Sun Labs document #96-0506

Towards a Universal Implementation Substrate for Object-Oriented Languages

Mario Wolczko, Ole Agesen, David Ungar

Sun Microsystems Laboratories[1]

{Mario.Wolczko, Ole.Agesen, David.Ungar}@sun.com

Abstract. Self is a minimalist object-oriented language with a sophisticated implementation that utilizes adaptive optimization. We have built implementations of Smalltalk and Java by translation to Self. These implementations were much easier to construct in Self than by conventional means, and perform surprisingly well (competitively with conventional, commercial implementations). This leads us to believe that a Self-like system may form the basis of a universal substrate for implementation of object-oriented languages.

This paper was presented at the OOPSLA 99 workshop on Simplicity, Performance and Portability in Virtual Machine Design, Denver, CO, Nov 2, 1999.

1 Building virtual machines is hard

Building a high-performance virtual machine (VM) for an object-oriented language such as Smalltalk or Java™[2] is a major undertaking. The conventional approach has been to hand-craft the entire VM in a systems language such as C or C++. The low-level access afforded by such a language, compared to using a higher level language, enables an efficient implementation of necessary components, such as a garbage collector, bytecode interpreter, dynamic compiler, and object storage system. Access to the hardware is important because many central data structures and algorithms must be tightly coded if the resulting VM is to perform well and use memory efficiently. Details such as object layout, inner loops of the garbage collector, and code sequences emitted by a dynamic compiler, must all be well tuned for maximum performance. C and C++ serve the role of "structured assembly language" in these cases, because the code emitted by optimizing C and C++ compilers is predictable and efficient. Performance of certain sections of code, such as the message dispatch sequence, is so critical that in some circumstances even these languages are not suitable, and hand-crafted machine code sequences are used to squeeze the last drop of efficiency from the machine.

Hand-crafting a VM in this way is very laborious. The efficiency gained by using C or C++ is at the cost of security; array accesses are not checked to see whether they are within bounds, errors in memory management can lead to memory leaks or dangling pointers, and unchecked type coercions can result in erroneous accesses. A high-performance VM will contain many tens of thousands of lines of source code (e.g., the Self VM

1. Correspondence may be addressed to the first author at: MS UMTV29-117, 2550 Garcia Ave., Mountain View, CA 94043, U.S.A.
2. Sun, Sun Microsystems, Java, JavaSoft and JDK are trademarks or registered trademarks of Sun Microsystems Inc. in the US and other countries. SPARC, UltraSPARC and SPARCstation are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries

is over 100,000 lines of C++ and 3,000 lines of assembly code), and there will be many global invariants involving the object storage system, garbage collection structures, compiled code conventions, and the like.

A single slip can result in a bug which manifests itself as a hard-to-reproduce anomaly in VM execution, the cause of the problem being far removed from the effect. The problem of debugging is aggravated by history-sensitive state changes made for performance reasons, such as adaptive optimizations [7] and changes of representation of run-time state such as stack frames [4, 9]. For example, in an pre-release version of Self it was observed that the system would run correctly for tens of minutes but thereafter input keystrokes would be occasionally lost. After a significant number of days were spent debugging the problem, it was found that a reference to an uninitialized structure was being passed on the stack to a system input routine. Initially, the stack locations involved would happen to be zero purely by chance, and the input routine would behave as expected. However, after some minutes of execution the dynamic optimizer was re-optimizing the Self method that called the input routine, and the optimized method would sometimes leave non-zero values in the locations subsequently occupied by the uninitialized structure, causing the input routine to discard keystrokes. As this example shows, it is easy to introduce a subtle error, which can be very difficult to find and fix, into a complex VM coded in a low-level language.

Changing a design decision in a VM can result in major changes requiring months of effort, culminating in a significant bug tail. Another example taken from the Self VM is the decision to alter the run-time system so that methods emitted by the primary, non-inlining compiler are not customized (i.e., receiver-type specific). This apparently simple change, initially estimated at two weeks' effort, took several months to make, because the assumption that a method was always created for a single receiver type permeated the code of the VM in subtle and diverse ways.

In summary, building a high-performance VM typically requires several man-years of effort by those experienced in the field (e.g., the Self VM, in its present form, is the result of some 25 man-years of work), and results in a large, complex program, which is difficult to modify and adapt.

1.1 Building Smalltalk and Java implementations in Self was easy

As part of other research [19, 2], we have built experimental implementations of Smalltalk [5] and Java [6], by translation into Self [17]. Both of these implementations were relatively easy to build (one and six man-months), and exhibit surprisingly good performance, competitive with commercial, conventional VMs.

Several unique characteristics of the Self language, implementation and programming environment made it possible to construct these implementations rapidly and with good performance. In this paper we will describe these characteristics, outline the implementations, present performance data, and speculate on what these experiences suggest for future VM construction.

2 Unique characteristics of Self

Many properties of the Self language, implementation and programming environment conspired to make it easy to construct the implementations of Smalltalk and Java. This section describes the most important of these properties.

2.1 The Self language

Self [17] is a prototype-based object-oriented language. As in Smalltalk [5], all computation in Self involves objects; even integers and booleans are represented as objects. A Self object is composed of a number of named slots. Each slot contains a reference to an object. Slots are untyped, in that any slot can contain a reference to any kind of object. Some slots contain references to method objects, which are executable. When a message is sent to an object, the slot with the same name as the message is located within the object. If the slot refers to a method, the method is executed. Otherwise, the object referenced by the slot is returned.

Unlike most object-oriented languages, Self uses object-based inheritance, rather than class-based inheritance. Any slots in an object can be designated as parent slots (by appending an asterisk to their names). If a message is sent to that object and the name of the message does not match the names of any of the slots in the object, then message lookup will continue to the objects referenced by the parent slots.

In Self, all operations, even assignment and integer arithmetic, are performed as a result of a message send. There is no invocation mechanism other than messaging.

Self does not include the notion of a class as a basic language concept. In Self, some objects are created directly by the programmer, and thereafter cloned by the running program. For example, the programmer may create an object with slots named `x` and `y` to represent a Cartesian point. This may be treated as a prototypical point, and cloned when the program requires a new point object.

By combining prototypes and object-based inheritance it is possible to build object structures which are analogous to classes. For example, one object might play the role of the class, containing all the behavior, i.e., methods, that its instances require. A prototypical instance is cloned when a new instance of the class is required. The prototypical instance, and its clones, inherit behavior from the class proxy via a parent link.

2.2 The Self implementation

The Self language presents a simple and pure model of computation based on objects. The task of the Self Virtual Machine is to map programs in this model onto the underlying hardware so as to make the program's use of the hardware as efficient as possible. Given the abstractness of the model, and the lack of low-level language features, this is no mean feat. It is made all the more difficult by the requirement that Self programs execute in an incremental programming environment, in which the programmer is free to halt the program at any time, inspect the state, modify any part of the program, and resume execution. This precludes the use of any analysis or optimization technique that would adversely affect the interaction between the programmer and the environment.

To reconcile these requirements, the Self VM uses adaptive optimization based on type feedback [7]. When first executed, a Self method is translated into machine code in a fast but naive way. The translation includes extra code to gather information about the execution profile and the actual objects being used in the execution. Once the VM has determined that a piece of code is being executed frequently, it retranslates one or more methods, utilizing the information gathered earlier to guide optimizations.

In Self, an object is created by cloning another object. Objects that are derived in this way from a single original object are treated by the VM as a clone family [3]. The sharable parts of the members of a clone family are represented as a distinct entity by the VM, known as a map. Maps describe the structure of objects, and contain the constant slots (which will always have the same value for each object in the clone family). Because Self does not allow method slots to be assigned, the methods of a clone family will be held in the map, and hence the VM can treat the map as the implementation type for the clone family. The programmer cannot detect the presence of maps in the system; they are purely an implementation artifact. Maps allow the Self programmer to create prototypes and clones as efficiently as instances of classes in a class-based language.

The most important optimization that is performed by the feedback-driven compiler is method inlining. Given the ubiquitous use of message sending as an operation invocation mechanism, dynamically bound calls are very frequent. Additionally, the Self language promotes a style of programming in which factoring is maximized, resulting in small methods. If compiled naively, programs would run slowly because they would spend all their time in calls and returns. To eliminate the call overhead, the Self VM, when recompiling a frequently executed piece of code, uses the observed types of message receivers (i.e., their maps) to inline method bodies. In most cases the compiler cannot be certain that the observed types will always be the types actually used, so must guard the inlined body with a type test. However, when inlining is performed repeatedly within a method based on an assumed receiver type, a single type test can guard a large region of code which contains many inlined message sends.

In the cases that the compiler chooses not to inline a send (because there is no single type which dominates, or the method body is large and the space occupied would be too great) then the instruction sequence used for message dispatch is chosen very carefully so as to be as fast as possible. When a single receiver type is invoked at a message send site, an inline cache [4] is used for dispatch. This adds only a few cycles of dispatch overhead to the call in the case that the cache hits. When multiple types occur at the call site, a polymorphic inline cache is used [8].

To support a wide variety of programming styles and techniques, the Self object management system is highly tuned to make object allocation, use and reclamation efficient. At peak rate, Self 4.0 on an UltraSPARC™ 1 with a 167MHz processor can allocate, initial-

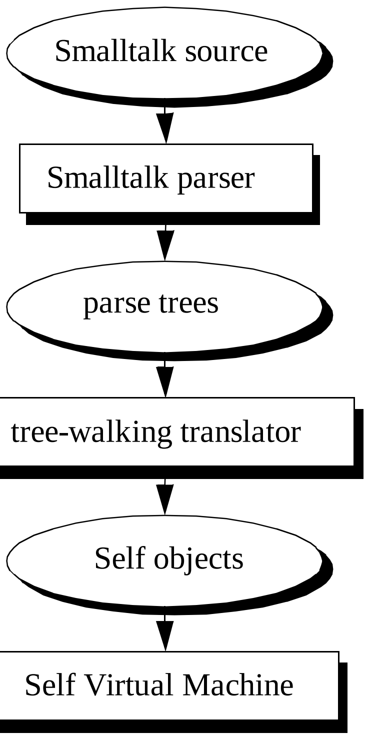


Figure 1. The structure of the implementation of Smalltalk in Self

ize and reclaim 2 million 10-word objects in a second, an allocation and reclamation rate of 75 Mbytes/sec.

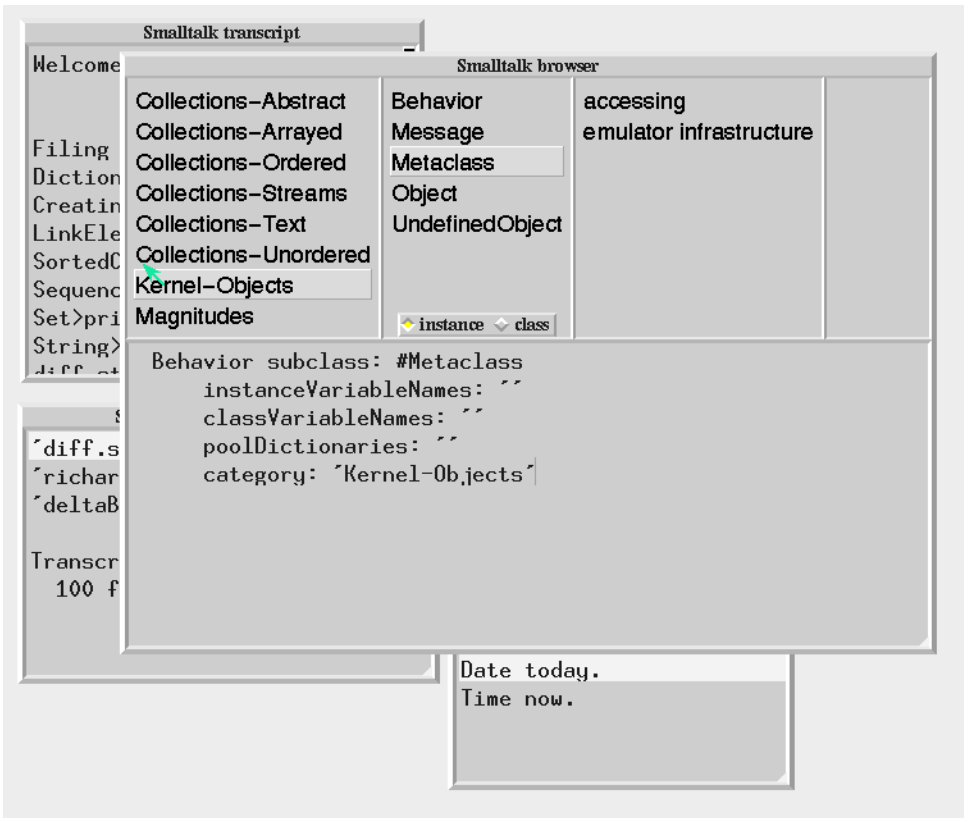
All these properties of the implementation, namely efficient representation of prototypes, profile-guided inlining, adaptive optimization, fast sends and efficient object management, were key to the success of the implementations of Smalltalk and Java.

3 Implementing Smalltalk in Self

The structure of the Smalltalk implementation is sketched in Figure 1. The implementation includes a Smalltalk parser generated by the Self parser-generator, Mango [1], and a translator which generates Self objects from the Smalltalk parse trees. The Self objects mimic Smalltalk classes and methods. The Smalltalk implementation was written entirely in Self, and did not require any changes to the Self Virtual Machine.

Additionally, there is a rudimentary development environment, comprised of browsers, workspaces, inspectors and a transcript. These were constructed in Self, and provide the usual facilities of browsing, editing and running code (see Figure 2).

Because the design of Self was heavily influenced by Smalltalk, much of the translation was straightforward and its description will be omitted. In particular, the semantics of method invocation in Smalltalk are a subset of those provided by Self. However, the rep-



```
Behavior subclass: #Metaclass
    instanceVariableNames: ´´
    classVariableNames: ´´
    poolDictionaries: ´´
    category: ´Kernel-Objects´
```

```
Date today.
Time now.
```

Figure 2. The Smalltalk-in-Self programming environment

resentation of instances, classes, metaclasses and their associated behaviors is of more interest.

To capture these behaviors, we chose to represent each Smalltalk class by a class proxy. The class proxy responds to messages in the same way as the original Smalltalk class it mimics. The behavior of its instances is contained in a separate object (the instance method repository), however, because an instance does not inherit its behavior from its class. The instance method repository is referenced from the class, and is inherited by all instances.

A metaclass is structured analogously to a class, i.e., there is a metaclass proxy and a method repository. The metaclass's method repository serves as the inherited behavior of the class. To support instance creation, this repository also contains a reference to a prototypical instance of the class, which is created when the class is defined, and updated whenever the structure-defining aspects of the class or any of its superclasses are changed. To implement the Smalltalk 'new' primitive, we shallow-copy the prototypical instance.

Figure 3 shows the skeleton of a sample class, `Date`. The Self objects representing that class and its metaclass are shown in Figure 4. The class variables of `Date` are in a separate object, inherited by both the instance and class methods.

```
class                   Date
superclass              Object
instance variables      days
instance methods
addDays: d
  ...
class methods
initialize
  ...
```

Figure 3. The (partial) definition of the Smalltalk class `Date`

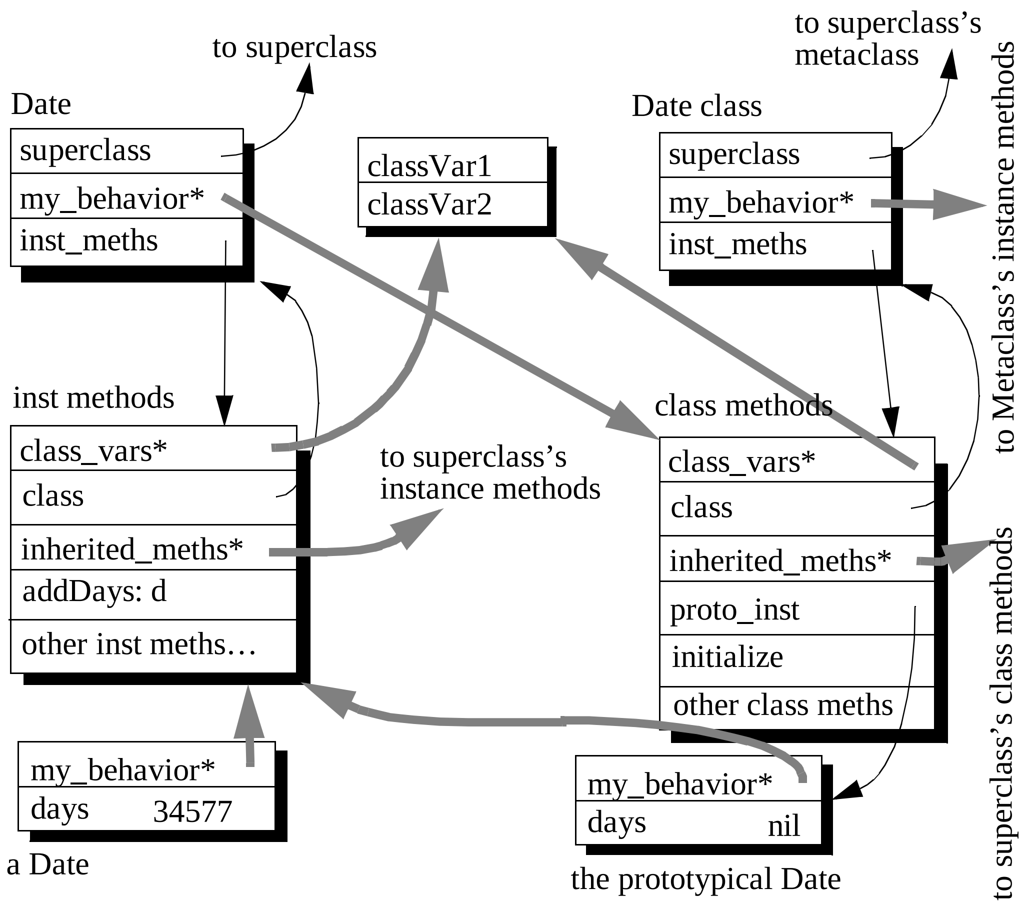


Figure 4. The Self objects representing class Date and an instance

Each class and metaclass is thus represented by a group of six objects: the class proxy, the metaclass proxy, the instance method repository, the class method repository, the class variable repository, and the prototypical instance. Each group is connected to others by inheritance links to mimic the Smalltalk class-metaclass hierarchies.

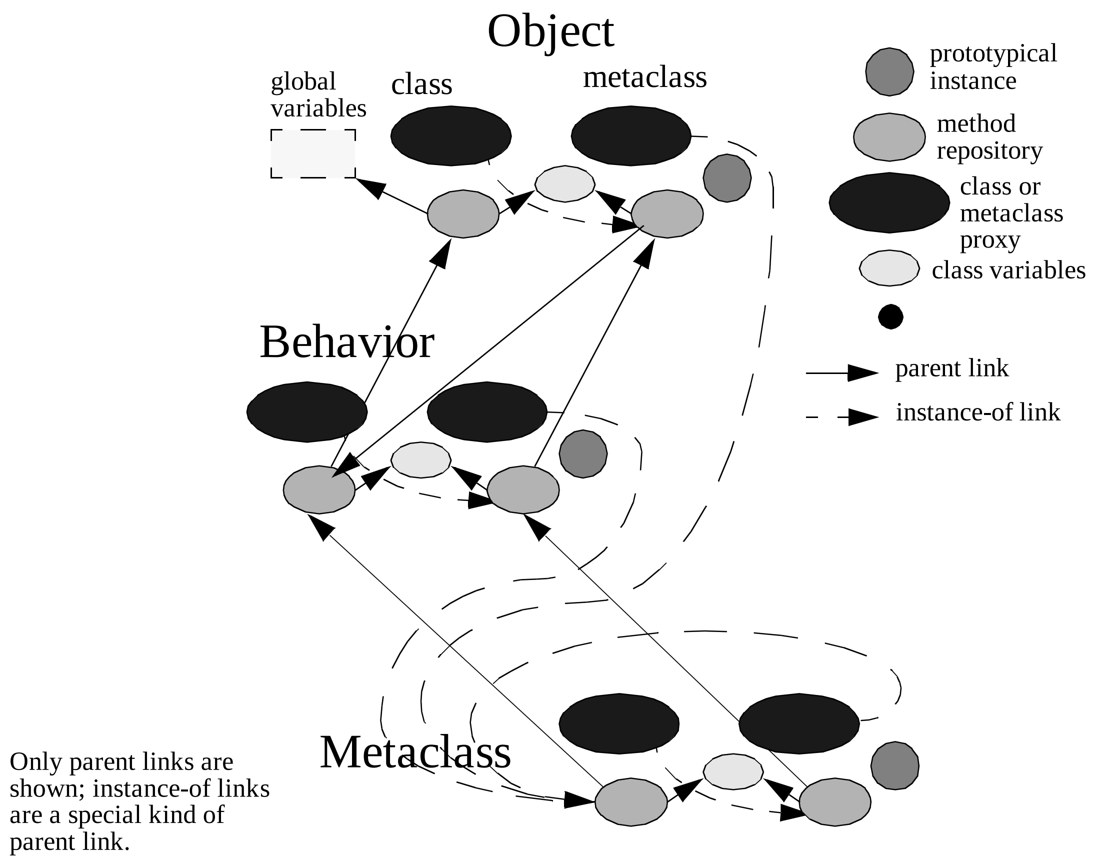


Figure 5. How the basic classes are connected

The classes and metaclasses at the core of a Smalltalk system, namely `Object`, `Behavior` and `Metaclass`, are shown in Figure 5. Each class is an instance of its metaclass. The metaclasses are instances of the class `Metaclass`, which is in turn an instance of its metaclass. Smalltalk global variables are held in a global variable repository inherited by the instance method repository of `Object`. This triad of classes was created ‘by hand’ using the Self interactive programming environment, using ‘arrow-dragging’ to build the network of objects [15]. Once the appropriate subclass creation behavior was added, the Smalltalk-to-Self translator could extend this hierarchy with new classes.

The Smalltalk class hierarchy is not connected to the hierarchies of standard Self objects, so that no extraneous behavior is inherited. The only exceptions to this are that Smalltalk `Booleans`, `SmallIntegers` and `Floats` are represented by Self booleans, smallInts and floats, for efficiency. The result is that these objects possess not only the behavior ascribed to them by their corresponding Smalltalk class definitions, but also the behavior inherited from their Self traits,. which could lead to incorrect behavior should the extra behavior be invoked inadvertently. This was deemed to be a minor problem; it could be addressed by renaming the Self or Smalltalk methods in a systematic way so as to make it impossible to invoke a Self method from Smalltalk-derived code.

The only serious problem encountered in the Smalltalk implementation was in the behavior of blocks, which are the Smalltalk and Self form of closures. In Smalltalk, a block can have arbitrary lifetime, even outliving the method context in which it was created. In the Self 4.0 implementation, blocks cannot be invoked once their enclosing context is no longer active. This property of Smalltalk blocks is used quite often in Smalltalk applications. To circumvent the problem, we devised a scheme involving Self objects which mimic blocks, but to determine when to use this scheme we required the Smalltalk blocks needing special treatment to be annotated in the source code [19]. While this solution is not acceptable for a production-quality implementation of Smalltalk, it was sufficient for our experimental implementation. It would be possible to extend the Self VM to support blocks of arbitrary lifetime, but the months of effort required to do this were considered to be beyond the scope of the Smalltalk implementation experiment.

In translating some of the syntactic features of Smalltalk not directly available in Self we could take advantage of Self's advanced compilation technology to make the translation simple. For example, Smalltalk provides a multiple assignment facility, whereas Self does not (in Self, assignment is achieved by a primitive method which returns the object containing the slot assigned to, not the value assigned). In translating a Smalltalk statement like

```
    a := b := c
```

to Self we introduced a temporary variable, by using a block, thus:

```
    a: ([| :t | b: t. t] value: c).
```

A simple implementation of this statement would be quite inefficient, binding and executing a block for each execution. However, the Self compiler inlines the block and its invocation, producing code which is just as efficient as if the multiple assignment were directly available.

3.1 Performance results

To assess the performance of Smalltalk code in this system, we took three medium-sized benchmarks and measured their runtimes in this system and in ParcPlace's ObjectWorks/Smalltalk 4.1, running on the same hardware (a Sun SPARCstation™-10).

In measuring Self programs, one has to be cognizant of Self's dynamic optimization system. This system instruments the initial version of a compiled method, and uses the measurements to recompile hot spots with a much more sophisticated optimizing compiler. Hence, the performance of a program is not constant, but improves asymptotically. The results we quote here are based on the best of 20 runs, which gives the optimizing compiler a chance to generate code close to its optimum. ParcPlace Smalltalk is based on a dynamic translator; the first run of a program will be slower while translation takes place, but subsequent runs will be very similar (as was observed). We took the best of 20 runs, just to make sure.

The selection of benchmarks was limited by incompatibilities in class libraries; our implementation had no graphics classes, and so any benchmark had to be entirely non-graphical. The benchmarks we used were:

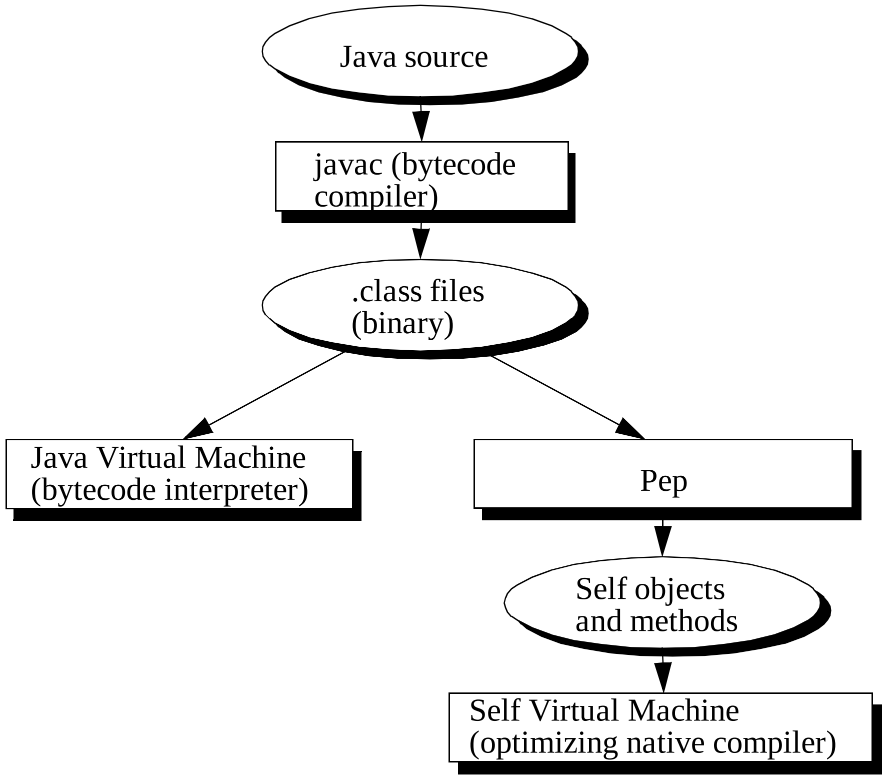
- Richards, an operating system simulator. This benchmark has been widely used in previous implementation experiments. It uses a very small number of library classes and methods, and therefore is a reasonable measure of language performance, independent of class library implementations. Richards shows the Self-based implementation of Smalltalk in its best light, running in 410 ms, 2.7 times faster than ParcPlace Smalltalk. Very little of the class library is used, and so the performance can be directly compared.
- Deltablue, a constraint solver [14], with two separate tests, the chain test and the projection test. The Self-based system runs the chain test in 830 ms, 1.4 times faster than ParcPlace Smalltalk, and the projection test in 450 ms, 2.2 times faster.
- Diff, a program for computing longest common subsequences. Diff spends most of its time in a single loop, doing binary-chopping searches down a list of sorted integers. Although the Self compiler discovers this loop, and inlines all the blocks involved, in Smalltalk this loop is written in terms of messages such as `ifTrue:` and `whileTrue:`, which the ParcPlace Smalltalk compiler treats specially and inlines anyway. The machine code generated for these loops is similar in quality in both systems, and hence the times are similar: 7000ms for the Self-based system, 1.1 times faster than ParcPlace Smalltalk. Both the Self-based system and ParcPlace Smalltalk suffer from relatively poor machine code quality; a C version runs approximately 10 times faster, the difference resulting from much tighter code in the inner loop.

4 Pep: running Java code on the Self VM

Java's rapid growth to become one of the most popular object-oriented languages was not been matched by the development of Java VMs. Consequently, even though the large number of Java systems currently in use amply justifies the construction of a high-performance Java VM, Java programmers today must make do with relatively underpowered virtual machines. For example, JavaSoft™'s current JDK™ 1.0.2 virtual machine is a bytecode interpreter having no native code compilation, using indirect pointers (so-called handles), and conservative garbage collection. In contrast, Smalltalk has had direct pointers, exact garbage collection, and "just-in-time" native compilation for more than a decade [4, 16].

Upon observing the paucity of high-performance Java implementations, we set out to determine the potential of Self-style dynamic optimization techniques for Java programs. Having only limited resources available, we rejected the option of reimplementing Self's optimizer in the context of a Java VM. Instead, we built Pep, a Java to Self translator. By translating Java class files (bytecodes) into Self objects and methods, Pep allows Java programs to execute on the Self VM where they benefit from the optimizing Self compiler.

Figure 6 shows a Java program's path from source to execution using either the Java VM (left branch) or Pep and the Self VM (right branch). The initial version of Pep translated Java code into Self source code that would run on an unmodified Self VM. Although avoiding VM changes was not a goal in itself, to get to run Java programs sooner we preferred "above the line" solutions requiring no VM changes. Our experience with this first version of Pep made us revisit some of the design choices. We subsequently implemented a second version of Pep, this time admitting VM changes in carefully selected areas to ensure better performance. These changes not only improved the performance of the Pep system, but enhanced the Self VM's ability to serve as a basis for general language implementation. The following subsections describe the initial version of Pep, the observations that motivated the shift from the first to the second version, and how the second version differs from the first.



Java source
javac (bytecode compiler)
.class files (binary)
Java Virtual Machine (bytecode interpreter)
Pep
Self objects and methods
Self Virtual Machine (optimizing native compiler)

Figure 6. The Java interpreter and Pep executing a Java program

4.1 Version 1: Generating pure Self source code

Consider again Figure 6. Pep maps Java class files to Self objects and methods. A Java class file is a binary representation of a single class, generated from the source description of the class by the javac bytecode compiler. The format of class files is described in detail in [10], but essentially a class file contains the information necessary to execute programs that use the class: the class's name, its superclass's name, a description of all fields and methods defined by the class, and a so-called constant pool containing all numerical and string constants used in the definition of the class. For each field, the class file specifies the field's name, type, and other attributes (such as whether the field is final and/or private). For each method, the class file specifies the types of the method's arguments and result, bytecodes defining its behavior, and other attributes (such as whether the method is synchronized and/or static).

Pep takes the information in a class file and generates two Self objects. The class object contains a parent pointer to the superclass's class object, translations of all instance and static methods, all static fields, and a lock (for synchronized static methods). The prototype object contains a parent pointer to the class object (for inheriting the instance methods and access to the static variables), a slot for each instance field defined by the class, and a lazily allocated lock object (for methods that synchronize on instances of the class).

The translation of the remaining Java constructs follows largely from the structure established by the class and prototype objects. For example, Pep translates Java code that instantiates a class (using `new`) into Self code that clones the class's prototype object. Java method calls are translated into Self message sends with some use of name mangling (to handle Java's compile-time method overloading based on argument types) and some use of wrapper methods (to accomplish static binding for Java's final calls and lock acquisition and release for synchronized methods). Two specific areas of the translation deserve attention: primitive types and control flow. (More details on all aspects of the translation can be found in [2].)

4.1.1 Primitive types

Java has a rich set of built-in primitive types: signed integers of size 8, 16, 32, and 64 bits, unsigned 16-bit unicode chars (integers), and floating point numbers of size 32 and 64 bits. Self, in contrast, has only 30-bit integers (smallInts) and floats (two bits being used for tagging). To map Java's built-in types onto Self's types, we first settled on a representation and then defined the necessary operations. For all the integer types, the representation we chose was the following: if the value fits in 30 bits (signed), represent the value as a Self smallInt, else represent it as a Self bigInt (bigInts are arbitrary-precision integers defined in Self and represented using a vector of digits and a sign). Thus, for the 8- and 16-bit Java integers, the value would always fit in a Self smallInt and for the 32- and 64-bit Java integers, some values would fit. Having defined the representation, we added methods to the Self smallInt and bigInt objects to express all the operations on Java integers. For example, here's the essence of the 32-bit integer addition method:

```
iadd: i = ( (self + i) trimTo32Bits ).
```

This method adds the receiver (a Self smallInt or bigInt) to the argument `i`, using Self's normal addition routine. Since addition in Self works with arbitrary precision, the result can be outside the 32-bit range. Subsequently, the value is reduced to 32 bits of precision by sending it the `trimTo32Bits` message. BigInts define this method as follows:

```
trimTo32Bits = (
  | modulus = (1<<16)*(1<<16). sign = (1<<15)*(1<<16). |
  "modulus is 2**32, sign is modules / 2"
  r: (self >= 0 ifTrue: self False: [modulus + self]).
  r: r % modulus.
  r >= sign ifTrue: [r - modulus] False: r.
).
```

SmallInts, being 30 bits only, use a much simpler and faster definition:

```
trimTo32Bits = ( self ).
```

The intent behind this smallInt/bigInt dual representation of Java integers is to be fast when possible (i.e., in the 30-bit range), and fall back on the slower bigInt computation when necessary. The Self compiler's inlining is particularly crucial in the 30-bit case, where inlining the above (empty) method eliminates any overhead, allowing the addition to proceed at full speed. We gambled that in practice, few integers would be outside the 30-bit range.

Finally, for Java's floats, we took a short-cut, mapping both the 32- and 64-bit types into Self's 30-bit floats. This choice, while acceptable for an experimental system since few of our benchmark programs rely on the exact behavior of floats, would not suffice for a Java implementation in widespread use. Indeed, we found that the impaired floats broke the `java.util.Random.nextDouble()` method, causing it to always return NaN.

4.1.2 Control flow

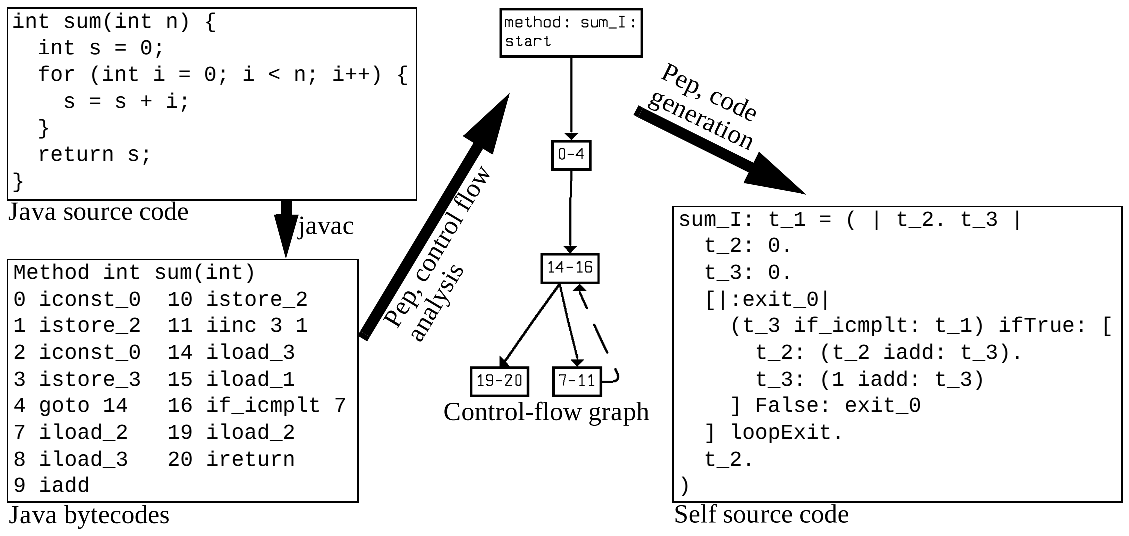
At the source level, Java has structured control-flow, using constructs such as `if-then-else`, `while`, `for`, `break`, `continue`, `switch`, and `try-catch` (for exception handling). The bytecode compiler, however, turns the structured source code into bytecodes with unstructured (goto-based) control flow. Since Self has no goto and no tail-call elimination, Pep must recover structured control flow from the Java bytecodes before they can be translated into Self code.

To recover the structured control flow for a method, Pep partitions the bytecodes into basic blocks, constructs a control-flow graph, computes the depth-first tree to find loops (back-edges), computes dominators (to ensure that the loops nest properly and have single entries), and finally iterates a local pattern matching "reduction" operation on the control-flow graph, in the process building equivalent structured control-flow statements (see [2]). This process usually produces Self code that resembles the original Java code.

Figure 7 summarizes the most important steps in the translation of a Java method to Self code. The Java method `sum` contains a `for` loop that sums all integers less than `n`. The Java bytecode compiler, javac, translates it into the bytecodes shown below the source code. Pep then constructs the control-flow graph, and subsequently turns it into Self source code (for readability, we improved the indentation and abbreviated some of the generated names). The loop in the generated code clearly parallels the original `for` loop, although the loop counter `i` has been renamed to the generated name `t_3` and the summand `s` has become `t_2`.

4.2 Version 2: Beyond pure Self code

The version of Pep described above was highly successful for some Java programs, running them up to an order of magnitude faster than the Java JDK 1.0.2 interpreter. For other programs, however, the Pep-generated code had disappointing performance. For example,



```
int sum(int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s = s + i;
  }
  return s;
}
```

```
Method int sum(int)
0 iconst_0  10 istore_2
1 istore_2  11 iinc 3 1
2 iconst_0  14 iload_3
3 istore_3  15 iload_1
4 goto 14   16 if_icmplt 7
7 iload_2   19 iload_2
8 iload_3   20 ireturn
9 iadd
```

```
sum_I: t_1 = ( | t_2. t_3 |
  t_2: 0.
  t_3: 0.
  [|:exit_0|
    (t_3 if_icmplt: t_1) ifTrue: [
      t_2: (t_2 iadd: t_3).
      t_3: (1 iadd: t_3)
    ] False: exit_0
  ] loopExit.
  t_2.
)
```

Figure 7. From Java source to Self source

javac running on the JDK 1.0.2 interpreter could compile a particular 500 line Java program in 9 seconds on a 50 MHz SPARCstation 10, but on Pep it took 140 seconds! What went wrong?

4.2.1 Problem #1: Slow integer arithmetic

Upon closer examination, we found that javac makes extensive use of `long` integers (64-bit integers), in many cases computing with values that cannot be represented as 30-bit Self smallInts. In fact, javac uses 64-bit integers as bit vectors in its local flow analysis of methods, using a bit for each local variable. In other words, javac violated our assumption that most integers in practice would be of the fast 30-bit smallInt kind.

We briefly considered adding support for 32- and 64-bit untagged integers to the Self VM in order to support Java's integer types directly. However, such a fundamental change, affecting the memory system at its very lowest level, was deemed to be too high an effort. Instead, we took the second-best approach, adding support for "boxed" 32 and 64-bit integers represented as Self byteVectors and with primitives (C routines) to perform 32-bit and 64-bit arithmetic. With boxed integers, arithmetic would still slow down after exceeding the 30-bit range (e.g., each result exceeding 30 bits would require the allocation of a byteVector), but not nearly as much as before. With boxed integers, javac's performance went from 140 seconds to just 13 seconds, an improvement of more than an order of magnitude. This large speed up confirmed that our hypothesis was right, that slow integer arithmetic was contributing to the slowness of javac. Moreover, it suggests that a further performance gain may be possible, once we unbox 32- and 64-bit integers.

4.2.2 Problem #2: Hard-to-optimize control structures (blocks)

Slow integer arithmetic was not the only problem that impacted the performance of Pep-generated code. Some of the control structures that Pep would generate when mapping the unstructured jumps in Java bytecodes to structured control flow made such intricate use of blocks that the Self compiler could not optimize them effectively. Consider this scenario: If a tight loop, defined in terms of blocks, as is the style used in Self programming, can be optimized to the point where no blocks remain in the native code, the loop will execute optimally. However, should just one block remain uninlined, the loop may slow down significantly because the Self VM must clone this block on each iteration of the loop. For some of the poorly performing programs, we observed extremely high allocation rates, suggesting that blocks remained uninlined. Besides loops, two other kinds of control structures caused concern. First, Java's switch statements were being translated into sequential test sequences (if-then-else), because the Self language and VM has no multiway branch. Second, exception handlers had to store closures in the Self heap to maintain a stack of active handlers, a certain way to defeat the compiler's optimizer. But why should blocks degrade performance when executing Java code, a language without closures?

At this point, we undertook the major effort of adding branch bytecodes to the Self VM. This work was almost complete after 4 man-months, attesting to the difficulty of implementing and changing high-performance virtual machines. Presently, branch bytecodes support efficient, closure-less translation of all Java control-structures except exception handlers (for which Pep still emits code with blocks). Using the branch bytecodes, Pep can translate Java bytecodes one by one directly into Self bytecodes, eliminating both the control-flow analysis and the need to generate Self source code.

To illustrate the effects of the branch bytecodes, we measured a version of the Richards benchmark that contains a switch statement in the inner loop. On a 167 MHz UltraSPARC 1, the benchmark takes 380 ms to execute when using no branches in the generated code. Enabling branches reduces the run time to 320 ms.

5 Discussion

Contrasting the one man-month spent on the Smalltalk implementation with the six man-months (or ten man-months, if counting the branch bytecode effort) spent on Pep, several points should be noted. Smalltalk and Self are more similar than Java and Self, so it should be expected that the Java system would require more work. However, the Pep project not only took more time, but also produced a superior Self system for implementing other object-oriented languages: better, although not perfect, 32- and 64-bit integer types; direct support for branches in the bytecode set; and, not discussed in this paper, a process scheduler with multiple priorities; faster and more general locking; and elimination of some I/O bottlenecks.

We have stressed the Self VM with three languages instead of one. It should come as no surprise that this broader exposure has revealed areas of relative weakness and that eliminating these weaknesses would further increase the value of the Self system as a general implementation substrate. The most important areas are:

- Local code quality. The Smalltalk diff program shows the need for better local code quality in the machine code generated by the Self compilers. The diff program's performance, even though all sends have been inlined, cannot compete with the C version.

This observation was also confirmed by our Java experience. Java methods tend to be larger than typical Self methods, raising the relative importance of traditional compiler back-end techniques compared with the message invocation optimizations that the Self compiler emphasizes.

- Efficient access to primitive types implemented by the hardware (in particular 32- and 64-bit integers and floats).
- Life-time of blocks. Self blocks cannot be invoked after their enclosing method returns. Lifting this restriction makes it easier to translate Smalltalk blocks that use this feature, as well as translating other languages with closure-like constructs.

It would be interesting to broaden our experience by implementing other object-oriented languages, such as Beta [11] and Eiffel [12], in this manner. This will undoubtedly expose further deficiencies in the Self VM. Consider Beta for a moment. The top-down method combinator (inner), the interactions of block structure and inheritance, and more simply multiple return values might all warrant additional work on the Self VM. Even so, we feel confident that many of the constructs in other languages will map to Self in a clean and natural way and that overall, implementing a language on top of the Self system will be significantly easier than implementing it directly from scratch.

6 Conclusions

We have described implementations of Smalltalk and Java which have respectable performance and yet were relatively easy to construct in Self compared to traditional virtual machine implementations in C and/or C++. These implementations were simple to build because they utilize a fairly direct mapping of source language constructs to more primitive elements available in Self (prototypical objects, slots, object-based inheritance, messages, etc.). The basic elements of Self semantics seem to form a minimal and yet general and useful object language capable of describing (almost) all of the object behaviors of other languages. Thus far we have only demonstrated this for two languages, Smalltalk and Java, and it could be argued that Smalltalk is so much a part of Self's ancestry that it does not count. However, the connections between Java and Self are much more tenuous, and yet the Java implementation was still easier to build by at least an order of magnitude compared to conventional approaches. We encourage language designers and experimenters to use the Self system in this way for their experiments.

Of course, it would be easy to provide a general implementation substrate for a wide variety of object-oriented languages if this substrate did not have to yield good performance. Hence, to be useful for this purpose, the Self VM has to incorporate implementation techniques which map combinations of these simple elements into efficient machine code and data structures. We feel the key implementation techniques are:

- Adaptive optimization based on feedback. It is senseless to devote equal effort to compiling all parts of a program when all real programs spend most of their time in a small part of the program. By concentrating the compilation effort on those parts that dominate the profile, more sophisticated optimization techniques can be brought to bear. The

Self VM shows the way in this respect, by performing aggressive inlining, but one can envisage many conventional optimization techniques being applied to improve the quality and footprint of the generated code.

- Efficient object representation through the use of maps. Using maps, the Self VM provides a simple, uniform object model where state and behavior can be located in the most natural place while memory is still used efficiently.
- Aggressive inlining and fast sends. The performance-neutral decomposition and factoring of code enabled by these optimizations are a great convenience for code generators producing Self code: they can emit code at a higher level (examples: Pep generates code that manipulates 32- and 64-bit integers; the implementation of Smalltalk's multiple assignments), yet efficiency will be preserved because the Self compiler inlines the definitions of the higher level operations. Most other code generators (e.g., those that produce assembly code) must cope with the complexity of emitting code at a fixed low level (e.g., the hardware level) or pay a performance penalty.
- High-speed object allocation, access and reclamation. Object allocation and access are such fundamental operations – comparable to procedure call in Algol-based languages – that it is essential to make them as cheap as possible. Of course, fast allocation must also be complemented by efficient reclamation.

The problems that were encountered in implementing the Smalltalk and Java systems arose from a lack of sufficiently general facilities in the Self VM, namely blocks with arbitrary lifetimes (Smalltalk), unboxed floats and long integers, and arbitrary control flow within a method (Java). These facilities were not provided in the Self VM because they were not required by the Self language. However, it is now clear that these and other currently absent features could be put to good use in the implementation of a number of languages.

This points the way to an implementation substrate, somewhat Self-like in nature, but containing a rich enough set of primitives to support the implementation of a wide variety of object-oriented languages. After all, it has been many years since static compilation technology became sufficiently well advanced that code generators could be produced which supported a broad spectrum of conventional languages. Many production compilers have back-end systems which produce code for C, C++, FORTRAN, Pascal and similar languages. Perhaps it is possible that a Self-like system could form the basis for a general dynamic compilation system for a variety of object-oriented languages. This would enable language designers to experiment more freely, basing their implementations on a flexible but efficient substrate, and it would allow language implementors to devise techniques which would apply to a multitude of source languages.

7 Acknowledgments

We would like to thank all the members of the Self and Kanban groups at Sun Labs who assisted and encouraged this work, in particular Neil Wilhelm, and John Maloney, who wrote an early version of the Smalltalk class browser. The Smalltalk class library used was derived from the GNU Smalltalk system, by Steve Byrne. The design of the Smalltalk

implementation was made much easier by the first author’s involvement, while at the University of Manchester, in various Smalltalk translation projects, by Borek Vokach-Brodsky, Neil Cope and Ivan Moore. Their work is described in [18] and [13].

References

[1] Agesen, O. Mango—A Parser Generator for Self. Sun Microsystems Laboratories Technical Report, SMLI TR-94-27, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, June 1994.

[2] Agesen, O. Design and Implementation of Pep, a Java Just-In-Time Translator. Submitted for publication, October 1996.

[3] Chambers, C., D. Ungar, and E. Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In OOPSLA’89 Conference Proceedings, New Orleans, LA, October, 1989. Published as SIGPLAN Notices 24(10), October, 1989. Also published in Lisp and Symbolic Computation 4(3), June, 1991.

[4] Deutsch, L.P. and A.M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, Utah, January 1984.

[5] Goldberg, A. and D. Robson. Smalltalk-80: The Language. Addison-Wesley, Reading, Massachusetts, 1990.

[6] Gosling, J., B. Joy, and G. Steele. The Java Language Specification, ISBN 0-201-63451-0, The Java Series, Addison-Wesley, 1996.

[7] Hölzle, U. Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, August 1994.

[8] Hölzle, U., C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In ECOOP’91 Conference Proceedings, Geneva, Switzerland, July, 1991. Published as Springer-Verlag LNCS 512, 1991.

[9] Hölzle, U., C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In Proceedings of the SIGPLAN ‘92 Conference on Programming Language Design and Implementation, p. 32-43. Published as SIGPLAN Notices 27(7), July 1992.

[10] Lindholm, T. and F. Yellin. The Java Virtual Machine, ISBN 0-201-63452-X, The Java Series, Addison-Wesley, 1996.

[11] Madsen, O.L., B. Møller-Pedersen, and K. Nygaard. Object-Oriented Programming in the BETA Programming Language, Addison Wesley/ACM Press, 1993.

[12] Meyer, B. Object-Oriented Software Construction. Prentice Hall, New York, 1988.

[13] Moore, I., M. Wolczko, and T. Hopkins. Babel – A Translator from Smalltalk in to CLOS, Proceedings TOOLS USA, Santa Barbara, CA, Aug. 1994.

[14] Sanella, M., J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. Software—Practice and Experience 23(5), p. 529-566, May 1993.

[15] Smith, R. B., J. Maloney and D. Ungar. The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity and Flexibility. OOPSLA ‘95 Conference Proceedings, Austin, Texas, October 1995.

[16] Ungar, D. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical

Software Development Environments, Pittsburgh, PA, April, 1984. Published as SIGPLAN Notices 19(5), May, 1984 and Software Engineering Notes 9(3), May, 1984.

[17] Ungar, D. and R.B. Smith. Self: The Power of Simplicity. Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June 1991. Also published as Sun Microsystems Laboratories Technical Report, SMLI TR-94-30, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, December 1994. Originally published in OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications, pp. 227-241, Orlando, Florida, October 1987.

[18] Vokach-Brodsky, B. R. B., and M. Wolczko, Smalltalk Application Compilers, Proceedings TOOLS Pacific, Newcastle, Australia, Dec. 1990, p. 69–78.

[19] Wolczko, M. Self includes: Smalltalk. Presented at the Workshop on Prototype-Based Languages, ECOOP '96, Linz, Austria, August 1996.