliners to enable fast dispatch. Assembly send inliners allow fast access to object headers, through directly writing machine code. Invoke send inliners provide for message sends without lookup, which is needed for lookup. We have taken advantage of this and configured the method nativizer to always use invoke for a set of very frequently sent messages. Through careful profiling and benchmarking we were able to remove the biggest performance bottlenecks.

The naïve #_lookupAndInvoke: method that was shown in section 4.1.3 was improved with a global lookup cache that speeds up lookup in the cases where mononomorphic inline cache fails. Currently we know that lookup is still a bottleneck, and we are working on the implementation of different optimizations to boost performance. When global cache fails standard lookup is done. Standard lookup is extremely slow, because it performs a linear scan in the method dictionaries of the object's behavior.

6.2 Optimizing compiler

The code generated by the JIT compiler is very efficient. However, to boost performance further, hot code paths could and should be made even more efficient. There is abundant research in this area that guarantees that important speed ups can be obtained. Adaptive optimization has been deeply studied, specially on Self [5, 11, 12, 19].

We have implemented an optimizing compiler. This compiler is run only for sets of methods that are known to be important performance-wise (for now, methods are selected manually). The optimizing compiler starts from an abstract syntax tree to construct an SSA-based call-flow graph of intermediate instructions [16, 18]. Through many stages it transforms this intermediate representation to finally emit native code. The different stages include speculative method inlining, peephole optimization, register allocation to finish in machine code emission. While still in early stages, this compiler has already provided a noticeable boost in performance.

6.3 Benchmarks

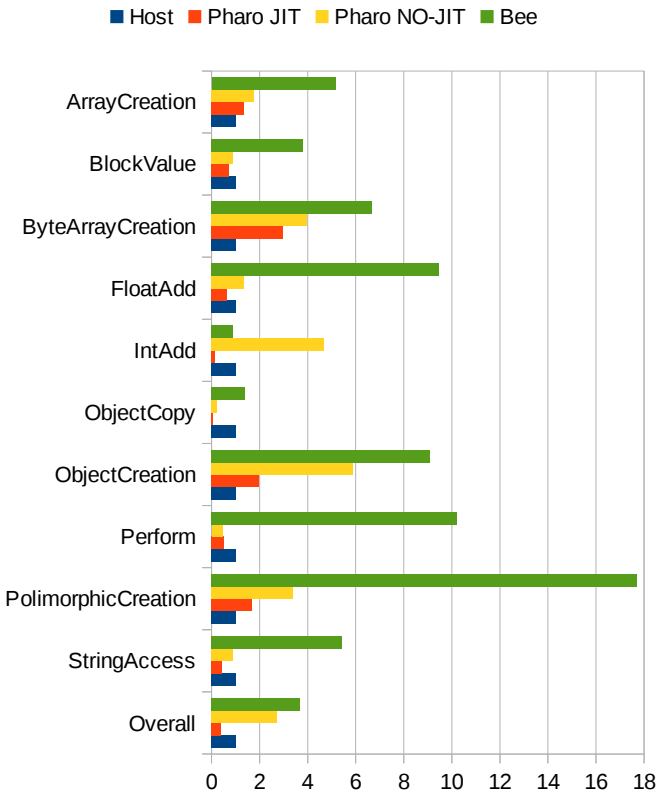
To measure performance we have run two different sets of benchmarks an compared the results against two other Smalltalk implementations: the host Smalltalk and Pharo [4]. In the case of Pharo, we have both run benchmarks with and without the JIT compiler.

The benchmark set is small but gives a view of the current Bee efficiency and also a preview of feasible performance levels that can be expected in Bee.

Slopstone is a well known Smalltalk benchmark that measures low-level operations as integer addition, block activation, object creation, and others. We split the results to give a better overview and also added some new sub-benchmarks to inquire about specific performance bottlenecks. Integer and float addition were tuned to run more iterations than in default Sloptone, because their execution time was so small that could not be correctly measured.

Some low-level performance details come to light in Figure 6.3. Results in this benchmark are highly diverse. On many cases Bee is between 3X and 6X slower than the host environment, with some notable exceptions. On the bright side, inline jitting makes integer addition even faster than the host virtual machine. On the other hand there are some notorious bottlenecks present on float operations, perform, monomorphic object creation and polymorphic object creation[5]. This last case is extremely slow because the

[5] With this we refer to creating objects of different classes



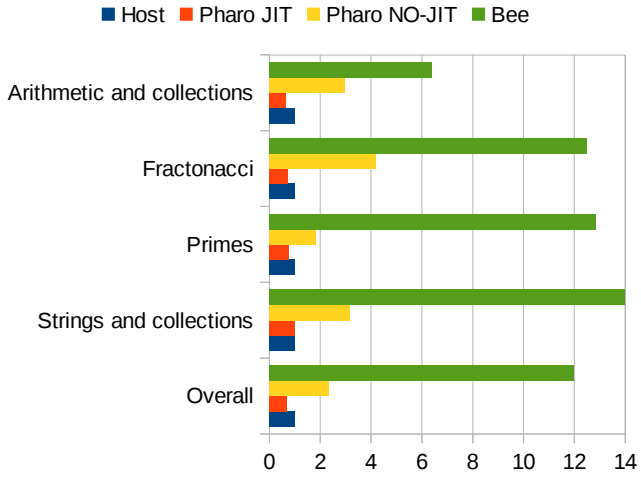
Benchmark	Pharo JIT	Pharo NO-JIT	Bee
ArrayCreation	1.36	1.77	5.16
BlockValue	0.71	0.89	3.8
ByteArrayCreation	2.98	3.98	6.67
FloatAdd	0.66	1.34	9.46
IntAdd	0.16	4.68	0.89
ObjectCopy	0.07	0.23	1.4
ObjectCreation	1.96	5.89	9.11
Perform	0.53	0.46	10.22
PolimorphicCreation	1.7	3.39	17.72
StringAccess	0.44	0.88	5.44
Overall	0.39	2.74	3.68

Figure 5. Normalized Slopstone execution times , relative to host virtual machine (lower is better).

monomorphic inline cache is not able to bear efficiently with polymorphic message sends.

Smopstone measures medium-level Smalltalk operations, which include recursive block and method calls, collection building and enumeration, streaming, and sorting. In a lower level, it performs arithmetic operations (mostly integer, with some fractions and floats), string manipulation, and streaming. As with Slopstone, we split the benchmark results to give a better overview.

In the case of medium-level operations we get an overall slowdown of around 12X, as shown in 6.3. This falls in line with the results of the previous benchmark if we take into account that the slowest results highly drag performance down.



Benchmark	Pharo JIT	Pharo NO-JIT	Bee
Overall	0.7	2.34	11.98
Strings and colls.	1	3.17	14
Primes	0.75	1.83	12.86
Fractonacci	0.73	4.17	12.5
Arithm. and colls.	0.67	2.97	6.41

Figure 6. Normalized Smopstone execution times , relative to host virtual machine (lower is better).

7. Related work

Squeak[13] is a self-hosted Smalltalk implementation. The code of the virtual machine is written in *slang*, a subset of Smalltalk. Slang code is automatically translated to C source and then compiled with a C compiler, allowing for very good performance. Yet, the code written in slang is not object oriented and more difficult to understand and modify than standard Smalltalk code. Programmers have to be familiar with C programming, compiling and debugging tools.

PyPy[17] is another example of a self-hosted virtual machine. PyPy consists of an interpreter and a translation framework. The interpreter code is written in RPython, a restricted subset of Python. Unlike with slang, PyPy's translator operates on RPython source through many stages of analysis and optimization. Different backends allow generation different outputs. The main backend writes C sources.

Jalapeño/Jikes RVM [2, 3] is a research project that implements a Java virtual machine in Java. Jikes implements different types of garbage collectors, supports multithreading and has different compilers that provide for adaptive optimization and highly efficient code. Access to object headers is done through *Magic*, a set of methods that are not implemented in Java but assembly, and allow direct access to memory and processor control.

Maxine JVM [21] is another Java virtual machine done in Java. While it shares many ideas with Jikes, Maxine distinguishes itself by its inspector, which lets the developer visualize and debug all the state of the virtual machine.

Tachyon [7] is a self-hosted Javascript virtual machine. Tachyon does not use a bytecode representation, it compiles directly to machine code. The compiler operates on different intermediate representations, applying different optimizations. To augment the semantics of the language, Javascript syntax is extended with type annotations and primitives that allow direct access to memory.

Klein is a metacircular virtual machine for Self written in Self[6, 20]. It enjoys a fully object-oriented design. Through the use of mirrors it achieves great code reuse and is able to access meta-object properties. Thanks to this, Klein can be remotely debugged from other PCs. Reactivity is highly appreciated and the environment provides many tools to create the illusion that the system is made of tangible, physical stuff.

8. Conclusions

Bee project was started with the implementation of a JIT compiler that recreated the host virtual machine's one but that was written in Smalltalk. The success in doing so brought the question of what other parts could also be directly implemented within the language. Access to the JIT compiler allowed the usage of underprimitives, which leveraged the implementation of the rest of the system. Today, Bee is far from finished, yet we know all required functionalities can be implemented. Furthermore, the resulting code is fully object oriented and can take advantage of all the benefits that a high-level environment brings.

The main remaining question to be answered is what is the maximum performance to expect from the system. We believe that the answer to that question will be highly positive, and that we will be able to unravel the mystery very soon.

9. Future work

Being such a big project, many ideas are still left to be explored. Garbage collection is ready to be plugged to the system, but requires some modifications to allow running in the self-hosted bootstrapped system.

Debugging of the self-hosted system is also not possible. Browsers, inspectors and debuggers are available while in the hosted system but not in the bootstrapped one. To make them work we have to implement a messaging system that wraps the one brought by the hosted environment.

We also plan to support out-of-process debugging and inspecting. This will allow us to run on resource-limited systems via remote debugging, even in places where graphical environments are not be supported.

Current implementation of Bee is more than 10x slower than the hosted environment, while only implementing small optimizations. Work on polymorphic inline caches, and the optimizing compiler will provide a big boost in performance.

Bee has initial support for native multithreading. While we have not deeply explored the subject, we believe this will provide bigger performance improvements and also ease the implementation and exploration of non-blocking and asynchronous message sending.

References

[1] P. C. Allen Wirfs-Brock. A smalltalk virtual machine architectural model. Technical report, Instantiations, Inc., 1999.

[2] B. Alpern, C. R. Attanasio, J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. E. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. ISSN 0018-8670. .

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005. ISSN 0018-8670. .

[4] 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. URL `http://pharobyexample.org`.

[5] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. . URL `http://doi.acm.org/10.1145/73141.74831`.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. . URL `http://doi.acm.org/10.1145/74877.74884`.

[7] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: An experience report. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4. . URL `http://doi.acm.org/10.1145/2047849.2047858`.

[8] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. . URL `http://doi.acm.org/10.1145/800017.800542`.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[10] B. Hayes. Ephemerons: A new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 176–183, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. . URL http://doi.acm.org/10.1145/263698.263733.

[11] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. . URL http://doi.acm.org/10.1145/178243.178478.

[12] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4): 355–400, July 1996. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/233561.233562.

[13] 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, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. . URL http://doi.acm.org/10.1145/263698.263754.

[14] S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0. . URL http://doi.acm.org/10.1145/2095050.2095099.

[15] G. Krasner. *Smalltalk-80 : bits of history, words of advice*. Addison-Wesley series in computer science. Reading, Mass. Addison-Wesley Pub. Co. cop.1983, 1983. ISBN 0-201-11669-3. URL http://opac.inria.fr/record=b1091689.

[16] F. Rastello. Ssa-based compiler design. 2015.

[17] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. . URL http://doi.acm.org/10.1145/1176617.1176753.

[18] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011. ISBN 012088478X.

[19] D. Ungar, R. Smith, C. Chambers, and U. Holzle. Object, message, and performance: how they coexist in self. *Computer*, 25(10):53–64, Oct 1992. ISSN 0018-9162. .

[20] 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. .

[21] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013. ISSN 1544-3566. . URL http://doi.acm.org/10.1145/2400682.2400689.