Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment

David Ungar
Sun Microsystems
4150 Network Circle, MPK16-158
Menlo Park, CA 94025
(650) 786-4905
david.ungar@sun.com

Adam Spitz
Sun Microsystems
4150 Network Circle, MPK16-158
Menlo Park, CA 94025
(416) 223-7248
adam.spitz@sun.com

Alex Ausch
Sun Microsystems
4150 Network Circle, MPK16-158
Menlo Park, CA 94025
(650) 786-2715
alex.ausch@sun.com

ABSTRACT

Can virtual machine developers benefit from religiously observing the principles more often embraced for exploratory programming? To find out, we are concurrently constructing two artifacts—a Self VM entirely in Self (the Klein VM), and a specialized development environment—with strict adherence to pure object-orientation, metacircularity, heavy code reuse, reactivity, and mirror-based reflection. Although neither artifact is yet complete, the environment supports many remote debugging and incremental update operations, and the exported virtual machine has successfully run the whole compiler.

As a result of our adherence to these principles, there have been both positive and negative consequences. We have been able to find and exploit many opportunities for parsimony. For example, the very same code creates objects in the bootstrap image, builds objects in the running VM, and implements a remote debugger. On the other hand, we have been forced to expend effort to optimize the performance of the environment. Overall, this approach trades off the performance of the environment against the architectural simplicity and ease of development of the resulting VM artifact. As computers continue to improve in performance, we believe that this approach will increase in value.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Debuggers; Incremental compilers; Memory management (garbage collection); Compilers; Interpreters; Run-time environments

General Terms: Design, Experimentation, Languages

Key Words

code reuse, debugger, exploratory programming, fix-and-continue, lenses, liveness, metacircularity, meta-recursive, mirror-based reflection, object oriented, prototypes, reactivity, remote reflection, virtual machine, Klein, Self.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '05, October 16-20, 2005, San Diego, California, USA.

© Sun Microsystems Inc.

ACM 1-59593-193-7/05/0010.

1. INTRODUCTION

The creation of a high-performance virtual machine (VM) for an object-oriented language is a daunting task. For example, the Self VM [16], comprised of 120,000 lines of C++ code, suffers from:

- *Replicated code:* An operation such as array access must be implemented once for the runtime, once for the simple compiler, and once for the optimizing compiler.
- *Debugging difficulty:* To inspect a Self object as a Self object, or to get the stack trace of an application that has crashed the VM, one must invoke a print routine in the VM being debugged. Running inside of a VM with dubious integrity, this operation becomes at best untrustworthy and at worst failure-prone.
- *Slow turnaround time*: A change to a C++ header file results in at least a five-minute wait for recompilation.
- *Complexity and brittleness:* The VM depends on the internals of the C++ runtime system. It must be able to parse both Self and C++ stack frames. If a new release of the C++ compiler changes the virtual function table (vtable) format, the Self VM breaks.

In the Klein project, we are attacking these problems by applying the very principles that have been espoused by OOPSLA papers for the past 19 years and exploited by applications programmers for even longer. To that end, we have been building a virtual machine and most of an environment for virtual machine construction that is:

- *object-oriented*. It uses a dynamic, pure-object-oriented language (Self) [17].
- *metacircular.* The VM and environment are written in the same language and share a great deal of code.
- *fostering code reuse* through language features, software architecture, and environmental support.
- *reactive*. The VM can be changed quickly, even while running. We aim to provide a compelling illusion of direct execution.
- exploiting *mirror-based reflection* for debugging the virtual machine (and for reusing code).

These principles are more difficult to apply to VM development than to application development, because many of them rely on

facilities that are implemented by the VM itself. Still, none are new, even in the field of VM development; others have built metacircular VMs for the Java™[1] programming language, which is a less dynamic language than Self. And others have built metacircular VMs for Smalltalk and Lisp. However, as we explain in Section 9, we are unaware of any efforts that aim quite so high in all of the above respects. Thus, the contribution of this paper consists of a report on the outcome (at least, so far) of the adherence to these principles for a VM development project.

As of this writing (July 2005), the efforts described in this paper represent a work-in-progress: not all of the debugging functionality is in place, and we are far from completing the Klein VM. We have, however, been able to compile the Klein compiler, and run the compiled compiler in a standalone Klein "VM." More detailed status is given below in Section 8.

In organizing this paper, we have faced the task of linearizing a two-dimensional matrix of principles and structures into a one-dimensional narrative. Thus, after giving some background, the paper delves into the principles to which we aspire, then explores each subsystem, calling out the connections to said principles, explains what is actually working at this time, compares this system to others, and finally concludes.

2. OVERVIEW OF THE KLEIN PROJECT

The Klein project seeks to simplify the construction and architecture of object-oriented virtual machines. Although we hope that the architectural lessons we learn will apply to a wide range of platforms from motes to supercomputers, we are targeting laptop-sized computers, so that we can do our research wherever we may happen to be. Additionally, we have chosen to both implement in, and target, the Self programming language [17], for both personal and technical reasons. We believe that Self's simplicity (fourteen bytecodes), purity (no Java-esque primitive data types[2]), dynamic object model (anything can be changed at runtime), and prototype-based model (it is somewhat easier to mimic class-based patterns with prototypes than vice-versa) make it a reasonable choice for this work. Our ultimate goal is to build a Klein VM for the Self language, with no C(++) code, that is as capable and efficient as the existing Self VM, and with only a third of the source code of the existing Self VM. Such a VM would also be as easy to debug and modify as a regular application written in Self or Smalltalk. Because the characteristics of the development environment shape the user experience and the artifacts produced, the creation of an appropriate environment for VM construction is part-and-parcel of our project.

The best lessons are fundamental ideas that can be applied in many different situations, and an understanding of the consequences that arise. Thus, the next section delves into the principles that Klein has embraced.

3. THE PRINCIPLES

The following notions, although familiar, are often nuanced differently from thinker to thinker. What follows is our slant. We owe a great philosophical debt to Smalltalk.

Object-Orientation: Klein follows the Self variety of object-orientation [17]. Everything is an object; there are no functions. Computation proceeds by sending a message to an object. The message names a task, but does not specify how the task is to be accomplished. Any object may refer to any other object in any of its slots (attributes). Even access and modification of state are accomplished by message-passing. Objects can inherit from other objects, and are free to override data slots (instance variables) with method slots. An object is self-describing; no class is needed. New objects are created by either cloning existing objects, or by using *mirrors* (see Section 5) to copy and add/remove/change slots of existing objects. Smalltalk-style *blocks* allow new control structures and extensions of old control structures to go hand-in-hand with new data structures and extensions of old data structures. Storage is reclaimed automatically. Programs are dynamically type-safe; the behavior of any program, correct or not, can be easily understood in terms of the source-level language semantics.

Metacircularity: The VM is written in the same language that it implements. The VM bootstrap image is statically compiled by the same compiler that is used to dynamically compile code in the VM at runtime. (See Section 6.1 below.) The interpreter (if present) is either generated by the same compiler, or constructed by a polymorphic specialization of the same compiler. The VM and the application use the same calling convention. Garbage collection (GC) uses the same mechanism to find roots in application stack frames as in VM stack frames. Whereas in a C(++)-based virtual machine, an object is described by another object (a class or map), which in turn is described by a different mechanism (hard-compiled C(++) structure declarations), in a metacircular VM, an object is described by another object (the class or map), which is described by another object (the map's map), etc. This infinite recursion must be terminated by choosing to "hard-wire" (or statically compile, or "configure") some facts about object layout, and those choices are part of our research agenda.

Reuse: Whenever faced with an architectural choice, we try to pick the path that would be most conducive to reuse. For example, we sacrifice the extra security of static type-checking in order to facilitate reuse.[3] We introduce *lenses* as a component of the architecture to allow our reflective code to work on different representations of Klein objects. (See sections 5 and 7.2 below.) The same reflection code builds the bootstrap image, performs remote reflection, and will also be used by the running VM itself to manage objects. Much of the Self programming environment is reused

---

[1] Sun, Sun Microsystems, the Sun logo, Java, JVM, and Java HotSpot are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

[2] For example, Java needs separate reflective operations for ints, floats, doubles, and longs, as well as separate "push" bytecodes for ints, floats, doubles, and longs.

[3] Some proponents of statically-typed languages maintain that static typing prevents only unintended or harmful reuse. For example, both the Gunfighter class and the Picture class may implement the "draw" message, but since they have such different meanings, one can not imagine (say the proponents) a call site in which one would want to send "draw" to an object that was either a Gunfighter or a Picture. To show how slippery the notion of "different meaning" is, we respond by asking about the "drawTwice" method. This is not to say that static typing does not have certain benefits, but rather that the Klein project has chosen to forego those advantages in its quest to maximize the potential for reuse.

to inspect, modify, and debug Klein objects. We plan to reuse the compiler as described above in the *metacircular* paragraph. Our assembler system reuses machine specifications to generate both an assembler and a disassembler. Perhaps the most surprising instance of reuse in Klein is our use of the Self transporter. This system was originally designed to allow us to save programs as source files, long before the Klein project was conceived. The transporter has been pressed into service to help build the Klein bootstrap image.

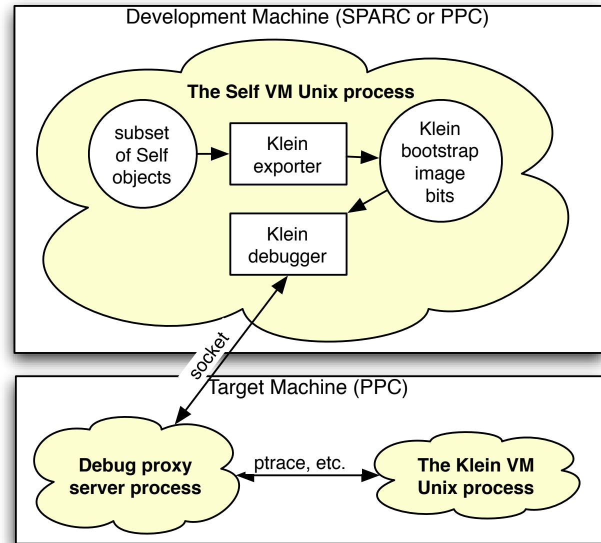
Reactiveness (a.k.a. liveliness): The Klein environment aims to create the precognitive illusion that the Klein VM is made of real, tangible, physical stuff. Ideally, anything that can be seen can be changed right where the user sees it, and the change happens instantly. In other words, (again, ideally) there are no visible compilations, no intermediate files, no includes or imports. In our system, when debugging at the assembler-level, the actual bits in memory are disassembled in real time, so the programmer can see and change what is really there. (See Section 7.4 below.) Comments in assembly-level code are recorded by embedding strings in the machine code itself. When debugging at the source-level, everything can be understood or *altered* (even in a running VM) in source-level terms. As the reader will see, we are still working to fulfill this promise.

Mirror-Based Reflection: As described in [4], Self separates out reflective operations, such as asking an object for the names of its attributes, into a separate interface, and uses special objects, *mirrors,* to implement these operations. For example, if object A needs to count the slots of B, it would first obtain a mirror on B from the reflection system, then ask the mirror for its size. This separation lets us build objects that appear to be Self mirrors from the outside while actually accessing and manipulating objects in either a Klein bootstrap image or a running Klein VM. This masquerade enables the reuse of most of the Self programming environment. (See Section 7 below).

4. COMPONENTS OF OUR VM DEVELOPMENT ENVIRONMENT

Refer to the figure below for an overview of the Klein architecture. Our system was designed to allow us to develop a VM for any platform, whether or not a Self VM is available for it. As it happens, we are currently developing for Mac OS X on the PowerPC, and running the development environment on both Macintoshes and Sun workstations.

We develop a Klein VM on two computers, one hosting the development environment (Self), and the other hosting a debug server (a C++ program) and the Klein VM being debugged. The two activities can also be colocated on one machine. All work on the development machine occurs in a single Self world. (A Self world is analogous to a Smalltalk image.) The exporter takes a subset of the objects in the Self world, turns those objects into bits, creates the meta-information (maps), and statically compiles the Self methods that comprise the bootstrap image. The Klein debugger leverages our mirror architecture to make the Klein VM tangible and manipulable. The debug server — containing the only C(++) code in Klein — responds to requests such as peek, poke, single-step, etc., by using whatever ptrace-like facilities are provided by the OS on the target machine.



The “subset of Self objects” contains:

- from the standard Self world, many collections and much of the reflection system, and
- from the Klein system, the Klein reflection system, object implementation, compiler, and assembler.

These are all the objects needed to run the Klein compiler.

Below, we describe how these Self objects are converted into the bootstrap image, which is a collection of bits in a byte-array containing objects, meta-objects (maps), and statically-compiled machine code. But first, it will serve the reader to understand something about the Klein object system.

5. THE KLEIN OBJECT SYSTEM: IMPLEMENTING AND REFLECTING OBJECTS

The Klein object system consists of a lot of Self code, and a few bits of assembly code (in the compiler and primitives). It defines the representation of Klein objects, tagged OOPs (object pointers), the object header, object vectors and byte vectors, constant and mutable slots, etc. This code builds Klein objects, implements mirrors (reflection), and helps the compiler build layout-dependent code.

Information about a Klein object, such as the names of its attributes, their offsets, and their contents (if the attributes are constant), is kept in another Klein object, called a *map*. Maps are canonicalized (shared by all members of a clone-family) and are immutable (copy-on-write). The Self VM also uses maps [7], but its maps are C++ objects, and thus defined by statically-compiled structure offsets and vtable entries. But in Klein, a map is just a Klein object[4]. Thus, to find out about an object, the VM must consult another object, and to find out about that, it might have to consult another object, etc. In this way, the Klein object system is *metacircular.* This uniformity simplifies everything, but creates an infinite space- and time- recursion. Where does it ever end?

[4] A map is a Klein object, but it is still a VM-level concept; an application programmer should never need to access a map or even be aware of its existence.

Two devices, one elegant, the other expedient, terminate the recursion. As in the Self VM, the immutability of maps implies that any two objects with the same layout can share the same map. And, since all maps have the same layout, all maps share the same map, terminating the spatial recursion. In addition, we have hard-wired (i.e. statically bound) some facts about map layout into the object system. These hard-wired facts are fairly sparse. In particular, Klein does not hard-wire function names to indices, as Java or C++ systems do with vtables. Consequently, virtual functions may be added or removed more easily. Klein does require that a map be an object vector, that the first element of the vector be a string giving the map's type (e.g. `byteVectorMap`, `objectVectorMap`, `smallIntegerMap`, etc.), and that the subsequent elements describe each slot (attribute) in the original object, with a particular organization. We regard the current choices for what is hard-wired as provisional, and worthy of future research.

The architecture of the Klein object system fosters *reuse;* indeed this system is used for at least three distinct purposes: by the exporter to build the bootstrap image, by the debugger to reflect upon a remotely running Klein VM, and by an application running inside a Klein VM to introspect upon local objects. The VM object holds both a memory interface object and a lens object. Basic operations are forwarded to lenses, which may interpret an object reference as either a number indexing a byte-array (via the memory interface), a number indexing a remote process, or a reference to an object in the running heap. Thus, one only need choose a VM object to shift the locus of discourse from self-reflection, to a bootstrap image, or to remote reflection. Lenses are described in more detail in Section 7.2 below.

The ability to have a mirror on a Klein object that is residing in a local byte vector provides one more aid to debugging; if an object appears to be corrupted in the running Klein VM, the programmer can look at it in the bootstrap image without manually parsing the object.

6. THE EXPORTER: BUILDING THE BOOTSTRAP IMAGE

The Klein exporter uses the object system to build binary representations of Self objects, the compiler to statically compile Self methods needed in the bootstrap image, and the assembler to assemble the compiled code.[5] For each exported object, a map is created and then exported. As mentioned above, maps are canonicalized. That is, whenever a new map is created, it is compared (in a hash-table) to all of the extant maps, and if a match is found, the preexisting map is used instead. The exporter terminates when all of the objects to be exported, all of the machine-code-containing objects (*nmethods*) and all of the helper objects (maps, etc.) have been converted to bits.

As we built the exporter, we discovered an unexpected parallel with the Self *transporter*. The Self transporter was developed to allow a Self programmer to work in a sea of live objects, and then save his "program" (a collection of slots) to a source file [20]. It employs various annotations on objects and slots that control how information is saved. For example, each slot is annotated with the name of the file (the "module") into which the slot is to be saved. Also, each slot is annotated with information regarding what is to be saved for the contents of the slot. For example, the actual contents may be saved, an initialization expression may be saved, or the name of a "well-known" object may be saved as the target of the slot. The annotations needed to transport objects into a new Self world turned out (obviously, in retrospect) to be exactly what was needed to transport objects into a new Klein world.

6.1 The Compiler

The Klein compiler is not the focus of our current research. Still, it is part of the development environment as well as the Klein VM, and does demonstrate the application of all of our principles.

Because of Klein's metacircularity, the Klein compiler does two jobs that, in the traditional Self VM, are done by two different compilers. The Klein compiler is used to compile the Klein VM itself (which is done by the C++ compiler in the Self VM), and will also be used to do straightforward compilation of application methods at runtime (which is done by the simple compiler in the Self VM). If and when Klein does get an inlining compiler, Klein's metacircularity will allow the compiler to inline VM hot-spots right into application code.

The compiler's design has been influenced by our desire for *reactivity.* In order to optimize edit-compile-debug cycle time, we have forsaken some optimizations (including customization [6]), so that the compiler can run faster. And of course, source-level debugging support requires that the compiler furnish a machine-level to bytecode-level mapping of code addresses and variable locations at each bytecode boundary.

6.2 Primitive Operations in a Metacircular Compiler

Primitive operations pose a challenge for a metacircular system. In the (non-metacircular) Self VM, a primitive operation such as _Clone is simply written as a C++ subroutine, and the compiler generates a call to it whenever the _Clone operation shows up. Since the Self language proper is more abstract than many other languages (e.g. Java), the Self system uses far fewer bytecodes but many more primitives (about 300).[6] Since many of these primitives share common functionality (e.g. the array `at:` and `at:Put:` primitives both include bounds checking), it would be possible to factor them into low-level, unsafe operations. For example, a system might use a machine-independent, low-level intermediate representation (IR) coupled with automatic instruction selection. Ian Piumarta has convincingly advocated such a "C-level" IR or interface level within the compiler [15].

Not wishing to undertake the construction of such an intermediate representation, and desiring to keep the semantic level of the primitives high enough to maintain pointer- and dynamic-type-safety, we have adopted a multi-pronged attack:

- Hard-wired machine instructions for simple operations: For example, loading a value from a register + offset just generates a PPC `lwz` instruction.
- A "C-level" API in the code generator to factor the compilation of inlined primitives: For example, there is a "generate bounds

---

[5] The exporter resembles the "image writers" described in [13], except that they did not compile bytecodes to machine code.

[6] Self primitives also perform the functions that native libraries do for Java.

check" method in the code generator that is called by the methods that generate the array `at:` and `at:Put:` primitives.

- Inline generation of simple, frequently-used primitives such as the small-integer addition operation: Since these operations all have much in common, including ensuring the operands are in registers and handling primitive failure, we only hand-code the specific portion of the primitives, and use reflection to automatically generate the actual methods called by the compiler.
- Callouts to "primitives" written in the full Self language: For example, the string-canonicalization operation does not need to be implemented as a primitive in the Klein VM for any reason other than compatibility with the Self VM, because the Klein VM's string table is a regular Klein object (unlike the Self VM's string table, which is a C++ object).
- Callouts to stubs written in an extremely-low-level variant of the Self language: For example, the clone primitive is implemented as an inlined simple primitive whose core is a callout to the clone stub, which forwards the clone operation to the object's map. The map code bottoms out in a routine written in a low-level variant of Self in which no blocks can be used, since block cloning would cause an infinite recursion in our metacircular architecture. Instead of normal, block-intensive control structures such as `ifTrue:` the routine relies on low-level branching bytecodes, such as `__BranchIfTrue:To:`. To date, our only such stubs are for cloning, and inline-cache miss-handling.

Our current approach has been expedient, but lacks uniformity and elegance. We remain unclear on the overall tradeoffs between this and more elegant approaches, such as special IRs. This area seems ripe for systematic exploration of the design space.

6.3 Exporter Performance Problems

There is no such thing as a free lunch, or a free principle. Each has both a cost and a benefit. In the export system, the cost of our principles has been the performance of the environment. Because we value *reactivity*, we want the export cycle to be fast. We want the ability to make a change to Klein and try it out quickly to get feedback on whether it worked or not. However, our other principles have made it difficult to achieve the kind of performance we need. Because we want Klein to be *metacircular*, the compiler and exporter are written in Self, but the Self virtual machine is likely not as efficient as today's commercial Java™ Virtual Machines[7]. Because we want Klein to be *object-oriented*, we have written it in an extremely well-factored, malleable object-oriented style, with many small methods. High-frequency, dynamically-typed, virtual function calls, the use of real blocks for all control structures (even "if"s"), and ubiquitous accessors all hurt efficiency.[8]

At its slowest point, the Klein export cycle took an hour (mostly due to the naïveté of our initial implementation). We have taken three different approaches to tackling this problem:

- *Optimizing the full export cycle*. Through various algorithmic improvements, we have managed to get the export process for the full Klein VM down to four minutes.
- *Mini Image*. For testing certain kinds of features such as arithmetic and blocks that do not require the full functionality of the Klein VM, we can build a small, stripped-down bootstrap image by exporting a subset of the modules included in the full Klein VM. Building the Mini Image takes less than twenty seconds.
- *Incremental update*. We have extended our export system to perform incremental updates of incremental changes. Self's mirror-based *reflection* architecture was a great help here because there is only one Self primitive used for all reflective mutation [4]. Intercepting this primitive with a publish-and-subscribe pattern allows our export system to detect changes automatically. The amount of time needed for an incremental update varies depending on which objects need to be updated, but most incremental updates take less than one minute.

Overall, keeping Klein's export cycle fast has proven to be more of a challenge than we anticipated. The experience of working on Klein is definitely not as smooth and *reactive* as the experience of working on a regular application program in an interactive high-level language; there is a significant difference between receiving feedback almost immediately and receiving feedback after four minutes. Also, it is frustrating that we periodically need to take time away from implementing VM functionality and spend it on optimizing the export cycle instead. By comparison, rebuilding the Self VM after a C++ header change can easily take five minutes (but the Self VM is complete), so we may not be too far behind.

7. KLEIN REFLECTION SYSTEM

When a development environment creates the illusion that the source-level constructs of a program are real, physical entities, it reduces the cognitive load placed upon the user by offloading mental tasks from conscious to unconscious mental facilities. Accordingly, the Self system takes pains to foster the illusion of direct execution, and the Klein debugging environment ought to do the same. Unfortunately, when the program being debugged is the VM itself, there is a problem: The existing Self environment — written in Self — runs on a working VM and uses a graphical user interface (UI). When either incomplete or broken, the Klein VM can neither run nor tickle pixels; it must be debugged from the outside. Therefore, a separate, possibly remote, Self VM hosts an environment that manifests the innards of the Klein VM at the source level. Thanks to Klein's *metacircularity* and Self's mirror-based *reflection* model, Klein can *reuse* a vast amount of already-written Self programming environment code.

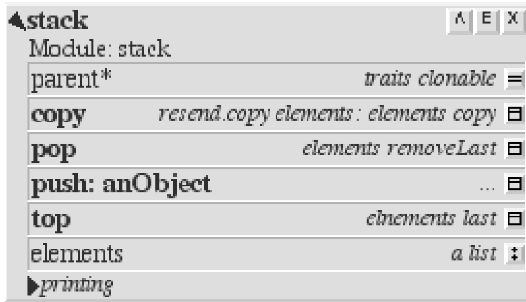
7.1 Reusing Self Outliners for Klein

In the existing Self environment, a program is constructed by directly manipulating an object, and an object is shown on the screen with an *outliner*. It displays the slots contained in that object, as well as other information about the object (as shown in the screenshot below).

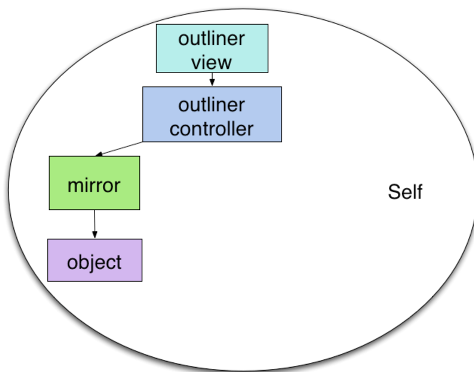
---

[7] The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

[8] However, previous work on the Self project has gone a long way towards making it possible for VMs to optimize code written in that kind of style (see [10], [7], [11], [9], [5], [3] and [6]).

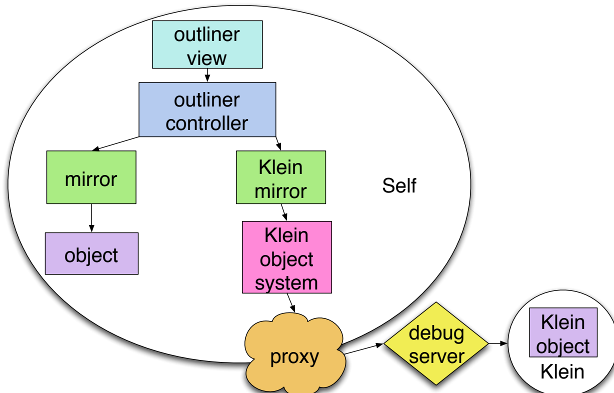


The outliner gets its information about the object by using a reflective object called a *mirror* (see figure below) [4].



A mirror for a particular object O answers questions such as, "What are the names of all the slots in O?" and, "What is contained in the 'foo' slot in O?" It also carries out requests to change an object, such as "Add a slot named 'foo' containing the method '3 + 4' to O." To create an outliner for O, Self first creates a mirror for O and then creates an outliner view and controller to display and manipulate the information provided by the mirror.

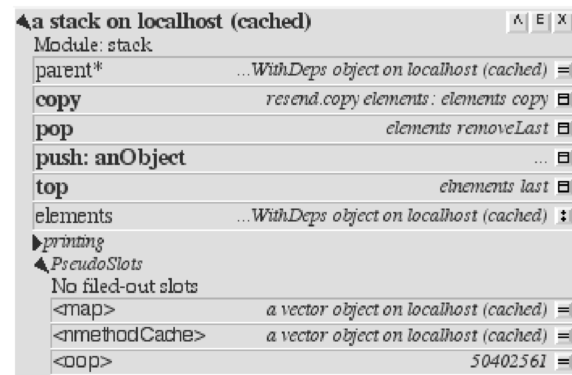
To create a Klein outliner, a *Klein mirror* substitutes for a Self mirror. A Klein mirror is an object that has the same interface as a regular mirror, but instead of reflecting a regular Self object, it reflects a remote Klein object. It does so by consulting the Klein object system to parse the object, and inspects either a bootstrap image in a local byte vector, or the raw memory of a remote Klein process via the debug server. All of the rest of the Self



outliner code, the whole view and controller (which is a significant amount of code), is *reused.*

The result is an outliner that shows an object in the remote Klein VM. As in an ordinary Self outliner, the object's slots are shown and can (someday) be manipulated in terms of objects and source code.

The Klein mirror (and thus the outliner) also provides for three "pseudo-slots" providing information about the map, native-method cache, and oop (object pointer) of the Klein object. Embedding this low-level information in the high-level outliner (as shown in the screenshot below) makes it easy to see the object from either a VM-level or an application-level perspective.



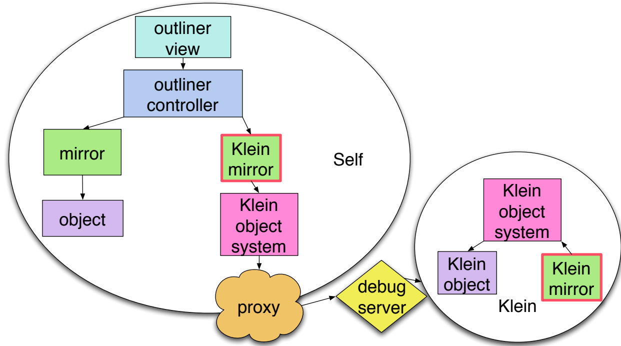
Because of Klein's *metacircularity,* the map, native-method cache, and oop are all regular Klein objects, and the user can click on any of the pseudo-slots to summon a Klein outliner on those objects.

7.2 More Reuse: Klein Reflection Code For Remote Debugging Also Works Inside a Running Klein VM

So far, we have explained how Klein mirrors support reflection for a remotely-running Klein VM or a local image in a byte vector. In either case, the object system manipulates integer addresses with the assumption that the objects hold still in memory while under observation. (The running Klein VM is halted during remote reflection.) However, Self's semantics and the design of the Klein object system collude to permit the same object system to work inside of a running Klein VM itself, even one with a copying garbage collector. In other words, the Klein mirror and object system code that was described above is the same Klein mirror code that functions as Klein's runtime reflective system.

To allow Klein's reflective code to function in these two different contexts, the architecture includes objects called *lenses,* because they allow reflective code to "focus" on either a remote Klein image or the local Klein image. Every low-level memory operation is double-dispatched through the lens of the currently-running process. There are two different kinds of lenses: the `localObjectLens`, which is used inside a running Klein process, and the `memoryLens`, which is used in a Self process to examine the memory of a foreign Klein process.

For example, our reflective code needs to be able to examine the tag of an oop (object pointer), in order to determine what type of



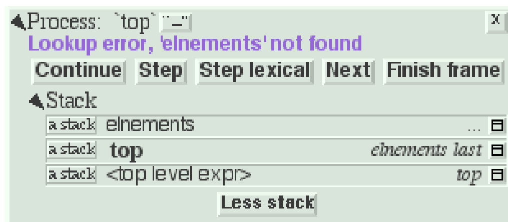
object that oop points to. The tag of a Klein oop is found in the two low-order bits of the oop. In one case, the `tagOf: x` operation is being executed by a Self process, and the argument `x` is a Self integer containing the oop. The operation is double-dispatched through the `memoryLens`, and the tag is found by performing the Self code `x && 3` (a bitwise-and operation to mask out all but the lowest-order two bits of the integer `x`). In the other case, the `tagOf: x` operation is being executed inside a running Klein VM, and the argument `x` is not a Self integer; it is a reference to the actual Klein object. The double dispatch goes through the `localObjectLens`, and the tag is found by a primitive operation that clears the leftmost 30 bits of the source register, shifts the result left by 2, and puts the final result in the destination register.

In this manner, the Klein reflective code can function both in a Self image, where it operates by passing around Self integers that represent the oops of the objects in the remote Klein image, and in a Klein image, where it operates by passing around references to the actual Klein objects.

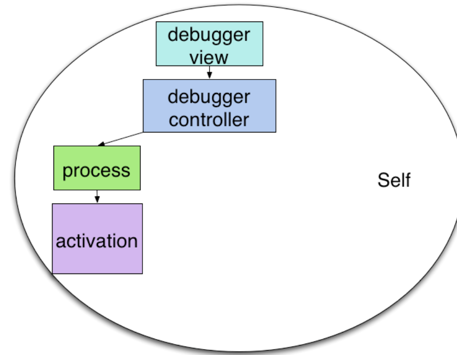
Although the Klein reflection code was designed for reuse, when we first implemented it we had no Klein VM and we could only use it to build and examine objects in the bootstrap image. Later, the reflection code was used to remotely debug the nascent VM. Finally, when the Klein VM had enough of the Self language implemented to support the reflection code, we tried exporting and running it inside a running Klein VM to reflect upon the live Klein objects. This code just worked, proving the usefulness of Klein's lens architecture and its *metacircularity*.

7.3 Remote Activations

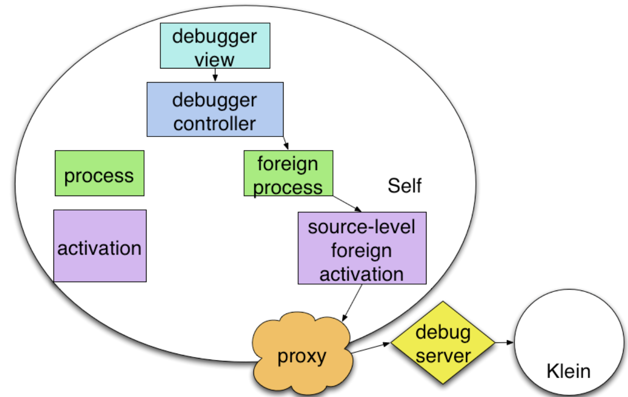
Just as a Self outliner portrays any Self object, a Self debugger (shown below) portrays a Self process.



Each process contains a stack of activations. The debugger view and debugger controller manifest a process and its activations in the looking glass of the computer screen.

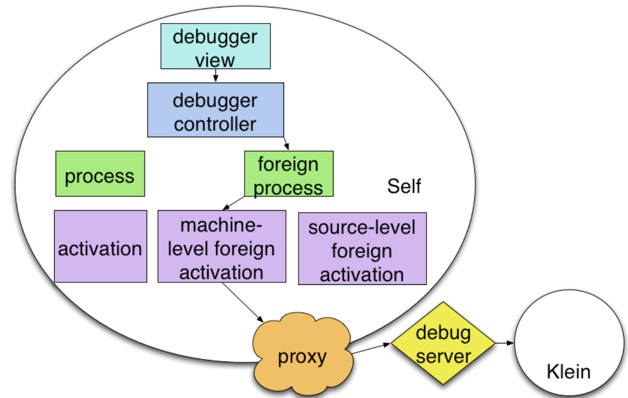


In the Klein debugger, *foreign process* and *foreign activation* objects substitute for the regular Self process and activation objects. This masquerade allows us to reuse the debugger view and controller code. Thus, the Klein VM process can be rendered as tangible as a Self application process. (Some features of the Self debugger depend on as-yet-unimplemented VM facilities and are not yet available in the Klein debugger.)

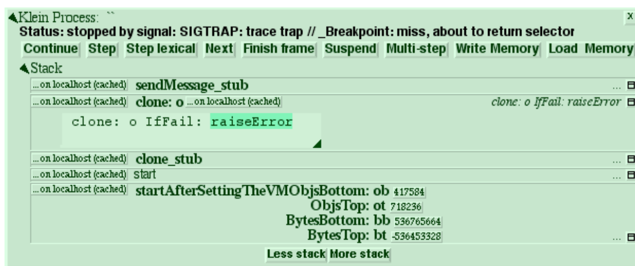


7.4 Machine-Level Remote Activations

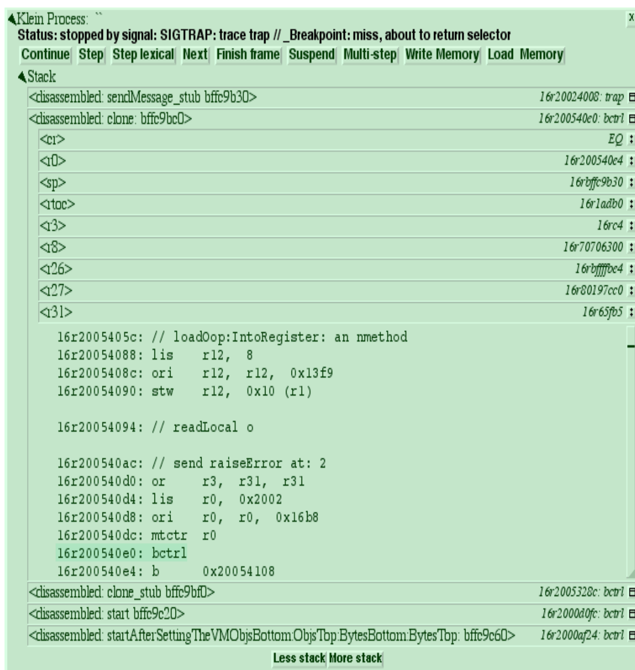
Working on Klein often requires the programmer to think about the compiled code from different viewpoints: either at the Self source code level, or at the machine code level. To make it easier to shift between these two different viewpoints, Klein includes two different kinds of foreign activation objects: *source-level foreign activations* and *machine-level foreign activations*.



This dichotomy of activation objects allows the debugger to operate at either the source or the machine level. In source-level mode, one sees the Self source for each activation in the running Klein process;



in machine-level mode, one sees the disassembled machine code for each activation (similar to the view that a traditional VM developer would see in a low-level debugger).



The assembly code displayed by the Klein machine-level debugger (including the comments[9]) is the code that is actually in the machine's memory; the machine-level foreign activation examines the contents of memory and disassembles it to produce what appears on the screen. So if memory were to become corrupted somehow, the screen would show the corruption. The machine-level debugger has other *reactive* features. For example, one can change the machine code in-place by typing new code right over the old; the modified method is reassembled and the change takes effect immediately.

8. CURRENT KLEIN FUNCTIONALITY

At present, although neither the development environment nor the virtual machine are complete, each does exhibit significant functionality.

8.1 Development Environment Functionality

As of July 2005, the export system can build a Klein bootstrap image, and incrementally update a Klein bootstrap image or even a running Klein process that is suspended in the debugger.

The Klein VM debugging environment has many, but not all, of the features of the Self environment. Remote object outliners can show Klein objects at the source level, but cannot yet modify Klein objects at runtime, nor evaluate arbitrary snippets of code in the context of a remote Klein VM. Remote debuggers show a running Klein process from either a source-level or a machine-level perspective, and do single-stepping and on-the-fly code modification at the machine level, and (somewhat crudely) at the source level.

8.2 Virtual Machine Functionality

Here is what works in the VM today:

- All Self language features except for dynamic inheritance. Dynamic inheritance allows an object's parent slot(s) to be assignable instead of constant. Experience with Self has cooled us somewhat to this feature, since such objects are sometimes difficult to debug, and the presence of an assignable parent slot interferes with most of Self's optimization tricks. Thus, dynamic inheritance is only used sparingly in the Self code base; so far we haven't needed to implement it in Klein.
- Almost 100 primitive operations that furnish basic functionality: 30- and 32-bit integer arithmetic, cloning, vectors, strings, system calls.
- Indirect (or reflective) message-sending (in Smalltalk terms, the "perform:" message, and all of its variants, including dynamic delegation).
- Reflection. This is probably the most interesting piece of current Klein functionality. As described above, Klein mirrors function both as part of the remote environment for inspecting Klein objects, and as Klein's runtime reflection system.
- The Klein compiler has compiled some simple methods, running inside Klein. (We will not be able to test the compiler on more complex methods until we have implemented garbage collection, because the Klein process runs out of memory when it tries to compile more complex methods.) This compilation uses all of the above functionality, including the reflection system. The compiler was written as a normal Self program, and thus uses many collections and other parts of the standard Self world, as well as the Klein assembler system.

The Klein VM now consists of about 155,000 objects and 12,000 compiled methods. Building it takes about four minutes on a 1.33 GHz PowerPC G4 Macintosh PowerBook.

There is still a lot of work to be done, to bring Klein to a comparable level of functionality with the Self VM. Here are some things that have not been implemented yet:

[9] The Klein assembler embeds comments directly in the machine code by inserting a branch, followed by a particular never-used instruction, followed by the bytes of the comment string. The Klein disassembler recognizes this pattern and displays it as a comment.

- A plethora of primitives, including graphics.
- Garbage collection.
- A more sophisticated compiler.
- Dynamic inheritance.

Although the project is not complete, we have made enough progress to see how our principles have shaped the development experience and the structure of the artifact.

9. PREVIOUS WORK

Metacircularity has a long and distinguished history, going back to Lisp [9]. Compiler writers have been bootstrapping for years. One notable example can be found in Ken Thompson's Turing Award lecture [19]. However, none of these efforts strove for the same development environment goals as Klein. Thus, we confine ourselves to metacircular implementations of dynamic object-oriented languages, developed within reactive environments.

9.1 Squeak

"For the Smalltalk devotee, nothing is more natural than the desire to attack all programming problems with Smalltalk."
—Dan Ingalls et al, in [12]

Squeak is a partially metacircular Smalltalk VM designed to run in small memory systems. Unlike Klein, Squeak is interpreted, and is not exactly written in Smalltalk. Rather it employs a restricted (very restricted) subset, and a non-object-oriented style to allow the interpreter to be translated in C and compiled. For example, this subset omits blocks, message sending, and objects [12].[10] This strategy has been a great success for Squeak — for example, Squeak ports are very easy — but does suffer some drawbacks: Squeak is interpreted, cannot be changed as it runs, cannot be debugged at the Smalltalk-level in the form in which it is deployed, and must incorporate two different mechanisms for finding roots in stack frames (one for Smalltalk stack frames, one for C stack frames).

9.2 Squawk

Squawk is a metacircular JVM™ designed for small devices such as smart cards [18]. Like Squeak, the heart of the VM — in this case the interpreter, and garbage collector — are written in a Java subset that is translated to C and compiled by the C compiler. Thus, it relates to Klein's principles much in the same way as Squeak does.

9.3 Jikes RVM (a.k.a. Jalapeño)

The Jikes RVM was the first high-performance *metacircular* Java VM [1]. Although others had previously built Java VMs in Java, theirs was the first that did not need another JVM to support it.

The Jikes RVM collects all of its unsafe primitives into a "Magic" class. A static check ensures that magic is only used in certain places.

Klein may enjoy more *code reuse* and less duplication than in the Jikes RVM. For example, the object layout code was originally duplicated across the runtime system and three compilers [David Grove, private communication]. It has since been centralized and cleaned up. However, there are still some routines with "modal behavior," that for instance initialize an object differently when running in the Jikes RVM vs. creating a bootstrap image by writing bytes into a byte array. And the absence of "Magic" operations at bootstrap creation time leads to other variations. In Klein, our lens architecture confines all such divergences to an intermediary object that double-dispatches the operations. We are not aware of a comparable factoring in the Jikes RVM system.

The Jikes RVM literature contains little mention of the *reactivity* of the environment. Despite Klein's devotion to fast turnaround, the Jikes RVM is clearly ahead at this point: Grove reports a 35-second turnaround time for building the unoptimized version of the Jikes RVM [David Grove, private communication]. Perhaps their VM runs Java faster than the Self VM runs Self. Or perhaps the Self programming style is a problem here.

Klein aims to support on-the-fly VM modification, including reflective changes to VM objects such as method addition and redefinition. In Self, any object can be sent any message, a freedom that has discouraged us from using vtables. Instead, we rely on inline caching, and a miss-handler coded in a message-free dialect of Self. The Jikes RVM, on the other hand, statically assigns integers to selectors, and hard-compiles vtables for the VM. It will likely be more difficult for the Jikes RVM to support on-the-fly modification. We thus suppose that the Klein VM will prove to be more *reactive* in this respect.[11]

Yet another facet of *reactivity* relates to how concretely the debugger can provide the illusion of direct execution. Both Klein and the Jikes RVM employ a remote debugging architecture in which the debugger runs in a different VM [2], [14]. Unlike the Jikes RVM, the Klein system provides for the debugger to run on a different physical machine by using a socket and a server process. Both systems provide both machine- and source-level debugging for their simpler compilers. The Jikes RVM's optimizing compiler does not support source-level debugging, while Klein as yet has no optimizing compiler. Apparently, the debugging strategy described in the literature is no longer supported in the current Jikes RVM [Michael Hind, private communication].

A reading of [14] seems to confirm our belief in mirror-based *reflection* architectures and pure object-oriented languages.[12] As Java lacked the benefits of a mirror-based reflection architecture, the authors were forced to resort to proxy objects and a specially-modified virtual machine with extended bytecode semantics to deal with proxies. Since the modified VM was based on the Jikes RVM, it seems unlikely that this facility could have been avail-

[10] Klein also employs a similar subset, but only for two stub routines: cloning and inline-cache missing.

[11] Although the Jikes RVM could be extended in the future to support the sort of "hot swap" facility that is in the Java HotSpot™ virtual machine, this facility is not as reactive as Self's abilities. HotSwap [8] employs wholesale class replacement, whereas in Self, it is possible to make any change to any individual object. The difference results from language semantics, Java's implementation of statics, the common use of vtables, etc.

[12] As an example of the latter, "==" is merely another message in Self. Just try to override it in Java!

able from the start. On the other hand, as described in [4], we were able to support remote reflection in Self for Klein with no VM changes.

10. CONCLUSIONS

We are constructing Klein, a virtual machine and a development environment for a Self VM written in Self. In so doing, we have striven to create an artifact that is object-oriented, metacircular, conducive to code reuse, reactive, and consistent with mirror-based reflection. For example, the same reflection system works in both the development environment and the running VM, and the bootstrap image reuses a system originally designed to file-out programs. We have found these principles to be powerful guides to help us build and debug a parsimonious system.

The area that has given us the most trouble is the reactivity of the export cycle: getting the bootstrap image built quickly. We were able to obtain a 15-fold speedup with algorithmic improvements, but its performance still lags behind other systems. We can only speculate that a combination of Self's extremely pure semantics, our extremely factored programming style (with many tiny methods), and the lack of state-of-the-art optimizations in the Macintosh Self VM may be to blame here.

We believe that the task of optimizing the development environment is far more straightforward than that of developing, maintaining, and debugging a sophisticated virtual machine. The principles described in this paper merit serious consideration when creating a virtual machine.

11. ACKNOWLEDGMENTS

We wish to thank Greg Wright for help with the bootstrap image building algorithm, and David Grove and Michael Hind for helping us understand the Jikes RVM and its development environment. Bernd Mathiske, Cristina Cifuentes, Greg Wright, Kristen McIntyre, Leo Ungar, Michael Furman and Mario Wolczko all reviewed versions of this paper and helped us considerably.

12. BIBLIOGRAPHY

1. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. 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. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. IBM System Journal, Vol 39, No 1, February 2000. http://www-106.ibm.com/developerworks/java/library/j-jalapeno
2. Bowen Alpern, C. R. Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, Stephen Smith. Implementing Jalapeño in Java. In Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA`99), volume 34.10 of ACM SIGPLAN Notices, pages 314-324. ACM Press, November 1999.
3. Ole Agesen and David Ungar. Sifting Out the Gold: Delivering Compact Applications From an Exploratory Object-Oriented Environment. In OOPSLA'94 Conference Proceedings, Portland, OR.
4. Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In OOPSLA'04 Conference Proceedings,, October 2004.
5. Craig Chambers. The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. Ph.D. Dissertation, Stanford University, 1992.
6. Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented Programming Language. In PLDI'89, pp. 146-160, Portland, OR, June, 1989.
7. Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, October, 1989.
8. M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. Presented at the Workshop on Engineering Complex Object-Oriented Systems for Evolution, at OOPSLA'01, October, 2001.
9. Urs Hölzle. Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming. Ph.D. Dissertation, Stanford University, 1994.
10. Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In PLDI'94 Conference Proceedings, pp. 326-336, Orlando, FL, June 1994.
11. Urs Hölzle, David Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In OOPSlA'94 Conference Proceedings, Portland, OR, August 1994.
12. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. The Story of Squeak, A Practical Smalltalk Written in Itself. In OOPSLA'97 Conference Proceedings, Atlanta, GA (October 5-9, 1997), pp. 318-326. http://users.ipa.net/~dwighth/squeak/oopsla_squeak.html
13. Krasner, Glen (editor). Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, 1983.
14. Ton Ngo and John Barton, Debugging by Remote Reflection, Euro-Par 2000, Munich, Germany, August 2000. http://jikesrvm.sourceforge.net/info/pubs.shtml#remotereflect
15. Ian Piumarta. Late Binding and Dynamic Implementation. Invitational Workshop on the Future of Virtual Execution Environments, Sept. 2004. http://www.research.ibm.com/vee04/video.html#piumarta
16. Self language website. http://research.sun.com/self
17. Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In ECOOP '95 Conference Proceedings, Aarhus, Denmark, August 1995.
18. Nik Shaylor, Douglas Simon, William Bush. A Java Virtual Machine Architecture for Very Small Devices. In LCTES'03 Conference Proceedings, pp 34-41, San Diego, CA, July 2003.
19. Ken Thompson. Reflections on Trusting Trust. In CACM, Vol. 27, No. 8, August 1984, pp. 761-763. Also in ACM Turing Award Lectures: The First Twenty Years 1965-1985, Copyright 1987, and in Computers Under Attack: Intruders, Words and Viruses, Copyright 1990, both from the ACM press. http://www.acm.org/classics/sep95/
20. David Ungar. Annotating Objects for Transport to Other Worlds. In OOPSLA'95, pp 73-87, Oct. 1995.