Published in *TAPOS, Theory and Practice of Object Systems* 3(2), p. 127-155, 1997.

Design and Implementation of Pep, a Java™ Just-In-Time Translator

Ole Agesen

Sun Microsystems Laboratories
2 Elizabeth Drive
Chelmsford, MA 01824, U.S.A.
ole.agesen@sun.com

March 1997

Abstract. Java, a new object-oriented member of the C family of languages, has become popular in part because it emphasizes portability. Portability is achieved by compiling programs to machine-independent bytecodes that can be interpreted on a Java virtual machine. Unfortunately, interpreted performance does not match native code performance. A just-in-time compiler can regain performance without sacrificing portability by turning the bytecodes into native code at runtime.

This idea has a proven track record: Deutsch and Schiffman presented a dynamic Smalltalk compiler in 1984 [5], and the Self system currently sports a dynamic type-feedback based optimizing compiler [12]. To study the performance potential of Java with this state-of-the-art optimization technology, we built Pep, a just-in-time compiler from Java bytecodes to Self. Following translation by Pep, Java programs can execute on the Self virtual machine and benefit from the optimizations performed by Self's compiler.

We describe the design and implementation of Pep, focusing on concepts and trade-offs, but also compare performance with the JDK 1.0.2 and 1.1 interpreters.

1 Introduction

Can Java be made fast enough to remain successful? For many applications, the current speed of Java suffices, but as programmers push the language into the realm of larger customizable and extensible systems, success will depend increasingly on high performance of Java's object-oriented features. Objects must be lightweight to permit their ubiquitous use and allow fine-grained decomposition of data; calls in general must be efficient to permit modularity of code; virtual calls in particular must be efficient to encourage code reuse through subclassing and application framework use.

For a decade, the Self system pursued these goals through extreme object-orientation [21]. In Self, even the smallest units of data, such as integers and booleans, are objects, and all computation, even variable accesses and arithmetic, take place through virtual calls. Despite this complete devotion to objects, Self code runs at up to half the speed of optimized C, the high performance being achieved through dynamic profile-driven optimization.

We built Pep to study the applicability of Self's optimization techniques to Java code. By translating Java bytecodes to Self bytecodes, and executing these on the Self virtual machine, Pep provides insight into the nature of Java programs and the Self system's performance on such code. This paper describes the design and implementation of Pep. While we present some measurements, the main focus of the paper is conceptual. We include enough data to illustrate the trade-offs, gains, and losses, but not so much data that it distracts from the concepts presented. Unless otherwise noted, all measurements report real time on a lightly loaded 167 MHz UltraSPARC™ 1.

The second aim of this paper is broader. Using Pep as a case study, we hope to convey an understanding of the general context in which Java just-in-time compilers operate. Pep's high-level target language (Self) allows us to study dynamic translation without being bogged down by details of assembly code that other just-in-time compilers generate (but this is also an inherent disadvantage: the unusual target language inevitably makes Pep different from other just-in-time Java compilers). Finally, since Pep processes Java bytecodes in a different manner than a Java interpreter

TRADEMARKS. Sun, Sun Microsystems, the Sun Logo, Java, and picoJava are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SPARCstation and UltraSPARC are trademarks of SPARC International, Inc.

and has a radically different target language than other just-in-time compilers, Pep exposes some interesting, and perhaps unforeseen, properties of the Java bytecode set.

The rest of this paper is organized as follows. Section 2 provides background information on Java and Self, emphasizing the properties that motivated the Pep project. Section 3 gives a high-level overview of Pep, focusing on where Pep's translation from Java bytecodes to Self bytecodes fits in the bigger picture. Section 4 describes macroscopic aspects of the translation such as how classes, methods, and calls are translated. Subsequently, Section 5 zooms in on the translation of expressions and statements. Section 6 explains techniques we have applied to reduce the translation overhead and make programs start up faster. Section 7 studies the performance of Pep, and Section 8 describes the current status. Section 9 offers final conclusions and directions for further work.

2 Background: Self and Java

The Self project began a decade ago, with the goal of developing a simpler yet more expressive successor to Smalltalk. In Self [21, 23], even the smallest units of data are objects, and all computation is performed through dynamically-dispatched (virtual) messages. This extreme use of objects constitutes a difficult challenge for efficient execution. To this end, the Self project has pioneered several optimization techniques that have eventually brought the performance of Self up to half the speed of optimized C [12].

Java, an object-oriented member of the C family of languages, appeared recently. It departs from the C/C++ tradition by enforcing type safety, providing automatic memory management, eliminating pointer arithmetic, featuring single inheritance only, and emphasizing portability rather than maximal execution speed. Portability is achieved by compiling Java programs into machine-independent bytecodes represented in a standard class file format [16]. The bytecodes can be interpreted on a *Java virtual machine* (JVM) or translated into native code for faster execution [4]. Currently, interpretation is the most common way to execute Java programs. In this regard, Java implementations resemble the P-code implementation of Pascal, known as UCSD Pascal, developed in the early eighties [17]. Then for Pascal, as now for Java, the goal was portability, and the price of interpreting bytecodes rather than executing native code was a slowdown of about an order of magnitude.

The side-by-side comparison of the Self and Java virtual machines in Table 1 reveals the greater maturity of the Self system. This comparison leads us to ask the question: what would it be like to have a similarly advanced virtual machine available for executing Java programs? Pep was built to provide an answer to this question without having to re-implement the Self virtual machine techniques from scratch in the context of Java.

	Self 4.0 virtual machine	**Java virtual machine (JDK 1.0.2)**
references	direct pointers	indirect pointers (“handles”)
execution	dynamic optimizing compiler for efficiency, non-optimizing compiler for responsiveness	interpretation
method calls	fast; often inlined away through profile directed optimizations	slow; a call/return to an empty method takes 990 ns on a 167 MHz UltraSPARC 1
performance	up to half the speed of optimized C	5x-30x slower than optimized C[a]
memory system	exact, generational, copying GC; allocation from contiguous area	conservative, mostly-compacting mark-sweep GC; allocation from free list

Table 1. Comparison of Self and Java virtual machines.

a. An inner loop written in assembler makes the recently released JDK 1.1 interpreter up to 3x faster. This paper, however, primarily refers to the JDK 1.0.2 release, current at the time when Pep was built.

2.1 Relevant aspects of the Self language

This section briefly describes aspects of the Self language that Pep exploits in the translation. Readers with a detailed knowledge of Self may safely skip this section. Readers who have not previously encountered Self or Smalltalk may wish to supplement this section with a more complete description of Self such as [23].

Objects and inheritance. Unlike Java and most other object-oriented languages, Self has no classes. Instead, objects are self-contained units, made up of named slots that can contain methods or references to other objects. Some of the slots in objects may be designated parent slots. When an object receives a message for which it has no matching slot, the lookup procedure will search the objects in the parent slots of the receiver and, if necessary, their parents, etc. This way, state and methods alike can be inherited. Inherited state is shared between the child and parent, since there is only one slot containing the state (the one in the parent): Self has no notion of copy-down slots.

Object creation. Class-based languages create objects by instantiating classes. Self creates objects by shallow-cloning existing objects (*prototypes*). Since a clone of an object initially contains the same references in its slots as the original, it will, in particular, inherit from the same parents. For example, a `point` object may have `x` and `y` slots for holding the coordinates, and a parent slot for inheriting behavior shared by all points. Any clone of the `point` object will therefore also inherit the point behavior, and, as one would expect, have its own `x` and `y` slots.

Variable access. A characteristic feature of object-oriented languages is the dynamically-dispatched message (virtual call). Self takes dynamic dispatch one step further, using it both to call methods and access variables. This allows methods to override variables and vice versa. From the perspective of the caller, it is impossible to tell whether the receiver object responds to a message by returning (or setting) the value of a variable or by executing a method.

Delegation. Smalltalk and Java, both of which use single inheritance, provide the `super` keyword to allow a method *M* to invoke a method in the superclass of the class containing *M*. Self, having multiple inheritance, generalizes super sends as follows: prefixing a send with `resend` causes the lookup to search all parents of the object defining the method *M*; prefixing a send with the name of a specific parent slot in the object containing *M* restricts the lookup to the specified parent and its parents. We will refer to the latter kind of send as a *directed resend*. In the single-inheritance case, when an object has only one parent slot, `par`, the directed resend `par.msg` is equivalent to the undirected resend `resend.msg`. As we shall see later, Pep makes intensive use of directed resends to translate Java's statically-bound calls.

Control flow and blocks. A block (or closure) is an object that contains a single method, whose code can refer to names of variables from its lexically enclosing environment. Like a `lambda` expression in Scheme, the code in the block does not run until the block is evaluated. For example, a block can be created, passed as an argument, stored in the heap, and later be evaluated. A Self block, like a Smalltalk block, can perform a *non-local return*, that is, a return from the method in which it was lexically defined. A non-local return forces termination of any activations on the stack between the current point and the point where the block was created. For example, if a method M_1 passes one of its blocks to a method M_2, which evaluates the block, and the block does a non-local return, the block's activation record and those of M_2 and M_1 are terminated.

Like Smalltalk, Self uses blocks to implement most control-flow structures, including conditional statements and loops. For instance, conditional statements send the message `ifTrue:False:` to a receiver object, which is `true` or `false`, passing two block arguments. The `ifTrue:False:` method defined in `true` evaluates the first block whereas the `ifTrue:False:` method in `false` evaluates the second block. Self implements loops with the primitive `_Restart`, which restarts the current method or block, i.e., it becomes the body of an infinite loop. To make a finite loop, a non-local return is used to force termination.

3 Overview of Pep

Pep, like other Java execution engines, takes Java bytecodes as input. Certain limitations of Java bytecodes, mainly their low-level control-flow specification discussed in Section 5.5, made us briefly consider basing Pep on source code or abstract syntax trees. In the end, however, we rejected these alternatives because their use would prevent Pep from operating in environments where only bytecodes are available; e.g., Pep would be unable to execute "applets" transmitted as bytecodes over a network. Figure 1 summarizes the path from Java source code to a program execution. The figure shows the paths both for execution with Pep (bottom branch) and with a Java bytecode interpreter (top branch). The Pep path involves more steps than the interpreter path. However, the additional steps are completely encapsulated in Pep: a person interacting with a Java program execution under Pep will not be aware of these extra steps, except for any timing effects they may have. Thus, Pep presents the same model of execution as does a Java interpreter.

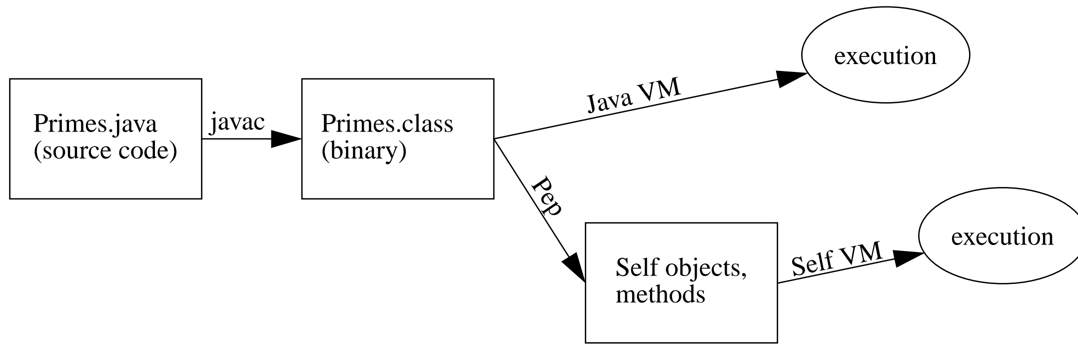


Fig. 1. Pep compared with a Java interpreter.

The next two sections describe in detail how Pep translates Java class files to executable Self code. Section 4 describes macroscopic aspects of the translation, including how Java classes are represented in Self (which has no classes) and the translation of method calls. Section 5 narrows the scope to individual methods, explaining how Pep translates the executable code inside a Java method to equivalent Self code.

4 Macrostructure of the translation: objects, classes, and methods

We illustrate the translation process by following a simple Java class, Primes.java, through the steps sketched in Figure 1. Figure 2 shows the source code for the class. It defines two methods: `isPrime` tests whether a given integer is a prime number (returning `true` or `false`), and `main` uses `isPrime` to print all primes between 2 and 50. The class also defines an instance variable, `maxPrime`, that records the largest prime number found so far.

```
class Primes {
  int maxPrime = 2;

  boolean isPrime(int p) {
    if (p % 2 == 0) return p == 2;
    for (int i = 3; i*i <= p; i += 2)
      if (p % i == 0) return false;
    maxPrime = p;
    return true;
  }

  public static void main(String ign[]) {
    Primes pr = new Primes();
    for (int p = 2; p < 50; p++)
      if (pr.isPrime(p)) System.out.print(p + " ");
    System.out.println();
  }
}
```

Fig. 2. Source code: Primes.java.

Javac, the Java source-to-bytecode compiler, translates a source file to a set of binary class files (one per class defined in the source file). Figure 3 shows parts of the information contained in Primes.class:

- the name of the superclass, here `java.lang.Object` since no other superclass was specified in the definition of class `Primes`,
- a list of interfaces that the class implements (empty in the `Primes` example),
- a constant pool, which contains numerical constants, strings, and references to classes, methods, and fields,

• a description of each field defined in the class, and

• a description, including bytecodes specifying the behavior, of each method defined in the class.

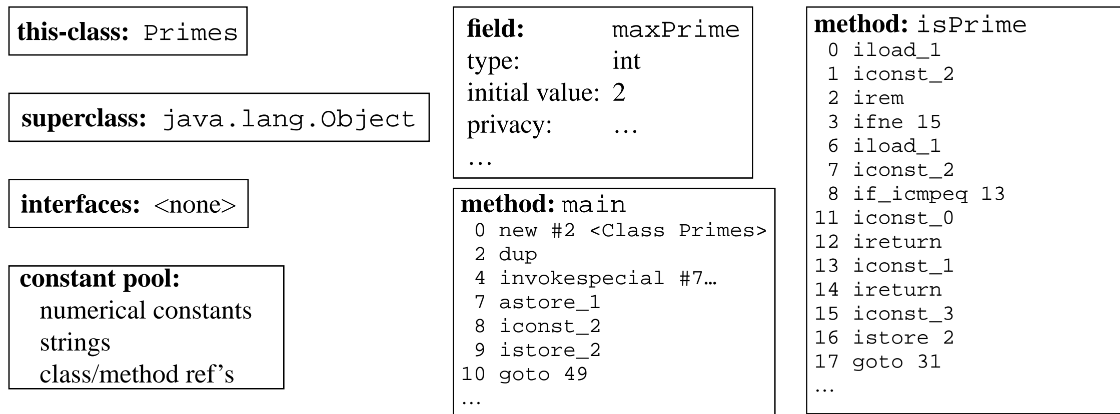


Fig. 3. Binary Java class file: Primes.class.

Pep translates each Java class file into two Self objects: an object defining behavior shared among all instances of the class and a prototypical instance of the class. Although both are simply objects from a Self semantics point of view, for convenience we refer to the former object as a *class object* and the latter as a *prototype object*. Figure 4 shows a Java class hierarchy with four classes and the eight Self objects resulting from translating the classes. For example, the class object for the class `Primes` has the name `class_Primes`; prototype objects are not assigned a name. Arrows denote inheritance relationships. The prototype object, and thereby all clones of it, inherits from the class object. In turn, the class object inherits from its superclass's class object. The figure does not show it, but Pep inserts a root object above `class_java_lang_Object`. The root defines behavior common for all Java values (including `null`, which is not a real object in Java but is in the Pep translation of a Java program).

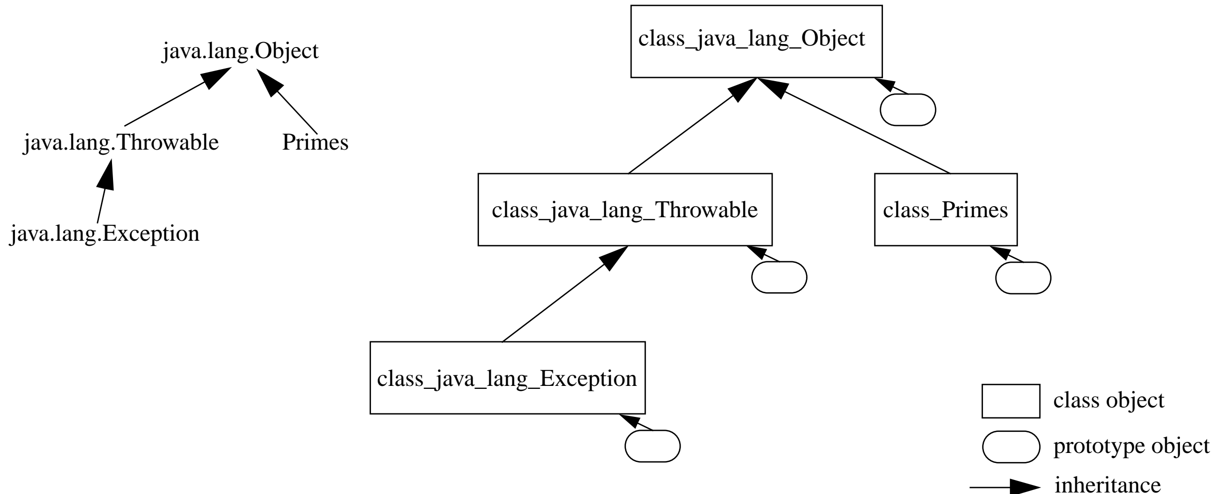


Fig. 4. Java class hierarchy (left) and the Self objects that Pep translates it into (right).

A more detailed picture of the objects generated by Pep can be found in Figure 5, a screen dump from the Self programming environment. The figure shows the class and prototype objects for class `Primes` and the class object for `java.lang.Object`. For readers who are not familiar with the Self programming environment, we should men-

tion that some of the objects' slots in the figure are hidden behind closed outliners. For example, none of the methods defined in `class_java_lang_Object` are visible. In a live environment, they could be exposed by clicking on the black triangular "open category" button at the left edge of the object.

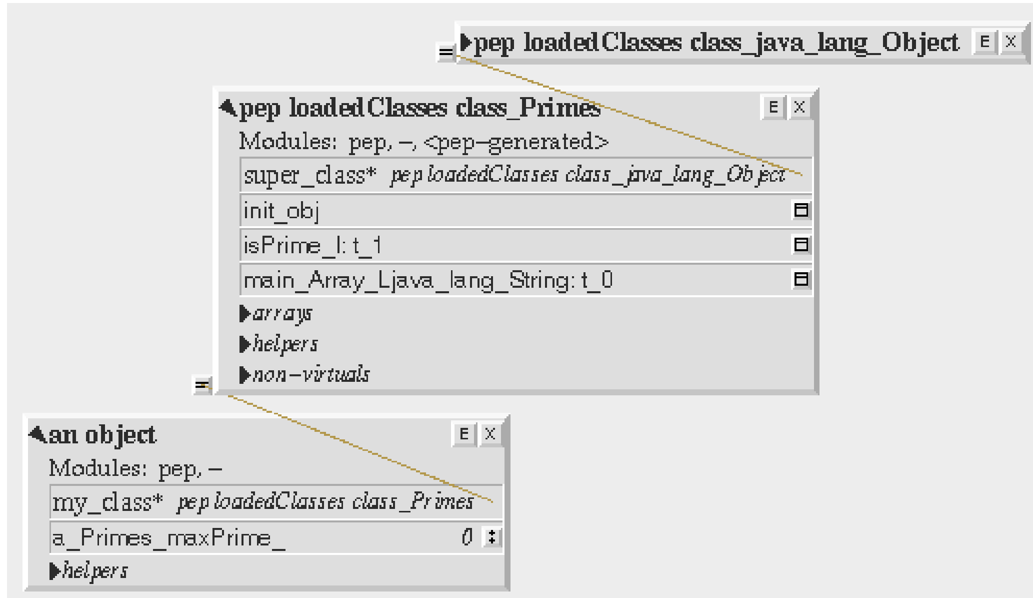


Fig. 5. Self objects produced by Pep when translating class `Primes`.

4.1 Compiling method calls

Java employs a mixture of compile-time and run-time method lookup [10]. The compile-time lookup determines a *method descriptor* for the invoked method. The descriptor consists of the name of the method and types of its arguments (the type of the result does not affect lookup). Essentially, the compile-time lookup deals with method overloading, i.e., cases where several methods have the same name but different types and/or numbers of arguments. The run-time lookup subsequently searches for a matching method descriptor, starting in a certain class and progressing along the superclass chain until a match is found. The JVM specification [16] states that the method descriptor is always found in the immediate class and, hence, no searching of superclasses takes place. This property, in fact, is an implementation aspect of Sun's JVM: at class resolution time, it builds a *method table* by copying the method table of the superclass and then appending the class' own method descriptors. Since the superclass' method table is a prefix of the class' method table, traversal of the superclass chain is unnecessary. C++ vtables use the same prefix layout [22].

The Java bytecode set contains four invocation bytecodes; see Table 2. The bytecodes differ in the class that the run-time lookup starts in, how the lookup is performed and how it can be optimized, and the kind of method invoked. We explain these differences briefly, present an example of the use of the bytecodes, and then describe how Pep translates them; for a complete description of the bytecodes, see [16].

For both `invokevirtual` and `invokeinterface`, the run-time lookup starts in the class of the actual receiver object. The former bytecode is used when the compiler knows a superclass of the actual receiver class, i.e., when the compile-time type of the receiver is a class type. In this case, the invoked method is always found at the same index in the actual receiver's method table, allowing substitution of a fast indexing operation for the method lookup (the Sun JVM performs this optimization by rewriting the bytecode into a "quick" variant; see [16]). The `invokeinterface` bytecode is used when the compile-time type of the receiver is an interface type. In this case, since the possible receiver objects need not have a common superclass that declares the invoked method, the lookup cannot be optimized as aggressively. It is fair to compare `invokevirtual` with C++ vtables and `invokeinterface` with Smalltalk method lookup.

For invokestatic, which invokes a static (class) method, the run-time "lookup" starts in a class specified in the bytecode. We write "lookup" because no searching will actually take place: if a static method is inherited, the compile-time lookup will determine the exact class defining the method and specify this class in the bytecode.

Finally, invokespecial is used in situations where the invoked method can be uniquely determined at compile time, i.e., the invocation can be statically bound (indeed, prior to release 1.0.2, this bytecode had the name invokenonvirtual). The bytecode is used in three different situations. First, since a private method can be invoked only from methods in the same class that declares the private method, a private method call can be statically bound and compiled into an invokespecial bytecode. Second, when compiling an expression like new Primes(), the new object's initializer method (<init>) must be invoked. Since the new objects' class is a compile-time constant, the initializer can be called with an invokespecial bytecode. Third, when a call specifies super as the receiver, it is compiled into an invokespecial bytecode. At run-time, the JVM can tell that the instruction implements a super call (the method is not private and does not have the name <init>). The JVM, then, performs the lookup in the method table of the superclass of the class containing the currently executing method.

bytecode name	class searched for matching method descriptor	use in source terms
invokevirtual	receiver object's class	regular method invocation (like C++ virtual method)
invokeinterface	receiver object's class	invoke method through interface (like Smalltalk method)
invokestatic	statically specified class	invoke a class method (static method)
invokespecial	statically specified class	private method invocation; object initialization (<init>); method invocations using super receiver;

Table 2. The four kinds of method invocation defined by Java bytecodes.

We explained how the JDK interpreter replaces the invokevirtual bytecode at run-time with a quick version. Similar optimizations apply to the other invocation bytecodes [16]. A just-in-time compiler can ignore these optimizations if it compiles methods before they have been interpreted, but otherwise must be capable of translating both the regular invocation bytecodes and their quick counterparts. Pep does not need to handle quick bytecodes since no rewriting of bytecodes take place in the Pep system.

Here is a small example to illustrate Java's compile-time/run-time lookup process.

```
class Invoke {
  private int first(int j) { return j % 1000; }
  int second(int j) { return j % 1000; }
  void test() { first(10000); second(20000); }
}
```

The class Invoke defines three methods: test, first and second. The test method calls each of the other methods. Figure 6 shows the bytecodes for test. Since first is private, the call is translated into the statically-bound invokespecial bytecode. The call's method descriptor is Invoke.first(I)I: the method is declared in the class Invoke, has name first, takes one Integer argument (and returns an Integer, although this fact is not used during method lookup). The call to second cannot be statically bound because the method may be overridden in subclasses of Invoke. It is therefore translated into an invokevirtual bytecode.

```
PC   instruction
 0   aload_0
 1   sipush 10000
 4   invokespecial #5 <Method Invoke.first(I)I>
 7   pop
 8   aload_0
 9   sipush 20000
12   invokevirtual #4 <Method Invoke.second(I)I>
15   pop
16   return
```

Fig. 6. Bytecodes for the test method.

Self emphasizes run-time lookup to a much higher degree than Java and has no separate compile-time lookup phase. To achieve efficient execution of Java programs, Pep must avoid overhead in the translated code implementing method calls. For example, executing Self code to "interpret" Java calls would be inefficient. We have devised a solution that allows Pep to compile a Java call directly into a Self call. This gives the Self compiler a better chance of optimizing and inlining the call than if a less direct translation was used. The solution involves two parts: use of name mangling to resolve overloading and use of delegation to make the lookup start in the appropriate place.

Resolving overloading. Consider a Java method call with the method descriptor $N(A_1, A_2, \ldots, A_n)$; i.e., the method name is N and it takes arguments of types $A_1, A_2, \ldots, A_n$. For this call to match a method, both the name N and argument types A_i must match. In Self, as in Smalltalk, the types of arguments play no role during lookup. The only requirement is that the method names (a.k.a. *selectors*) match. To implement the stricter Java matching rules, Pep folds the Java argument types into the Self selector. The Java method descriptor $N(A_1, A_2, \ldots, A_n)$ becomes the Self selector $N_A_1{:}A_2{:}\ldots A_n{:}$. Accordingly, a method call and a method definition have matching names in the Self domain if and only if they have matching names and types in the Java domain. A few concrete examples should make this idea clear; see Table 3. (The extra "`L`" before class names originates from Java's encoding of class names; it is nonessential and harmless to the present discussion.) The last example in the table shows how array types are handled by adding `Array_` in front of the class name as many times as the array type has dimensions.

Java method $N(A_1, A_2, \ldots, A_n)$	**Self selector** $N_A_1{:}A_2{:}\ldots A_n{:}$
`ensureCapacity(int)`	`ensureCapacity_I:`
`indexOf(java.lang.Object)`	`indexOf_Ljava_lang_Object:`
`setElementAt(java.lang.Object,int)`	`setElementAt_Ljava_lang_Object:I:`
`copyInto(java.lang.Object [])`	`copyInto_Array_Ljava_lang_Object:`

Table 3. Selected methods from the class `java.util.Vector` and their mapping to Self.

Starting the lookup in the right place. For the `invokevirtual` and `invokeinterface` bytecodes, the lookup starts in the class of the receiver object. Since this starting place is the default one in Self, Pep simply translates these bytecodes into regular dynamically-dispatched sends to the receiver object. Pep translates `invokestatic` into a dynamically-dispatched send to the class object specified in the bytecode (recall that the class object is just a regular Self object). Thus, a static method executes with a class object as the receiver and therefore can access static variables defined in the class, but not instance variables. For the `invokespecial` bytecode, recall from the above explanation that we need to invoke a method from a statically specified class. This case requires the most work to translate, simply because Self emphasizes dynamically-bound calls to an extreme degree.

We use Self's directed resend (delegation) feature to force a lookup to start in a specified class. In typical Self programming, a directed resend is used to invoke an overridden method in a specific parent object. However, directed resends can also delegate through non-parent slots. Pep exploits this property, giving each class object a constant slot, `this_class`, that contains the class object itself. Methods defined in the class can perform statically-bound calls to other methods in the class by delegating through the (non-parent) `this_class`. To allow methods defined in other classes to perform statically-bound calls, we add an externally accessible wrapper method that performs the delegation through `this_class`. For example, consider the class `Primes`. In addition to translating and installing the `isPrime_I:` method in the class object, Pep generates this wrapper to obtain statically-bound calls:

```
a_Primes_isPrime_I: t_1 = ( this_class.isPrime_I: t_1 ).
```

The name of the wrapper is the concatenation of the class name and the method name (in this case, an additional "`a_`" is prepended to prevent the first letter from being upper-case since Self rejects method names that begin with an upper-case letter). In Figure 7, the category *non-virtuals* contains all the wrappers for the class `Primes`. By uniqueness of the wrapper method names, a statically-bound call to a method can be replaced by a dynamically-bound call to the wrapper. Hence, the `invokespecial` bytecode can be translated easily.

The use of delegation through `this_class` can be compared with the `this` and `super` keywords in Java. Consider the classes *A*, *B*, and *C*, where *C* extends *B* and *B* extends *A*. If a method *m* in class *B* contains a call of the form `this.foo()`, the lookup starts in the dynamic class of the receiver (which could be *B* or *C*). If *m* contains the call `super.foo()`, it will always invoke the `foo` method in *A*, regardless of the class of the current receiver. Finally, turning to the Pep generated objects, if the class object produced from *B* contains the call `this_class.foo`, it invokes the `foo` method in class *B*, regardless of the class of the current receiver.

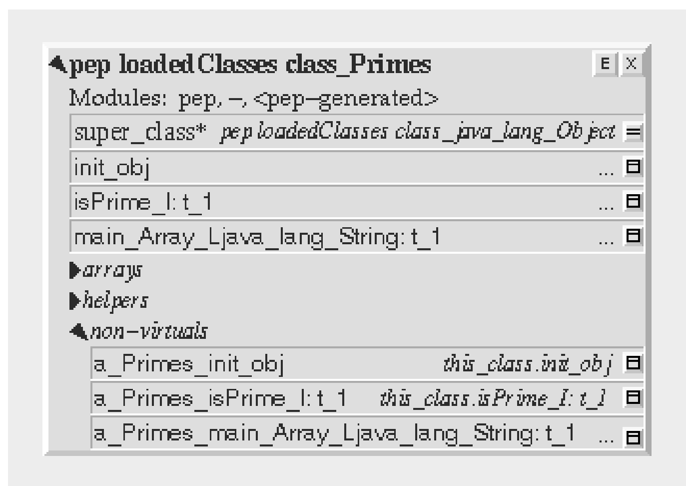


Fig. 7. Pep uses wrapper methods and delegation to implement statically bound calls in Self.

4.2 Synchronized methods

We have explained how Pep uses wrapper methods to implement statically bound calls in Self. Java's synchronized methods are also translated using wrapper methods. A synchronized method must acquire the lock on the receiver object before it starts executing and release the lock once execution completes. Pep implements this behavior by introducing a wrapper method between the caller and the actual code of a synchronized method. The wrapper locks and unlocks the object, taking care to ensure that the lock is released even if an exception abruptly terminates the execution of the synchronized method. Figure 8 shows examples of synchronized methods: the methods `setSeed`, `next`, and `nextGaussian` in class `java.util.Random`. The wrapper has the name that the translated method would have had, had it not been synchronized. The wrapper performs the synchronization and invokes the non-synchronized version of the method. All wrappers have the same form as the one highlighted in Figure 8, invoking a method `sync_do:` with a block that calls the wrapped method.

To give the full picture, consider where the wrappers for implementing statically-bound calls fit in. A statically-bound call to the synchronized `nextGaussian` method will result in this call path in the translated code:

… → `java_util_Random_nextGaussian` → `nextGaussian` → `sync_do:` → `nosync_nextGaussian`.

4.3 Fields

A class in Java has all the fields of its superclass (recursively) in addition to the ones defined in the class itself. Accordingly, when generating the prototype object for a class, Pep collects field definitions from the entire chain of superclasses from that class up to the class `java.lang.Object`.

In Java, as in C++, field accesses are statically bound and fields cannot override methods or vice versa. A subclass can even define a method with the same name as a field in one of its superclasses. Since Self uses dynamically-dispatched sends to access both fields and methods, if a child object defines a method with the same name as a field in a parent object, the code in the parent would access the method instead of the field. The first step in translating fields from Java to Self, therefore, is to make field names distinct from method names. Pep appends "_" to field names to achieve this separation. (Pep employs a simple trick to ensure that names with a single underscore cannot clash with a name from a Java program: whenever Pep translates a name found in a Java program, underscores are "doubled." Thus, `apple_` becomes `apple__`, and therefore cannot clash with the mangled name of a Java field.)

A perhaps surprising feature of Java is the ability to define a field with the same name in a class and its superclass. Instances of the class will then have two fields with the given name. Methods defined in the superclass will assign to and read the field defined there. Methods defined in the class will assign to and read the field defined in the class itself and ignore the definition in the superclass. To translate this behavior into Self, which does not allow a single object to contain multiple fields with the same name, Pep mangles field names by prepending the name of the class declaring

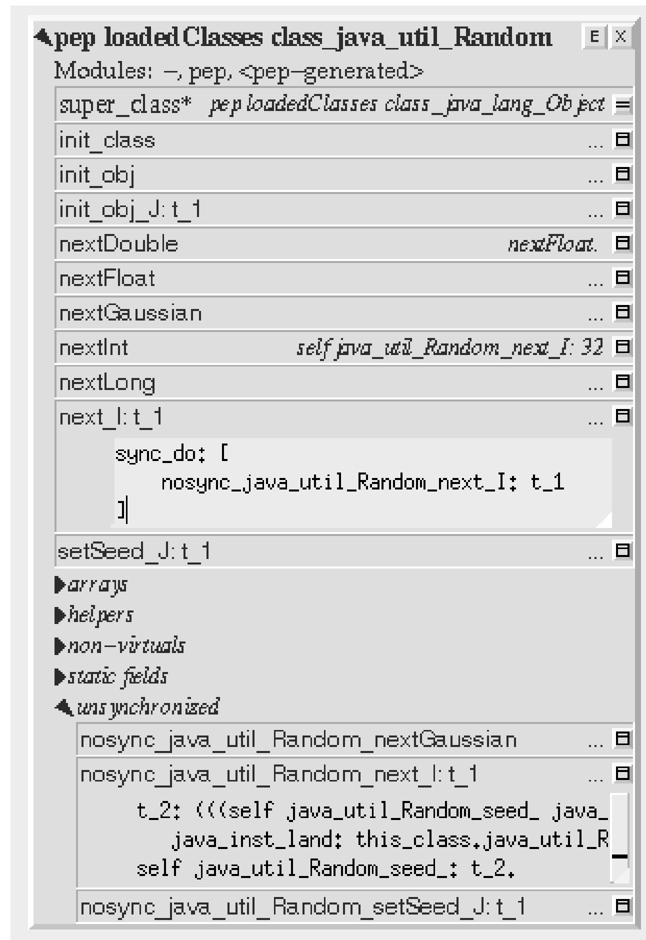


Fig. 8. Pep uses wrapper methods to implement synchronized methods.

the field. For example, the `maxPrime` field in class `Primes` is translated as `a_Primes_maxPrime_` (again, the initial "`a_`" is necessary because Self does not permit names to start with an upper-case letter).

With field names separated from method names by appending "`_`" and made unique by preprending class names, Pep can straightforwardly compile field reads and writes into dynamically-dispatched sends.

4.4 Summary of the macrostructure of the translation

This concludes our description of the high-level aspects of Pep's translation. Table 4 summarizes the most important aspects of the mapping. Next, we consider the details of translating the executable code found inside methods.

Java	**Self**
class	2 objects: prototype and class
superclass	parent slot in class object
unsynchronized method	2 methods: virtual and non-virtual
synchronized method	3 methods: virtual, non-virtual and unsynchronized
non-static field	slot in prototype object
static field	slot in class object
object creation (new)	clone prototype object
method call	method call

Table 4. Summary of how Pep maps Java constructs to Self constructs.

5 Microstructure of the translation: expressions and statements

In this section, we focus on the translation of the bytecodes that define the behavior of methods. Both Java and Self define behavior using bytecodes that conceptually execute on stack machines. However, the bytecode sets are quite different. The Java bytecodes are described in detail in [16]. There are many of them because many operations on primitive data types (integers, floats) have their own bytecodes to allow fast interpretation. For example, the bytecode for adding two 32 bit integers is different from the bytecode for adding two 64 bit integers. Java bytecodes implement control-flow using branches (unconditional, conditional, and switched), have direct support for Java's exception and synchronization schemes, and support all of Java's primitive types directly (with the exception of the `boolean` type, which javac compiles into a `byte` representation). In contrast, Self's bytecodes were designed for compactness rather than fast interpretation (Self has always been compiled), provide higher-level control flow primitives in the form of blocks and non-local returns, have no direct support for exceptions or synchronization, and support only 30 bit integers and floats as primitive types. Table 5 gives a side by side comparison of the two bytecode sets.

	Java bytecodes	**Self bytecodes**	**section**
design goal	fast interpretation	minimality, compactness	
number of bytecodes	200 (shown in full in appendix)	7 (described in detail in [2])	
primitive types	8, 16, 32, 64 bit integers; 32, 64 bit floats; 16 bit chars	30 bit integers (2 tag bits); 30 bit floats	5.1
operations on primitive types, e.g., integer add	each operation has its own bytecode	done by calling “primitive” methods	5.1
expression stack operations	several: `dup`, `dup_x1`, `dup_x2`, `dup2`, ..., `swap`, `pop`	none	5.2
exception handling	direct support: exception tables, `athrow`, `jsr` and `ret` bytecodes	minimal: can trap non-local return through activation frame	5.3
synchronization	direct support: `monitorenter` and `monitorexit` bytecodes	not a primitive concept	5.4
control flow	jump based: goto, if-goto, switch-goto	structured: blocks, restart current method, non-local return	5.5

Table 5. Comparison of Java and Self bytecode sets.

These differences in the bytecode sets must be bridged by Pep. The following subsections describe how Pep handles each area.

5.1 Primitive types and their operations

The JVM implements a richer set of primitive types than the Self virtual machine. Java's virtual machine provides signed integers of size 8 (byte), 16 (short), 32 (int), and 64 (long) bits, unsigned 16 bit integers (unicode characters), and floating point numbers of size 32 (float) and 64 (double) bits. The Self virtual machine, reserving two bits for tags, implements only 30 bit signed integers and 30 bit floats. However, unlike Java's primitive types, the Self primitive types are *objects*: virtual calls can dispatch on integers just as they can dispatch on any other kind of object. Thus, Self methods can be polymorphic over primitive objects and “real” objects alike. We use this fact to implement, in Self, data-types that provide the exact semantics of Java's primitive types.

Representation of Java integers. Java integers whose *values* (not bit size) fit in 30 bits, are represented using Self integers. Some 32 and 64 bit integers cannot be represented in 30 bits. For these values, Pep resorts to a “boxed” representation. This dual representation avoids the boxing overhead for the more common small numerical values. Java's 8 and 16 bit integers never need to “overflow” to an alternative boxed representation. Here's an example of how the dual representation is implemented. Java's integer addition, in the case when both integers are represented as 30 bit Self integers, is implemented by this method:

```
iadd: i = (
  self _IntAdd: i IfFail: [int32 add: self And: i]
).
```

The `iadd:` method is defined in a common parent object of all integer objects in Self. The method attempts to add the receiver `self` to its argument `i` using the primitive `_IntAdd:IfFail:`. The primitive operation succeeds if

and only if both `self` and `i` are 30 bit integers and their sum does not exceed 30 bits. If the primitive fails, the block following the keyword `IfFail:` will invoke a more general addition method, `add:`, defined in the object `int32`. The `add:` method converts the two operands from 30 to 32 bit integers and computes their sum with 32 bits of precision.

We tried two different boxed representations for large integer values. The earliest version of Pep used Self's arbitrary-precision integers. However, it soon became clear that performing arbitrary precision arithmetic followed by truncation to 32 or 64 bits was too slow for the relatively frequent use of 32 and 64 bit values in some Java programs. Consider javac. It uses bit-vector operations to flow analyze the program it is compiling. Each bit-vector is represented as a `long` (i.e., it has 64 bits, indexed from 0 to 63). Thus, many integer values exceed 30 bits. When Pep used Self's arbitrary-precision integers, javac took 140 seconds to compile a version of the Richards benchmark whereas on the JDK 1.0.2 interpreter javac took just 9 seconds (measured on a 50 MHz SPARCStation™ 10). Subsequently, we changed Pep to use a fixed-size representation for boxed integers. We chose bytevectors of length 4 to represent large magnitude 32 bit Java integers and bytevectors of length 8 to represent the 64 bit integers. After this change, Pep could run javac on Richards in 13 seconds, a speed-up of more than an order of magnitude. We suspect that even with the more efficient boxed representation, the overhead of doing 32/64 bit arithmetic is still significant. An extreme example is a program that executes these two empty `for` loops:

```
loop 1:    for (int i =          0; i <    1000000; i++);
loop 2:    for (int i = 1000000000; i < 1001000000; i++);
```

Table 6 shows, as one would expect, that JDK 1.0.2 executes both loops in the same time. Pep, however, loses a factor of nearly 200 when the large numbers in loop 2 forces use of boxed integers. From being 16 times faster than JDK 1.0.2 on loop 1, Pep's performance degrades to being 12 times slower on loop 2.

	JDK 1.0.2	**Pep**
loop 1	600 ms	37 ms
loop 2	600 ms	7,200 ms

Table 6. Boxed integers slow down Pep on loop 2.

Representation of Java floating point numbers. The current version of Pep takes a shortcut: it represents Java's floats and doubles using Self's 30 bit floats. Self's floats have the same precision as Java's 32 bit floats, but reduced range (the two tag bits have been taken out of the exponent). While this shortcut allows Pep to run most Java programs, some programs fail because of the restricted floats. For example, the method `nextDouble` in `java.util.Random` always returns NaN[1] because the long-to-double cast produces a floating point value outside the limited range available in Self.

```
public double nextDouble() {
  long l = ((long)(next(26)) << 27) + next(27);
  return l / (double)(1L << 53);
}
```

The next step in evolving Pep from an experimental system to a practical system would be to add support for unboxed 32/64 bit integers and 32/64 bit floats to the Self virtual machine. This addition would require work on the memory system because tag bits can no longer be assumed, so it was beyond the scope of the initial Pep system.

5.2 Expression stack operations

The JVM provides a rich set of bytecodes for manipulating the expression stack. These bytecodes view the stack as consisting of untyped 32 bit words. For example, the `dup` bytecode will duplicate the top-most word on the stack, regardless of whether it is an integer, float, or a pointer. The `dup` bytecode comes in several variants for duplicating one or two words and for placing the duplicated words at various depths in the stack. Likewise, Java provides two pop bytecodes: `pop` removes one word from the top of the stack and `pop2` removes two. Finally, `swap` comes only in one version: swap the top two words on the stack.

1. NaN: not a number; the result of a float computation which has over- or under-flowed to produce an indeterminate value.

Although the stack manipulation bytecodes view the stack as consisting of untyped 32 bit words, Java does impose some type-like restrictions on their use. For example, using `swap` on two words constituting a double on the top of the stack is illegal. Likewise, swapping a pointer on the top of the stack with one of the words in a long immediately below the pointer is illegal, as is using `pop` to discard half of a long on the top of the stack. However, using `pop2` to discard two pointers is legal. As a rule of thumb, the stack manipulation bytecodes must never compromise the integrity of 64 bit values on the stack, even though they view the stack as consisting of untyped 32 bit words.

For any bytecode in a method, including the untyped stack manipulation bytecodes, Gosling explains in [9] that it is always possible to perform an abstract interpretation of the method to determine the type of the word(s) that the bytecode manipulates. The fact that this type is unique, no matter which path to the bytecode flow of control takes, is an invariant established by javac and verifiable by this abstract interpretation. In fact, security concerns require that this property is verified prior to execution.

Self, in contrast to Java, contains no bytecodes for manipulating the expression stack. Instead, Pep generates sequences of other bytecodes to achieve the effect of the Java stack manipulation bytecodes. For example, to pop the top-most value, Pep would invoke a binary method that always returns its first argument (the method was called "`;`" so `4;5` would evaluate to `4`). Here, Pep would rely on Self's optimizing compiler to inline away the actual call to the "`;`" method. To swap and duplicate values on the stack, Pep would transfer the values into local variables and then push them back onto the stack in the order and multiplicity required. Later, we added a pop bytecode to the Self virtual machine, allowing some of these operations to be performed more directly. (The pop bytecode became *necessary* when branches were added to Self's bytecode set; see Section 5.5.2. The reason is that branches introduce the possibility that control-flow paths can meet with different expression stack heights, so one or more pops must be inserted along one of the paths.)

The translation of Java's stack manipulation bytecodes is straightforward in most cases. However, there is one important exception caused by the lack of explicit type information in the bytecodes. Consider the `pop2` bytecode, which may be used either to pop two single-word values (such as integers or references) or one double-word value (a double or a long). In the former case, Pep would need to generate code that pops *two* values from the Self expression stack. In the latter case, it would need to generate code that pops *one* value off the Self expression stack since, in the translated code, a double is a single object. Thus, in order to translate a method that contains a `pop2` bytecode (or any of the `dup2` variants), Pep must perform Gosling's abstract interpretation to determine whether to generate code that pops one or two Self objects off the stack.

It is unfortunate that the explicit typing of Java's other bytecodes does not extend to the stack manipulation bytecodes. Had `pop2` and the `dup2` variants been reserved exclusively for popping and duplicating longs and doubles, no analysis or abstract interpretation of methods would ever be needed. Java bytecodes could be translated one by one without ever considering the method as a whole. Instead, as things stand, Pep must do abstract interpretation of methods containing `pop2` and `dup2` bytecodes to extract one bit of information for each such bytecode: does it pop (duplicate) one double-word or two single-words?

Although we have not done specific measurements to determine the cost of this abstract interpretation and the latency it incurs on program start-up, its complexity is comparable to that of the bytecode verifier [16]. Indeed, in a production system, it would be possible, and appropriate, to extract the necessary information about `pop2` bytecodes from the verifier itself.

5.3 Exception handling

In source terms, a Java exception handler has the following form:

```
try {
   … code that may cause an exception to occur …
} catch(java.lang.ArithmeticException exc) {
   … code that handles arithmetic exceptions …
}
```

The `try-catch` specifies a *guarded* range of statements (following `try`) and a handler (following `catch`). Exceptions are thrown implicitly by primitive operations or explicitly using the `throw` statement.

Javac compiles `try-catch` statements as follows. The code in the `try` part is compiled into a consecutive region of bytecodes, the *guarded region*. The bytecodes from the `catch` part are placed separately from the guarded region. Finally javac records the exception handler in the method's *exception table.* This entry ties together the guarded region and the catch entry point, as follows:

```
Exception table:
   from   to  target type
     2    31    34   <Class java.lang.ArithmeticException>
```

Each method has a (possibly empty) table of exception handlers. The above exception table specifies that if an exception of class `java.lang.ArithmeticException` or a subclass thereof occurs, the code starting at program counter 34 should be executed to handle the exception. The exception handler guards the range of bytecodes from program counter 2 to 31 and any methods invoked from these bytecodes. Outside the guarded region, the exception handler has no effect. The guarded regions of exception handlers nest properly: if one region is not completely contained in another, they must be disjoint.

Pep translates exception handlers by wrapping the guarded regions inside blocks. For example, the exception handler shown above is translated into this Self code:

```
[ … code resulting from translation of bytecodes 2-31 … ] tryCatch: [|:exc|
    exc is_java_lang_ArithmeticException
] With: [|:exc|
    … code handling the exception; translated from bytecode 34 on …
]
```

The code invokes the method `tryCatch:With:` on three blocks: first, the *guarded block* contains the guarded code; second, the *test block* determines if a given exception matches the type handled here; third, the *handler block* handles the exception. This translation essentially reverse-engineers the original structured source-level exception handler from the bytecodes. The `tryCatch:With:` method maintains a stack of active exception handlers. It will:

- push the test block and handler block on the exception handler stack,
- evaluate the guarded block, and
- at completion of the guarded block pop the test and handler blocks off the stack.

Now consider `throw`, which javac translates into the `athrow` bytecode. Pep translates this bytecode into a call to a method that searches the exception handler stack (from most to least recently installed blocks) for a test block that returns true when invoked on the thrown exception. If such a test block is found, the corresponding handler block is invoked on the exception object. In fact, it is guaranteed that some test block *will* return true since Pep installs a universal handler at the bottom of the stack. This handler catches any otherwise uncaught exception and rethrows it in the parent thread of the current thread in accordance with the Java language definition.

Important parameters that should guide the implementation of exception handling code are the frequencies of installing exception handlers, throwing exceptions, by-passing handlers, and handling exceptions. The passive exception handler tables used by the JVM (they are only consulted after an exception has been thrown) make installation free. This design choice favors the programming model in which exceptions occur infrequently, but preparing to handle them occurs frequently.

In comparison, the code Pep generates for exception handling does not achieve zero-cost installation of exception handlers. While it is possible for the Self compiler to inline-expand the call of `tryCatch:With:` and subsequently the call of the guarded block, the test and handler blocks must still be pushed onto and popped from the exception handler stack. Worse still, the Self compiler treats heap-stored blocks such as the test and handler blocks more conservatively, so when an exception occurs, it can be relatively expensive to invoke the test and handler blocks. To illustrate, we ran the program shown in Figure 9 on JDK 1.0.2 and Pep. The method `imain` measures the time for one million invocations of `method1` and `method2`, two identical methods, except that the latter contains an exception handler. This admittedly trivial program clearly demonstrates the weakness of Pep's exception handling. Table 7, shows that Pep can execute the first loop, which has no exception handler, more than 20 times faster than JDK 1.0.2. Although the exception handler in the second loop never needs to catch an exception, its mere presence slows down

Pep by a factor of 70, the net result being that Pep is now 3 times slower than JDK 1.0.2, which is unaffected by the presence of the exception handler.

```
class TimeExceptions {
  int j = 0;

  void method1() { j++; }
  void method2() { try { j++; } catch (ArithmeticException exc) {} }

  // Want instance method, so use main to call imain.
  public static void main(String args[]) { new TimeExceptions().imain(); }

  void imain() {
    long t1 = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) method1();
    long t2 = System.currentTimeMillis();
    System.out.println("method1: " + (t2 - t1) + "ms");

    long t3 = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) method2();
    long t4 = System.currentTimeMillis();
    System.out.println("method2: " + (t4 - t3) + "ms");
  }
}
```

Fig. 9. Program to demonstrate inefficiency of exception handlers in Pep.

	Pep	**JDK 1.0.2**
`method1`	120 ms	2,800 ms
`method2`	8,900 ms	2,800 ms

Table 7. Timing results from the program in Figure 9.

Pep was designed to translate Java exception handling into facilities already present in the Self language (i.e., blocks). When it was observed that exception handling incurred a significant overhead on some Java programs, we revisited this design choice. An extension of the Self virtual machine, which permits specification of Java-style "passive" exception regions, is currently underway. With this extension, Pep's exception handlers will be light-weight, avoiding, in particular, the creation of up to three blocks for each exception handler installed.

5.4 Synchronization

Threads may synchronize on any Java object by executing an explicit synchronization statement or by invoking a synchronized method. In the latter case, if the method is static (a class method) the synchronization takes place on the class itself. The two forms of synchronization differ in their manifestation in the bytecodes.

- *Synchronized statements* are implemented using two separate bytecodes, `monitorenter` and `monitorexit`. They take a reference to the object that they lock/unlock. It is an error for a thread to unlock an object it does not hold the lock for. Dynamic checks in both Pep and the Java interpreter will throw a monitor-state exception if this condition is violated.
- *Synchronized methods* have no manifestation in the bytecodes. Instead, the invocation bytecodes check to see if the called method is synchronized. If it is, the invocation is bracketed in `monitorenter` and `monitorexit` operations on the receiver object (or class, if the method is static).

Pep factors the synchronization into a separate object. Each object, whether it represents a Java instance or class, has a lazily allocated *lock* object that implements Java's synchronization operations. For example, to invoke `moni-`

torenter on an object, the lock is allocated (if necessary) and then acquired by sending it the monitorenter message.

Whenever a monitorenter bytecode occurs, correctness requires that the matching monitorexit bytecode must be executed, even if an exception should occur. For synchronized statements, Javac ensures this by following the lock acquisition with an exception handler that traps any exception, releases the lock, and re-throws the exception. For synchronized methods, which have no explicit locking in the bytecodes, Pep uses a similar re-propagating exception handler to clean up any locks. This exception handler is found in the sync_do: method, called by the wrapper for synchronized methods (see Figure 8 on page 10).

The inefficiencies in the current implementation of exception handling described in the previous section make synchronized method invocation too expensive in Pep. While we could avoid the exception handler by using Self's _OnNonlocalReturn: primitive, which traps all returns through an activation frame, unfortunately this primitive is currently even costlier than the exception handler.

To quantify the high cost of synchronization, we ran a simple program that executes 1 million calls of a regular and a synchronized method, each of which simply increments an integer instance variable; see Table 8. Synchronization slows down the Pep version by a factor of almost 80 making it slower than the JDK 1.0.2 version, which is also slowed down by synchronization, but "only" by a factor of 3. While this example is extreme, it is not unusual to find a high fraction of synchronized method calls even in single-threaded Java programs. Library routines, which must work both for multi-threaded and single-threaded Java programs, perform synchronization regardless of the number of threads in the actual program. We conclude that efficient Java implementations must have fast synchronization and hope to address this shortcoming of Pep in future work.

	Pep	**JDK 1.0.2**
regular	180 ms	2,800 ms
synchronized	13,900 ms	9,600 ms

Table 8. Time for loop doing one million method invocations.

5.5 Control flow

The Java language has structured control flow (if-then-else, while, do-repeat, switch), but javac compiles away the structure, emitting bytecodes that express control-flow using low-level jumps (unconditional, conditional, and indexed goto's). Self, as described in Section 2.1, builds control-flow structures using dynamic dispatch, blocks, the _Restart primitive, and non-local returns. In particular, Self has no facility for implementing arbitrary jumps directly, and since Self does no tail-call elimination, implementing a jump as a tail-call from one block to another is also not an option. Thus, Pep is in the unusual situation for a compiler to have to map a source language with unstructured control-flow (Java bytecodes) into a target language with structured control-flow (Self source).

We expected that mapping Java control-flow into Self control-flow would be one of the harder challenges in developing Pep. This prediction held true. The control-flow analysis complicates Pep and slows down the translation. Moreover, the Self code generated by Pep uses blocks intensely and is hard to optimize for the Self compiler, reducing the run-time performance of the Java programs. Measurements on several Java programs indicated that these problems were severe enough to justify adding branch bytecodes to the Self virtual machine. In turn, Pep was re-engineered and simplified to take advantage of the branch bytecodes. Sections 5.5.1 and 5.5.2 describe the pre-branch-bytecode and post-branch-bytecode versions of Pep. Section 5.5.3 concludes the control-flow discussion.

5.5.1 Control flow without branch support in Self

The first version of Pep was restricted to mapping Java bytecodes into the existing set of Self bytecodes. Rather than extending the Self bytecode set to accommodate the translation, we preferred to work harder in the translator to "make do" with pure Self. To map the unstructured control-flow of Java bytecodes into Self, Pep would do extensive control-flow analysis.

The control-flow problem that Pep solves resembles the problem addressed in the pioneering work by Baker on structuring Fortran programs. Baker transformed Fortran programs with gotos into programs with if-then-else and repeat

statements [1]. One difference between Baker's work and the present work is that upon encountering an irreducible flow graph, she could leave in an occasional goto. For Pep, this option does not exist: the target language has no gotos. On the other hand, Baker's algorithm deals with arbitrary control-flow graphs; Pep only needs to handle the graphs that javac produces from (structured) Java programs. Baker's approach, like recent work by Cifuentes [3] on decompilation, uses graph traversals and node numbering to compute the control flow structures. Pep uses graph transformations, similar to those of Lichtblau [15], the main difference being that Pep has an additional operation (splitting) to handle graphs that would otherwise be irreducible, e.g., graphs resulting from short-circuiting boolean expressions.

To recover structured control flow from a Java method, Pep performs these steps:

- partition the bytecodes into basic blocks and build a control-flow graph;
- find loops by looking for back-edges in the depth-first tree of the control-flow graph;
- for each loop, determine the *loop header*, the basic block that the back-edge points to, and the *loop body*, the set of basic blocks reachable by backtracking from the tail of the back-edge up to but not through the loop header;
- remove the back-edges from the control-flow graph;
- *reduce* the control-flow graph into structured control-flow, splitting basic blocks when necessary to meet structural restrictions;
- generate source code with structured control-flow from the reduced flow graph, adding the loop edges back in as necessary.

We illustrate these steps by means of an example, the method `isPrime` shown in Figure 2 on page 4. Javac compiles the method into the bytecodes shown in Figure 10. Before describing the translation in detail, let us preview the final result produced by Pep for these bytecodes: the Self method shown in Figure 11 (for readability, we reduced the name mangling slightly). Even a cursory look at this method reveals the structured control-flow and the intensive use of blocks.

```
 0 iload_1          15 iconst_3        32 iload_2
 1 iconst_2         16 istore_2        33 imul
 2 irem             17 goto 31         34 iload_1
 3 ifne 15          20 iload_1         35 if_icmplt 20
 6 iload_1          21 iload_2         38 aload_0
 7 iconst_2         22 irem            39 iload_1
 8 if_icmpeq 13     23 ifne 28         40 putfield #15 <Field Primes.maxPrime I>
11 iconst_0         26 iconst_0        43 iconst_1
12 ireturn          27 ireturn         44 ireturn
13 iconst_1         28 iinc 2 2
14 ireturn          31 iload_2
```

Fig. 10. The `isPrime` method translated into bytecodes (by javac).

To turn bytecodes into structured control-flow, Pep first finds the basic blocks and then builds the control-flow graph. Figure 12 shows the control-flow graph for `isPrime` (each box in the figure is a basic block; the numbers in the boxes denote program counter ranges). Next, Pep performs a depth-first traversal of the flow graph to identify loops (each basic block pointed to by a back edge is a loop header). The dashed edge in the figure is the loop edge for the single loop in `isPrime`.

Having built the flow graph, Pep temporarily removes the loop edges to make it acyclic. The acyclic graph is then *reduced* by repeatedly applying reductions on the graph until it consists of a single node. Intuitively, the reductions eliminate edges (branches) and as a side-effect build structured control-flow statements that will realize the control transfers represented by the edges. Pep iterates three different reductions: absorption, promotion, and splitting:

- *Absorption* moves a basic block into its predecessor and can only be applied to basic blocks with a single predecessor; see Figure 13.

```
isPrime_I: t_1 = ( | t_2 |
  (t_1 irem: 2) ifne ifTrue: [
      t_2: 3.
      [|:exit_0. :restart_0|
          ((t_2 imul: t_2) if_icmple: t_1) ifTrue: [
              (t_1 irem: t_2) ifne ifTrue: [
                  t_2: (2 iadd: t_2).
                  restart_0 value
              ].
              ^ 0
          ] False: exit_0
      ] loopExitContinue.
      self a_Primes_maxPrime_: t_1.
      ^ 1
  ].
  (t_1 if_icmpeq: 2) ifTrue: [^ 1].
  0
)
```

Fig. 11. Self method resulting from Pep-translating the `isPrime` bytecodes.

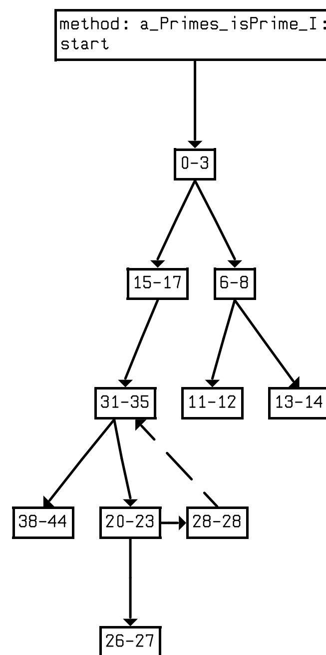


Fig. 12. Control-flow graph for the `isPrime` method.

- *Promotion* handles control-flow merges such as after if-then-else structures (but also cases with more than two predecessors) by lifting a basic block to the point that covers all its incoming branches; see Figure 14. Promotion is legal when the destination basic block and all its descendents have outgoing edges only to the promoted basic block.

Figure 15 shows that some situations offer a choice between promoting and absorbing. In the figure, the control-flow graph on the far left, which we say has a "one-way merge" into *D*, is first reduced using two absorption steps to the structure in the center. At this point, only the edge into *D* remains unreduced. It can be eliminated in three different ways: using an absorption or one of two promotions. In such situations, Pep favors promotions over absorptions and

opts for a “higher” promotion (in this case promoting *D* into *A*) over a “lower” promotion (promoting *D* into *B*). This way, Pep favors reductions that create less nesting in the resulting source code.

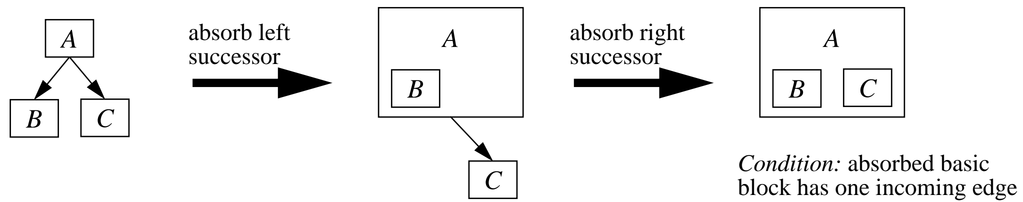


Fig. 13. Absorption creates nested structures (e.g. if-then-else).

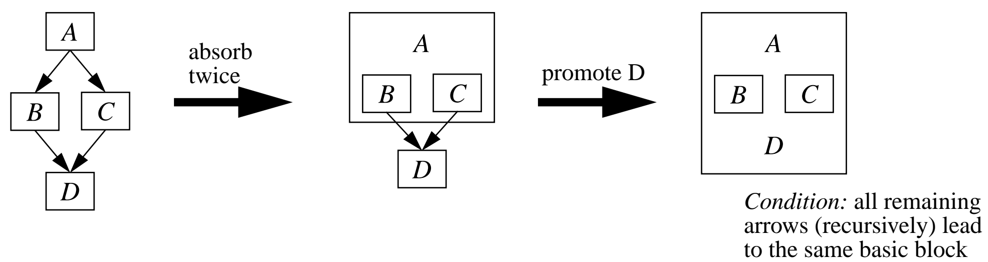


Fig. 14. Promotion (right half) handles control-flow merges.

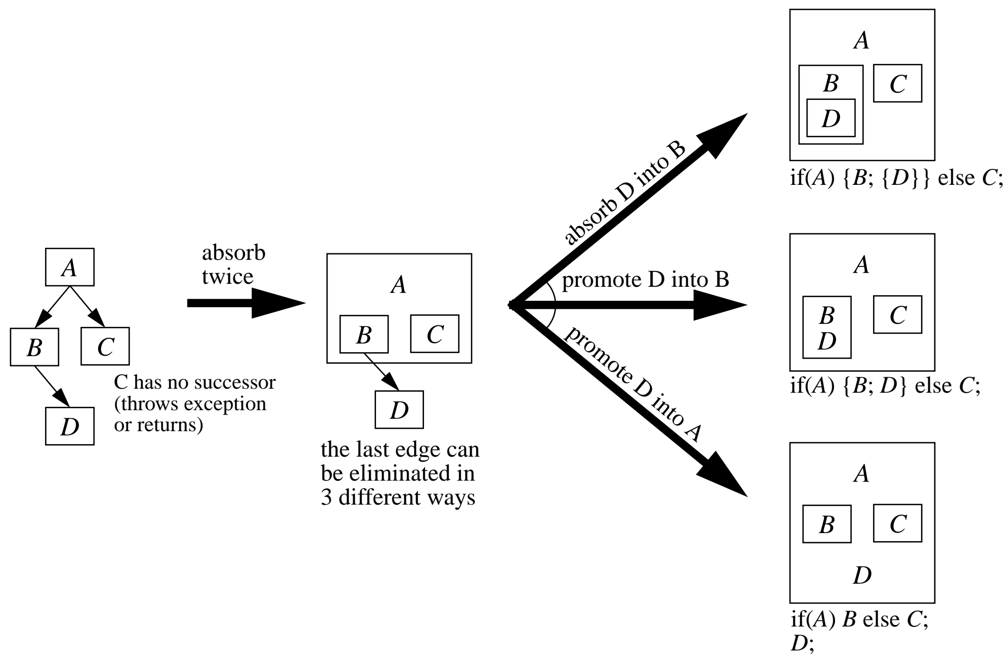


Fig. 15. Promotion also applies to “one-way” merges and gives less nesting than absorption.

There are situations when choosing a high promotion is incorrect because it may lift a basic block out of a loop that it should belong to. To avoid this problem, Pep precedes promotions by *structural checks*. It verifies that no ancestor on the path along which a basic block is being promoted is the header of a loop that should include the promoted basic

block. A similar structural check prevents a basic block, which is supposed to be the successor of a loop, from being pulled into the loop.

Some control-flow graphs cannot be reduced to a single node by absorption and promotion: we simply run out of legal moves. The left-hand side of Figure 16 shows an example of such a graph.

- *Splitting*, then, re-enables progress by copying a basic block and partitioning the incoming edges among the copies; in the right-hand side of Figure 16, the basic block 23-24 has been split. The copies will have fewer predecessors and therefore less structural constraints, allowing absorption or promotion. Indeed, after splitting 23-24, the flow graph in the figure can be fully reduced.

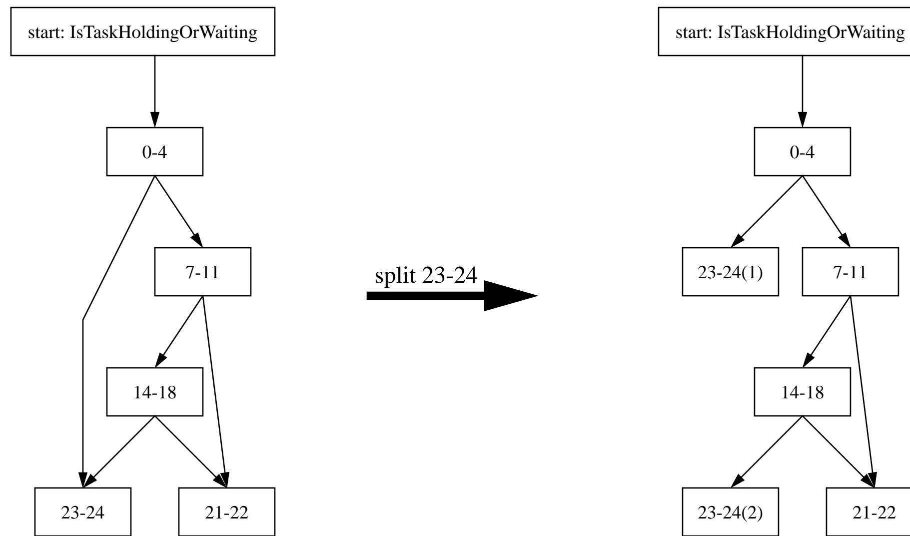


Fig. 16. A flow-graph whose reduction requires splitting.

Splitting simplifies the structure of flow-graphs to make them easier to reduce. The cost, however, is code expansion. To keep the code expansion in check, Pep applies two strategies. First, Pep only splits basic blocks when a situation has arisen that permits no absorption or promotion. In particular, the splitting illustrated in Figure 16 would not happen until several absorptions had been done. Second, when splitting must be done, Pep attempts to split less drastically than fully. For example, a basic block with four incoming edges need not be split into four copies; it may suffice to split it into two, where one copy receives three incoming edges that can be promoted away in a single step and the other copy receives the single “trouble maker” edge (which can now be absorbed away). These simple strategies keep the code expansion at less than 15% for most Java programs; e.g., for javac the code expansion is 11.5% (3,784 bytecodes out of 32,890).

The need for splitting arises because absorptions and promotions cannot cope with certain kinds of control-flow merges. Typically, javac generates such flow graphs from short-circuiting boolean expressions (the short-circuit edges can be troublesome) and from switch statements in which some cases have no `break` statements (the edges representing the fall-through to the next case can be troublesome). Indeed, the control-flow graph shown in Figure 16 resulted from this Java method with a short-circuiting boolean expression:

```
boolean IsTaskHoldingOrWaiting() {
  return IsTaskHolding() || !IsPacketPending() && IsTaskWaiting();
}
```

The two leaves in the pre-split graph return the possible results of the method, i.e., `true` or `false`.

The reduction process is guaranteed to terminate: each absorption or promotion will eliminate at least one edge from the graph, and splitting, although it introduces new nodes, does not increase the number of edges. The reduction, however, is *not* guaranteed to succeed: it may stop with more than a single node remaining. The following paragraph explains why.

The control-flow reduction process, as described so far, always succeeds. Exception handlers, however, complicate the reduction process by imposing extra constraints on the legality of reductions. Briefly, exception handlers specify consecutive regions of bytecodes. Such regions may span several basic blocks. When performing the control-flow reduction for a method with exception handlers, all basic blocks in a guarded region must remain together. When promoting or absorbing a basic block, care must be taken to avoid lifting it away from the other basic blocks in the exception handler regions that it belongs to. Likewise, we must avoid lifting a basic block into a structure where it will be guarded by too many exception handlers. These problems are similar to the structural constraints that loops impose on promotions but more complicated to deal with since exceptions are orthogonal to the control-flow in the method (they are simply specified as program counter ranges in the exception table). We have no guarantees that the control-flow reduction will always succeed under the additional constraints imposed by exceptions, although in the JDK 1.0.2 system, we have found no methods for which the control-flow analysis falls short, so such cases seem to occur rarely in practice. This situation could change in the future if exceptions become more used or if javac performs more optimizations that involve reorganizing bytecodes from several statements.

In summary, Pep's control-flow analysis is complicated, slows down translation, is brittle in the context of exceptions, and produces code that is harder to execute efficiently. For all these reasons, the second generation Pep was implemented.

5.5.2 Control flow with branch support in Self

The second version of Pep takes advantage of branch bytecodes added to the Self virtual machine to side-step the need to do control-flow analysis. These branch bytecodes were designed to be sufficiently expressive to cover all of Java's branches. There are four of them:

`branch L`	branch unconditionally to offset `L`
`branchIfFalse L`	branch if the object at the top of the expression stack is `false`
`branchIfTrue L`	branch if the object at the top of the expression stack is `true`
`branchIndexed` $[L_1, \ldots, L_n]$	branch to offset L_i where i is taken from the expression stack.

Space does not permit us to go into details about how these bytecodes were implemented in the Self virtual machine and their impact on the type-feedback-based optimizations performed by the Self compiler. Instead, we focus on how the branch bytecodes help the Java to Self bytecode translation performed by Pep.

With the four new Self branch bytecodes, Pep can translate Java bytecodes one by one. In particular, control-flow transfers can be translated directly by emitting appropriate Self branch bytecodes. For example, Pep can translate Java's `lookupswitch` and `tableswitch` bytecodes (sparse and dense switch statements, respectively) into `branchIndexed` bytecodes. Although the Self branch bytecode in this case supports only a dense multi-way branch, it is straightforward to compress sparse ranges (Pep uses a hash table or a linear search for compression, depending on the number of cases in the switch bytecodes).

To illustrate the translation with branch bytecodes, consider the first few bytecodes in `isPrime`. Pep translates these bytecodes in the order they occur in the method, for each Java bytecode emitting one or more equivalent Self bytecodes; see Figure 17. (Although Self bytecodes look different from Self source code, it is fair to think of them as the essence of Self. A different syntax could be chosen for source code, as long as the Self parser turns it into bytecodes like the ones in the figure.) In this particular case, each Java bytecode except the conditional branch was translated into one Self bytecode. Other Java bytecodes translate into somewhat longer sequences of Self bytecodes. In general, however, we can factor long sequences of bytecodes into a method and simply emit a call to the method instead of the sequence of bytecodes. With the inlining performed by the Self compiler, the method call does not incur a performance cost.

The directness of the translation has implications beyond the translation itself.

- The pre- and post-translation expression stacks are isomorphic. Given a program counter PC_{Java} in a Java method, let PC_{Self} denote the corresponding program counter in the translated Self method. Imagine executing the original Java method to the point PC_{Java} and the translated Self method to the point PC_{Self}. The directness of the translation ensures that the two method invocations will have expression stacks of the same height and contents (modulo the translation of the values).

Java: PC	bytecodes	comment	Self: PC	bytecodes	comment
0	`iload_1`	load the 1st local variable	0	`implicitSelfSend t_1`	Self accesses local variables this way
1	`iconst_2`	push integer constant 2	1	`literal 2`	push the constant 2
2	`irem`	compute remainder	2	`send irem:`	send msg. to compute remainder
3	`ifne 15`	jump to 15 if result non-zero	3	`send ifne`	returns true iff receiver is non-zero
			4	`branchIfTrue 13`	conditional jump to 13
6	`iload_1`	load the 1st local variable	5	`implicitSelfSend t_1`	load the local variable t_1

Fig. 17. When translating with branches, the mapping from Java to Self is very direct.

- Pre- and post-translation program counters correspond directly to each other. Thus, if the translated Self program is suspended, it is straightforward to find the corresponding point in the Java bytecodes.

In conjunction, these two properties make it possible to use Pep not only to execute Java programs on the Self system, but also to inspect and debug Java program executions. By customizing the Self debugger to show Java code instead of Self code, to present the expression stacks of processes in Java terms instead of Self terms, to undo the name-mangling, and to single-step at the granularity of Java bytecodes rather than Self bytecodes, the full power of the incremental Self programming environment becomes available for developing Java code.

The version of Pep that uses Self branch bytecodes can translate any Java method. However, the Self virtual machine currently does not implement a "trap on return" primitive that the branch bytecode generator uses to translate exceptions efficiently (see Section 5.3). Until this primitive is implemented, Pep falls back to using the original code generator, which does not take advantage of branches, when translating methods with exception handlers.

5.5.3 Discussion: branches or not

The translation without branch bytecodes has only one significant advantage: it requires no extensions to the Self virtual machine. This property was crucial in getting Java programs to run soon after we started work on Pep. However, it soon became clear that, in the long term, performance would not be good without direct support for branches in the Self virtual machine. For one, it makes little sense that Pep must work hard to turn low-level branches into structured control flow, just to have the Self compiler work hard to optimize away the structured control-flow in order to generate low-level unstructured machine code. While performance is fine when the Self compiler succeeds in completely eliminating all the blocks, if just one block remains un-inlined in a loop, performance often degrades significantly. In addition, many Java programs, including Javac, make heavy use of switch statements. Switch statements incur a particular penalty on the no-branch Self virtual machine: when there is no efficient way to perform a multi-way branch one must resort to using a sequence of conditional statements. (True, a multi-way branch *could* be implemented using dynamic message dispatch, but this approach would require explicit management of local environments, since Self's lexical nesting applies to blocks only.)

The difficulties of mapping unstructured (goto-based) control-flow into Self's control-flow could have been avoided by working instead from Java source, which *is* structured. As mentioned in Section 3, we considered this option, but rejected it to remain compatible with the standard Java execution model. Other intermediate code formats than Java's bytecodes, however, retain the structure of the source program. For example, Clarity MCode has `BeginLoop` and `EndLoop` bytecodes [14], and Oberon's "Slim Binaries," which can be executed on the Juice virtual machine, consist of compressed abstract syntax trees [7, 8].

Given the unstructured control-flow in Java's bytecode format, at this point it is clear that the many advantages of adding branch bytecodes to the Self virtual machine outweigh the one disadvantage (the high implementation effort). Table 9 summarizes the impact of branches on Pep.

6 Reducing translation overhead

When compiling dynamically, each cycle spent compiling is one less cycle available for execution of the application. The Self system, in which advanced optimizing compilation has been explored, employs a two-compiler approach to

	without Self branches	with Self branches
speed of Pep	sluggish (involves many steps)	fast; very direct mapping
speed of Self compiler on generated code	not fast: must optimize code that uses blocks in complicated ways	potentially faster
efficiency of generated code	brittle: excellent if Self compiler inlines away *all* blocks, inefficient otherwise	predictable
robustness	vulnerable to future changes in javac because exceptions constrain reduction operations	highly robust due to directness of translation
implementation effort, Pep	high	low
implementation effort, Self virtual machine	zero	high: had to add branch support to both Self compilers

Table 9. Control-flow without and with Self branch bytecodes.

balance compilation and execution while ensuring both quick start-up and fast execution. Initially, code is compiled using a fast non-optimizing compiler. The machine code generated by the non-optimizing compiler is instrumented to identify the time-consuming parts of the application, and to collect information required for optimization. The optimizing compiler selectively recompiles and optimizes the hot spots as they are identified [12].

The initial version of Pep did not have low translation overhead as a specific goal. Nevertheless, as Pep kept evolving, some effort was taken to make Pep translation less intrusive. First, laziness was introduced to postpone translation as long as possible to reduce start-up time of Java applications. Second, a faster *binary* code generator was added. We describe the use of laziness in Section 6.1 and the source vs. binary code generation issue in Section 6.2.

6.1 Laziness: improving start-up time

Pep employs laziness at two levels to reduce start-up time of applications: classes are never loaded until they are needed and, within classes, methods are generally left untranslated until their first invocation.

Lazy class loading. In Pep, three circumstances establish the need to load a class:

- an instance of the class must be created,
- an array of (array of …) the class must be created, and
- a subclass of the class must be loaded.

Thus, when a program attempts to create an instance of a class, the class, its superclass, the superclass's superclass and so on all the way to class `java.lang.Object` will be loaded (unless they have been previously loaded, of course). For example, if a piece of code executes `new Primes()`, the class `Primes` will be loaded but no further classes will be (since the superclass of `Primes`, `java.lang.Object`, has already been loaded).

While it would be possible, with various degrees of effort, to relax the three circumstances that establish the need to load a class, doing so incurs additional complexity in Pep. For example, currently the ability to create arrays of a class is implemented with methods defined on the class itself. Defining these methods in a different object than the class, would allow Pep to create an array of a class without loading the class (since array elements initially are `null`). In our development of Pep, we have found that the above three circumstances strike a good balance between laziness (i.e., limiting the number of classes being loaded) and implementation complexity.

Pep implements the lazy loading of classes using Self's ability to catch message-not-understood errors. Pep-translated Java code has access to an object, `java_classes`, which contains all the classes that have been loaded by Pep[2]. A class is accessed by using the mangled name of the class as the selector of a message to the `java_classes` object. For example, javac translates the expression `new Primes()` into the Java bytecode

```
new <class Primes>
```

2. To be precise: the object `java_classes` actually holds none of the loaded classes. Instead, it *inherits* from a separate object, `loadedClasses`, which contains all the loaded classes. We have used two objects in order to separate the behavior of on-demand class loading from the organization of the loaded classes.

which Pep then translates into the Self expression

```
java_classes class_Primes new.
```

The above expression first sends the message `java_classes` to obtain a reference to the object holding the loaded Java classes. Then the code sends the message `class_Primes`, obtained by mangling the class name `Primes`, to get the class object. At this point one of two things may happen:

- the `Primes` class has already been loaded, in which case the message simply returns the class object, or
- the `Primes` class has not been loaded, in which case there is no slot in the `java_classes` object that matches the message `class_Primes`. Consequently, a message-not-understood error happens, which the `java_classes` object traps. The trap code inspects the failing message's selector, `class_Primes`, unmangles it to obtain the name of the requested class, `Primes`, and invokes the class loader to dynamically load, translate, and install this class so that subsequent accesses to it run at full speed.

Thus, in either case, the expression `java_classes class_Primes` returns the class object for the class `Primes`. Finally, the class object is sent `new`, to create an instance of the class (by cloning the prototype object).

Figure 18 shows a screen snapshot of the code involved in the lazy class translation. The front-most object defines a method with the name

```
undefinedSelector:Type:Delegatee:MethodHolder:Arguments:
```

to trap message-not-understood errors. The method simply invokes Pep's loader to fetch and translate the requested class. The object in the background, a parent of the foreground object, contains the loaded classes' class objects. Among them is the class `Primes`, and several other classes required to execute the `Primes` program (most of the other classes define input/output operations).

Lazy method translation. Even when classes are loaded on demand, translation overhead still delays application start-up. The most time-consuming part of translating a Java class is translating the bytecodes defining method behavior into Self expressions. By deferring this step as long as possible (i.e., until each method is invoked), start-up time can be improved significantly.

Even though lazy translation of methods is generally a win, laziness does incur a small additional overhead over eager translation in the case where the method *will* be invoked. For example, class initializer methods `<clinit>`, must by definition be executed before we can use the classes. Accordingly, Pep never defers translation of class initializer methods. Similarly, for very short methods, which can be translated quickly, Pep does not employ lazy translation. For large methods, on the other hand, the translation time itself dwarfs the overhead of setting up lazy translation. The threshold above which Pep deploys lazy translation is an adjustable parameter. The optimal threshold depends on translation speed and the likelihood of a method being needed. For the slower source code generator, we have found a limit of 15 bytecodes to provide good results; for the faster binary code generator, the limit can be raised to 50 bytecodes.

Lazy translation of methods can be implemented easily in the dynamic Self environment. When translating a class with a method that is a candidate for lazy translation, instead of translating the bytecodes into Self code, Pep builds a stub method. Pep installs the stub in the class object under the same name as the method would have been installed if lazy translation had not been employed. Hence, when a client of the class later tries to invoke the method, the stub will run instead. The stub contains the information needed to translate the Java bytecodes into Self code, replace itself with the resulting Self method, and finally invoke the method. To the client, the presence of the stub is invisible, except for timing effects. Figure 19 shows an actual stub in the `Primes` class: the stub for the method `isPrime` (i.e., the figure shows the appearance of the class *before* the `Primes` program has executed, since the `isPrime` method *will* be executed when the `Primes` program runs). The object `code_attrib` defined locally in the stub, performs the actual translation ("`0 _AsObject`" is a name that the Self user interface has made up for the object, which would otherwise have no name). The last statement in the stub invokes the method resulting from the lazy translation, passing in the argument `t_1`, the number being tested for primality.

Lazy translation of methods not only improves application start-up, but can also reduce total execution time, since, for a given application, many of the methods in the classes it uses may *never* be invoked. For example, consider javac, a moderately large Java program. When invoking the JDK 1.0.2 version of javac to compile Richards.java, a version

Fig. 18. Lazy class loading implemented by catching message-not-understood errors.

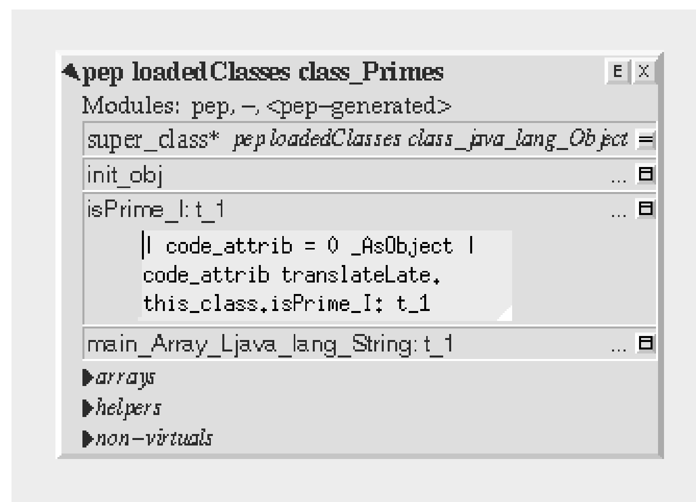


Fig. 19. Stubs implement lazy translation while shielding callers.

of the Richards program, Pep loads 155 classes with a total of 1392 methods. If the lazy translation threshold is 15 bytecodes, 1033 methods are translated while 359 remain as stubs. Table 10 gives numbers for other threshold values.

lazy threshold, #bytecodes	#methods translated	#methods remain as stub
0	810	582
15	1033	359
50	1256	136
∞	1392	0

Table 10. Effect of lazy method translation for different size thresholds (measured on javac).

As mentioned above, the Self system mixes fast compilation and optimizing compilation to maximize application progress. While Pep itself does not perform optimizations in the translation of Java bytecodes to Self bytecodes (the Self compiler does, of course, optimize when compiling the Self bytecodes), an analogous technique could still be used in Pep to maximize application progress. To illustrate, consider class initializer methods. Each Java class defines an initializer method, named `<clinit>`. This method executes exactly once, when the class is loaded. By adding an interpreter to Pep, class initializers could be interpreted rather than translated to Self code and then executed. Interpretation of Java bytecodes will likely be as fast as, or faster than, the translation into Self code, and, since class initializers execute only once, the result will be an overall speed-up of the application. The use of an interpreter could be taken one step further: translation of any method could be deferred until the method has been executed a few times. Exploring this venue, however, remains future work.

6.2 Source code generation vs. binary code generation

We have discussed laziness as a way to defer and reduce translation overhead. The output of the translator can be either source code or binary code. This choice affects raw translation speed significantly. Figure 20 illustrates the difference. By generating binary bytecodes, which in itself is faster than generating source code, the Self parser can be skipped. (The Self parser is part of the Self virtual machine, but the virtual machine also provides means to bypass it.)

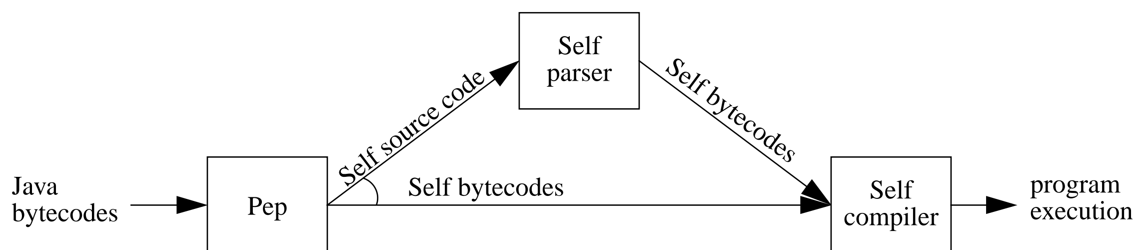


Fig. 20. Binary code generation (bytecodes) allows Pep to bypass the Self parser.

The initial version of Pep generated source code. This choice was made to facilitate debugging: with source code, the Self environment can be used to inspect (and even patch) the generated code. When Pep was redesigned to take advantage of Self's new branch bytecodes, Pep had been exercised enough that we could safely go to a binary code generator. Furthermore, since the use of branch bytecodes eliminates the need for the expensive control-flow analysis, the savings from switching to binary code generation are all the more visible. Thus, the two versions of Pep that exist today couple the choice between source vs. binary and branches vs. no branches: (no branch, source) and (branch, binary). Decoupling the choices would be an interesting addition to implement, since it would give us the ability to isolate the effects of the two choices.

We have refrained from performing measurements of the translation speed of the two versions of Pep. For one, translation speed was never an important concern in the implementation of either version. Moreover, the primitives that the two versions invoke in the Self virtual machine to build and install the generated methods were never tuned for raw performance. For example, turning a vector of bytecodes into an executable method, as is done repeatedly when using the binary code generator, currently takes time proportional to the product of the number of bytecodes and local variables (i.e., quadratic time in the worst case). While this performance deficiency is insignificant in the interactive Self programming environment where a programmer *edits* methods one at a time, the performance of the primitives matters a lot for a translator like Pep, which *generates* a large number of methods in a short time.

7 Performance measurements

In this section we present performance numbers for Java programs executed on Pep. To put the numbers into perspective, we compare runtimes of the programs when executed on Pep and the JDK 1.0.2 Java virtual machine (the Java system current at the time when Pep was developed). We also report numbers for the recently released JDK 1.1 system. Some precautions must be made, before we give specific numbers.

- *Ongoing Pep and Self virtual machine improvements.* Although the Self virtual machine has reached maturity for executing Self code, only recently did we start feeding it translated Java programs. In particular, the branch bytecodes were added to the optimizing compiler only recently, so much tuning still remains to be done.
- *Class library differences*. We were able to execute our benchmarks on both the JDK 1.0.2 and JDK 1.1 class libraries. However, even if the class libraries provide the same functionality, differences in their implementation could still cause the actual bytecodes executed for a given benchmark to differ significantly depending on whether it is executed with one or the other class library (e.g., as class libraries mature, improved exception testing and additional synchronization tend to make the code slower while algorithmic improvements could introduce the opposite trend).

Ideally, we would have liked to keep the class libraries constant while varying the execution engine. Unfortunately, this was not possible. For example, even a trivial application like HelloWorld cannot be executed on the JDK 1.1 interpreter with the 1.0.2 class library (the result is a core dump). Likewise, the Java native methods in Pep's runtime system match the 1.0.2 class library, but cannot support the new 1.1 class library. Thus, to avoid comparing apples to oranges, we primarily compare Pep and the JDK 1.0.2 interpreter. As a supplement, we include JDK 1.1 numbers, i.e., measured on the 1.1 interpreter with the 1.1 library.

We present measurements for Pep both with and without Self branch bytecodes. By now, the latter set of numbers is somewhat academic, since any practical Java system based on the ideas in Pep and the Self virtual machine would certainly implement branches directly.

Java, like C++, is a hybrid language in which primitive types such as integers are not objects, i.e., they cannot have virtual methods, and in which methods can be non-virtual or virtual. Therefore, Java permits a spectrum of programming styles ranging from C-like, with most calls statically bound, to Smalltalk-like, with most calls virtual or through interfaces. Accordingly, when benchmarking Java, it is important to understand the implications of the programming style. Aiming directly at this critical issue, Mario Wolczko wrote six increasingly object-oriented versions of the Richards benchmark, an operating systems simulator; see Table 11. Richards, originally written in BCPL by Martin Richards and later used by L. Peter Deutsch to benchmark Smalltalk, at 500 lines is sufficiently large to be non-trivial yet small enough that it can feasibly be ported to a number of languages and programming styles. Briefly, from Richards1 to Richards6 objects become finer-grained (for higher reusability), and virtual calls more frequent (for higher customizability).

version	**main characteristic (change from previous version)**
Richards1	Translated to Java by Jon Gibbons from original BCPL source. The least object-oriented version. Encodes multiple booleans as integer, switching on the integer to distinguish state of booleans.
Richards2	Uses direct representation of booleans rather than the encoding. No switch statement.
Richards3	Moves some task state into separate objects like in Peter Deutsch's Smalltalk version.
Richards4	Accesses object state via non-virtual (final) accessor methods.
Richards5	Changes accessor methods to be virtual. Comparable to Deutsch's Smalltalk version of Richards.
Richards6	Uses interface types, typical of frameworks that operates on independently developed classes.

Table 11. Mario Wolczko's six versions of Richards.

In addition to the Richards versions, we included single versions of other programs in our benchmark suite; see Table 12. We chose Javac and javadoc because every Java programmer knows them, they solve real and non-trivial problems, and they require no interaction. RayTracer, provided by Jeff Chan, has been used to characterize the performance of the picoJava™ CPU architecture [18]. Wolczko's Java version of DeltaBlue was included because, like Richards, it has an established track record as a benchmark for many object-oriented systems. Richards and Deltablue, although written specifically for benchmarking, are *not* micro-benchmarks that stress selected language features

or answer questions such as “how many cycles does an array store operation take?” Rather, they aim to be well-rounded object-oriented programs.

For each benchmark, Table 12 lists the approximate static size as the number of source lines and the dynamic size as the number of bytecodes executed. In addition, since our main interest is to understand performance issues related to object-orientation, the table reports the frequency of virtual calls and, for comparison, non-virtual calls. Here, the Richards versions, as expected, show increasing frequencies of virtual calls. Perhaps surprisingly, even Richards4, which uses `final` accessor methods, executes many virtual calls. The reason is that although javac *could* statically bind and inline away the accessor method calls, it only does so when invoked with the “-O” flag. For the present study, we ruled out using “-O” because the inlining that javac then performs often produces illegal bytecodes. For example, inlining a method that accesses a private field of its class is *not* correct when the caller resides in another class (this problem has now been corrected by restricting inlining in the JDK 1.1 version of javac).

			dynamic counts (using 1.0.2 class library)		
program	**description**	**source lines**	**bytecodes**	**virtual calls, incl. interface**	**non-virtual calls**
javac, 1.0.2	compile Richards	~20,000	6,997,379	3.91%	1.35%
javadoc, 1.0.2	gen. html documentation of class	~4,000	9,557,012	3.97%	1.13%
RayTracer	render 200x200 picture [18]	3,546	1,999,281,260	12.6%	0.70%
DeltaBlue	2-way constraint solver [20]	1,127	856,625	7.10%	1.12%
Richards1	operating system simulator: six different versions implementing the same functionality but increasingly using the object-oriented features of Java	407	5,145,123	2.96%	0.65%
Richards2		389	6,012,841	3.62%	0.56%
Richards3		453	7,188,798	5.91%	0.79%
Richards4		518	11,247,285	14.2%	1.38%
Richards5		517	11,247,179	14.2%	1.38%
Richards6		552	11,247,267	14.2%	1.38%
Linpack	java version of Linpack [6]	573	9,711,971	0.06%	0.06%
LinpackOpt	hand-optimized version [11]	585	8,026,293	0.13%	0.00%

Table 12. Characterization of benchmark programs.

In stark contrast to the other benchmarks, the Linpack benchmarks, shown last in Table 12, exhibit extremely low call frequencies. These Fortran-derived benchmarks, while stressing Java’s floating point performance, are poor benchmarks if one is interested in object-oriented features such as virtual calls. We should mention that, in fairness to the Java interpreter, we changed the Linpack programs to compute with floats instead of doubles, since Pep does so anyway.

Table 13 shows the raw execution speed of the benchmarks. To reduce statistical variation due to caching, garbage collection, and other uncontrollable factors, each benchmark was iterated 200 times, discarding the first 100 iterations and reporting the average execution time of the following 100 iterations. For the long-running RayTracer we scaled back to 2 warm-up runs followed by 5 measurement runs. The use of multiple runs, in addition to reducing noise, allows the dynamically optimizing Self system to reach steady-state performance.

Compare first the numbers for Pep with and without branches. For most benchmarks, the performance is quite similar. The general tendency, although a weak one, is that branch bytecodes speed up benchmarks that use switch statements intensely (javac and Richards1), but slow down other benchmarks slightly. The almost identical geometric means confirm that branches do not improve the overall performance of this set of benchmarks. This observation, of course, does not diminish the important simplifications of Pep enabled by the branch bytecodes, nor does it rule out future performance gains as the Self compiler is tuned to optimize code with branches more aggressively.

Now compare the 1.0.2 interpreter’s performance against that of Pep. The measurements partition the benchmarks into three distinct groups:

- javac, javadoc, and DeltaBlue. Pep runs javac slightly slower than the interpreter and javadoc, and DeltaBlue about as fast as the interpreter. Preliminary studies indicate that a combination of factors are responsible for the lack of speed-up despite the Self compiler’s type-feedback optimizations. First, javac performs many operations on integer values of magnitude larger than the 30 bits that Self can represent directly. Recall from Section 5.1 that javac ran an order of magnitude faster when we changed Pep from using bigInts to using boxed machine integers.

	JDK 1.0.2 interpreter	JDK 1.1 interpreter	Pep, no branches	Pep, branches
javac, version 1.0.2	7,950	3,700	9,230	8,900
javadoc, version 1.0.2	6,160	2,900	6,590	5,800
RayTracer	570,000	191,000	208,000	218,000
DeltaBlue	520	210	530	520
Richards1	1,050	540	380	320
Richards2	1,250	650	170	200
Richards3	1,510	750	210	210
Richards4	2,900	790	230	250
Richards5	2,880	1,450	230	250
Richards6	2,870	1,710	210	270
Linpack	1,460	820	960	970
LinpackOpt	1,230	790	970	910
geometric mean	3,213	1,514	995	1,012

Table 13. Runtime per iteration (real-time milliseconds on a 167 MHz UltraSPARC 1).

The boxed integers still incur overhead, explaining in part why javac executes slower on Pep. Second, for tight loops, likely to appear in the lexers of both javac and javadoc, the backend of the Self compiler falls short: the major design goal of the Self compiler was optimization of virtual calls whereas little effort was spent on ensuring good local code quality through traditional back-end techniques such as delay slot filling, register allocation, avoidance of redundant loads, and common subexpression elimination [12]. We also suspect backend issues explain the lack of speed-up of DeltaBlue, but we have yet to confirm this hypothesis.

- RayTracer and all Richards versions. For these call-intensive programs, Pep is significantly faster than the interpreter by a factor of 2.6 (RayTracer) to 13 (Richards5). On these programs, the Self compiler's ability to inline method calls, whether virtual or statically bound, delivers Pep a clear performance win. Moreover, and no less important, Pep attains almost constant performance for all Richards versions. This stands in sharp contrast to both versions of the interpreter for which performance degrades by almost a factor of 3 (from Richards1 to Richards6). Figure 21 visualizes this pattern by plotting the performance numbers normalized to the time of Richards2 executed on Pep without branches (the fastest version). To make the picture clearer, we connected the data points with lines, although, of course, the dataset is discrete. This result is extremely important. Pep, unlike the interpreters and all other Java implementations that we are aware of, does not penalize good object-oriented programming style with bad performance. Promoters of Java as a high-level, safe, object-oriented language should strive to attain this property, as it is the only way to prevent the prevailing Java programming style from turning C-like for performance reasons.
- Linpack and LinpackOpt. Pep executes these non-object-oriented benchmarks only 30-40% faster than the interpreter. We attribute this somewhat disappointing performance mainly to the backend of the Self compiler and the tagged floating point numbers. For Linpack-like code, even a simple dynamic compiler with a strong backend should be capable of obtaining C-like performance. Compared with JDK 1.0.2, both Pep and JDK 1.1 significantly narrows the gap between Linpack and LinpackOpt, indicating that many of the optimizations that Hardwick performed by hand would be superfluous, had he been using a stronger Java implementation than JDK 1.0.2.

In summary, although the benchmarks point to deficiencies in the backend of the Self compiler, on the object-oriented and call-intensive benchmarks, Pep, helped by the Self compiler's type feedback, sustains constant and high performance even when object-orientation is taken to the extreme. Pep demonstrates clearly that type feedback can effectively optimize heavily object-oriented Java programs.

Finally, consider the numbers for the JDK 1.1 interpreter in Table 13. Recall that in this case the benchmarks have been executed against a different class library so the comparison must be taken with a grain of salt. JDK 1.1 is between 1.6x (LinPackOpt) and 3.0x (RayTracer) faster than JDK 1.0.2. The geometric means indicate that JDK 1.1 is about 2x faster than JDK 1.0.2 and that Pep, in turn, is about 1.5x faster than the JDK 1.1. However, this is one case in which the mean values do not reveal the whole truth: for javac and javadoc, JDK 1.1 is significantly faster than Pep.

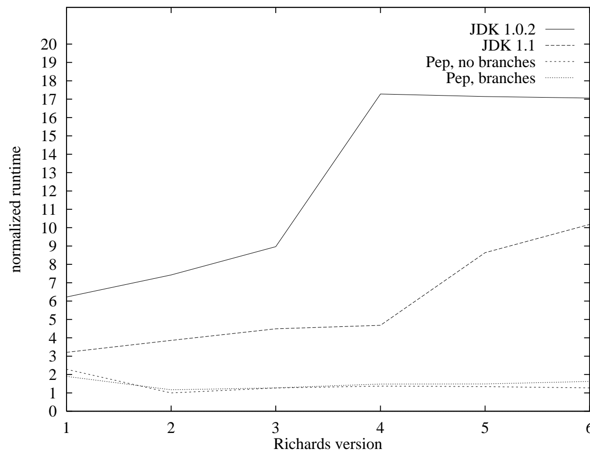


Fig. 21. Runtime of Richards normalized to time of fastest version.

Looking once more at Figure 21, the curves for JDK 1.0.2 and JDK 1.1 are similar in shape, but with one difference: JDK 1.0.2 slows down dramatically at Richards4 whereas JDK 1.1 manages to delay the slow-down until Richards5. The reason is that JDK 1.1, on a small scale, performs a dynamic optimization not unlike those of the Self compiler. When certain conditions are met, JDK 1.1 can dynamically inline methods to eliminate the calling overhead: the inlined method must be at most three bytecodes long to fit in the space that used to occupy the call and must be final (non-virtual). Richards4 has a lot of final accessor methods, so JDK 1.1's inliner helps significantly on this benchmark. Richards5, however, changes the accessor methods to be virtual so they can no longer be inlined by this simple inliner. (The Self compiler can still do so, as the flat performance curves for Pep demonstrate.)

8 Current status

Currently, Pep supports the full Java language as of the JDK 1.0.2 release with the following exceptions:

- floating point values are restricted to the 30 bits provided by Self,
- finalization and weak pointers are missing (the Self virtual machine provides no such facilities), and
- the native method interface provided by Pep is non-standard.

The latter point deserves some explanation. Native methods are written in some language other than Java (typically C) and translated into the host computers machine instructions. They provide access to services defined by the operating system, windowing system, network, and so on. Native methods can be passed references to Java objects, so that they can read and write fields in these objects. Since Pep uses a different object layout than do other Java implementations (for one, the objects are really Self objects and as such have tagged integers and references), most native code does not work with Pep. It is generally accepted that the 1.0.2 native interface, although efficient because of the direct exchange of pointers, is too unrestricted. To achieve better isolation of Java and native code, alternative interfaces, such as JNI [13], have recently been proposed. We expect that Pep can support such an interface.

Meanwhile, we must re-implement native methods specifically for Pep. Typically, we use Self code to do so. When translating a class containing a native method, Pep consults a special object that contains definitions for native methods. This object, partially shown in Figure 22, currently contains some 120 native methods. If the native method object contains a method with a matching name, the method will be copied into the generated class object. If Pep finds no definition of the native method, it generates one on the fly that simply reports an error if it is executed: "missing native method."

Having described the missing parts, let us conclude by looking briefly at what Pep *can* do. Pep executes Java programs that:

- use threads; Java threads are mapped to Self threads, which have been extended with priorities.

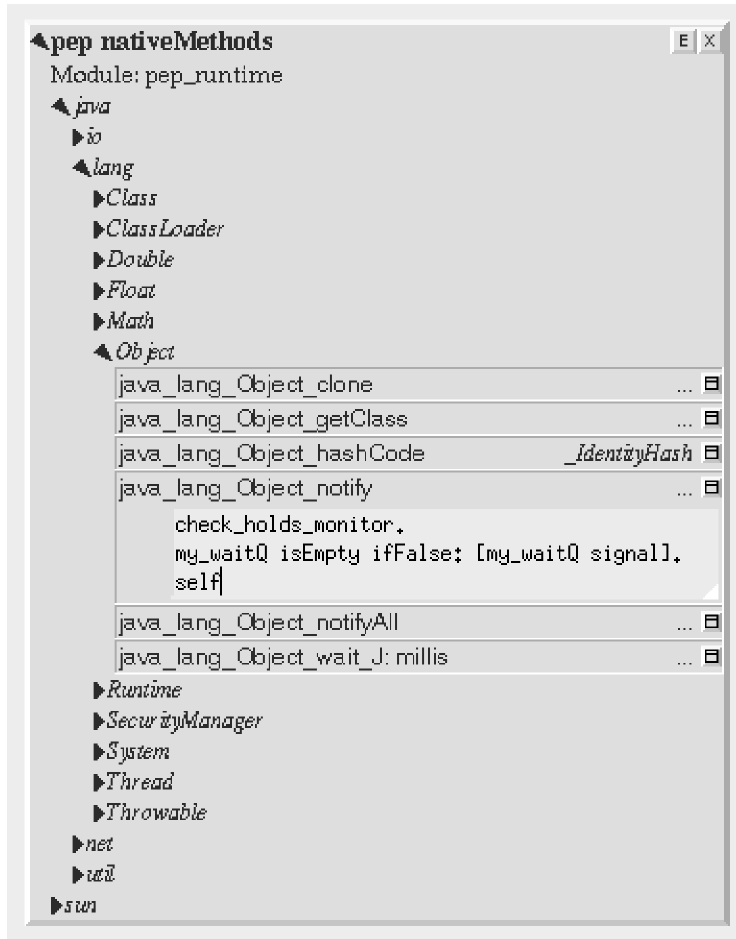


Fig. 22. Pep uses Self code to implement native methods for Java.

- perform non-blocking input/output (the thread doing IO blocks, other threads keep running).
- use sockets to communicate over networks; for example, Pep can run Javasoft's Jeeves web server.
- use graphical user interfaces, including applets; we ported the "Tiny" version of the AWT library to Pep.

As Pep is subjected to more programs, undoubtedly the need for more native routines will appear. Still, we can confidently say that Pep can execute many and non-trivial Java programs.

9 Conclusions

Java, the latest member of the C family of languages, became popular in a short time. Its strict enforcement of type-safety, requirement for garbage collection, elimination of unsafe pointer arithmetic, and simplifications over C++ all contributed to the popularity. The single most important factor in Java's success, perhaps, is portability: the language is completely specified, program behavior does not depend on word size or other aspects of the host computer, and binary programs can be exchanged seamlessly over computer networks regardless of the host architectures. Portability, achieved by compiling programs into machine-independent bytecodes for a virtual machine rather than native code for an actual processor, slows down programs if the bytecodes are interpreted. To regain performance without sacrificing portability, Java implementations can compile bytecodes into native code on-the-fly.

We built Pep to study the effectiveness of the Self compiler's type-feedback based optimization on Java code without having to reimplement the optimizer from scratch. Pep, in itself a just-in-time compiler, translates Java bytecodes into

Self bytecodes that can subsequently be executed on the Self virtual machine benefiting from the optimizations performed by Self's just-in-time compiler.

In this paper we have described the design and implementation of Pep. Several aspects of Java's bytecodes were particularly challenging and led us to reconsider earlier design decisions:

- *Control flow.* Java bytecodes are not a loss-less translation of Java source code: control-flow, structured in Java source code, has been reduced to low-level branches in the bytecodes. The initial version of Pep worked hard to reconstruct structured control-flow from the bytecodes, but in the process produced code that was hard for the Self compiler to optimize. In response to observing this deficiency, David Ungar added branch bytecodes to the Self virtual machine. The second version of Pep, taking advantage of the new branch bytecodes, has a simpler translation process because the need for control-flow analysis has been eliminated, and produces code that potentially can execute more efficiently (although presently does not) because the complicated use of blocks is avoided.
- *Exception handling*. Java exception handlers, currently used moderately often, but likely to become more pervasive as the Java code base matures and robustness demands increase, should be very cheap to install and relatively cheap to activate. Pep's current translation scheme, which relies on blocks, should be replaced with a more efficient one.
- *Primitive types*. Java specifies full 32 and 64 bit integers and floats; Self uses tagging and therefore provides only 30 bit integers and floats. Our results indicate that an efficient Java implementation cannot afford to use boxed integers, not even for just the values that exceed 30 bits.
- *Synchronization* must be efficient, even when executing single-threaded programs, since many library method calls are synchronized. Pep's current synchronization mechanism should be improved.

We also observed general properties of just-in-time compilers. Since compilation happens at run-time, each cycle spent compiling is one less cycle available for execution. Pep addresses two specific concerns:

- *Reducing translation overhead.* The first version of Pep generates source code and performs extensive control-flow analysis. The second version of Pep generates binary code and performs little control-flow analysis. Unfortunately, the lack of explicit type information in the `pop2` bytecode, which does not affect an interpreter, necessitates abstract interpretation of methods containing this and similar bytecodes. We consider this a flaw of Java's bytecode set and recommend a redesign to support fast just-in-time compilers.
- *Reducing start-up time.* While total translation overhead is important, fast program start-up is even more so in interactive systems. Pep employs lazy class loading and lazy method translation, two techniques that can benefit any just-in-time compiler. Two other techniques to reduce start-up time were considered: using an interpreter for rarely executed methods and, like the Self system, distinguishing between fast compilation and optimizing compilation.

Pep, although from the outset an experimental system, executes most Java programs, even large programs using threads and graphics. Moreover, the directness of the binary translator enables the use of Pep and the Self system, not just as a Java execution engine but as a full program development environment: stacks and program counters correlate so directly across the translation that the executing Self program can be easily inspected in terms of the original Java source code. In future work we hope to use Pep to study the behavior of Java programs in a broader sense. The prospects are promising because Pep executes in a more flexible environment (Self) than the typical Java interpreter written in C.

Pep executes Java programs faster than the JDK 1.0.2 interpreter, but not always faster than the improved 1.1 interpreter. Several factors, specific to Pep and the Self system but not inherent to dynamic optimization, impaired the performance of Pep: slow boxed integers, slow exception handling, slow synchronization, and lack of traditional backend optimizations in the Self compiler. Nonetheless, a very encouraging result came from observing the relative performance of Pep and the Java interpreters on the six increasingly object-oriented versions of the Richards benchmark. Pep, in contrast to the interpreters, attains *nearly constant* performance as Richards is pushed from a C-like programming style to a highly object-oriented, extensible programming style. This constant performance, unique to Pep among the Java implementations that we know of, could be crucial to Java's future success: only an implementation with this property can steer Java away from the path where a fully object-oriented programming style is sacrificed for performance—the path that poisoned C++.

Acknowledgments. David Ungar and Mario Wolczko did essential work on the Self VM, co-designed Pep through many discussions, and provided feedback on this paper. Neil Wilhelm has been and is (I'm sure!) an unlimited source of encouragement, bright ideas, and visions. We would like to thank Cristina Cifuentes and the anonymous reviewers for carefully reading the paper and offering suggestions that helped us significantly improve it. We used Georg Sander's vcg tool to draw control-flow graphs [19]. Jeff Chan gave us the RayTracer benchmark.

References

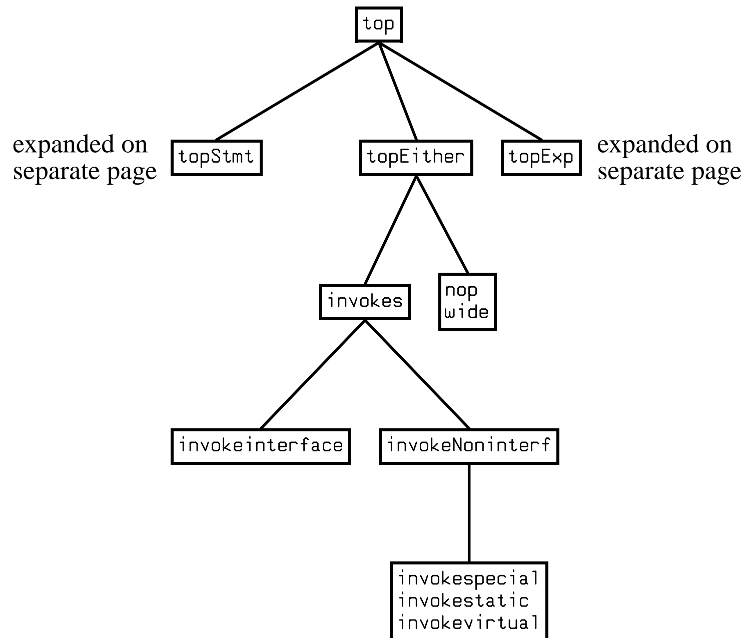
1. Brenda S. Baker. An Algorithm for Structuring Flowgraphs. In *Journal of the ACM*, 24(1), p. 98-120, January 1977.
2. Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June 1991. Originally published in *OOPSLA'89 Object-Oriented Programming Systems, Languages and Applications,* p. 49-70, New Orleans, Louisiana, October 1989.
3. Cristina Cifuentes. Structuring Decompiled Graphs. In *proceedings of the International Conference on Compiler Construction*, p. 91-105, Linköping, Sweden, 1996. Springer-Verlag (LNCS 1060).
4. Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time. To appear in *IEEE Micro.*
5. L. Peter Deutsch and Alan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, p. 297-302, Salt Lake City, Utah, January 1984.
6. Jack Dongarra and Reed Wade. *Linpack Benchmark—Java Version*, http://www.netlib.org/benchmark/linpackjava, April 1996.
7. Michael Franz and Thomas Kistler. *Juice*. http://www.ics.uci.edu/~juice/intro.html, June 1996.
8. Michael Franz and Thomas Kistler. *Slim Binaries*. Technical Report no. 96-24, Department of Information and Computer Science, University of California, Irvine, California, U.S.A., June 1996.
9. James Gosling. Java Intermediate Bytecodes. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, p. 111-118, January 1995. Published as *ACM SIGPLAN Notices* 30(3), March 1995.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, The Java Series, Addison-Wesley, 1996.
11. Jonathan Hardwick. *Optimizing Java Linpack.* http://www.cs.cmu.edu/~jch/java/linpack.html, September 1996.
12. Urs Hölzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. In *ACM Transactions on Programming Languages and Systems*, 18(4), p. 355-400, July 1996.
13. JavaSoft. *Java Native Interface Specification*. http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/spec/jni-TOC.doc.html, February 1997.
14. Brian T. Lewis, L. Peter Deutsch, and Theodore C. Goldstein. *Clarity MCode: A Retargetable Intermediate Representation for Compilation.* Sun Microsystems Laboratories Technical Report SMLI TR-95-43, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, May 1995.
15. Ulrike Lichtblau. Decompilation of Control Structures by Means of Graph Transformation. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, p. 284-297, Berlin, March 1985. Springer-Verlag (LNCS 185).
16. Tim Lindholm and Frank Yellin. *The Java Virtual Machine,* The Java Series, Addison-Wesley, 1996.
17. Daniel R. Perkins and Dennis Volper. UCSD Pascal on the VAX, Portability and Performance. *Software—Practice and Experience*, 14(5), p. 473-482, May 1984.
18. Peter Wayner. Sun Gambles on Java Chips. *Byte*, p. 79-88, November 1996.
19. Georg Sander. Graph Layout Through the VCG Tool. In Roberto Tamassia and Toannis G. Tollis (Eds.) *Graph Drawing, DIMACS International Workshop GD'94*, p. 194-204, October 1994. Springer-Verlag (LNCS 894).
20. Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan 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.
21. Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming,* p. 303-330, Århus, Denmark, August 1995. Springer-Verlag (LNCS 952).
22. Bjarne Stroustrup. Multiple Inheritance for C++. In *Proceedings of the European Unix Users Group Conference '87*, p. 189-207, Helsinki, Finland, May 1987.

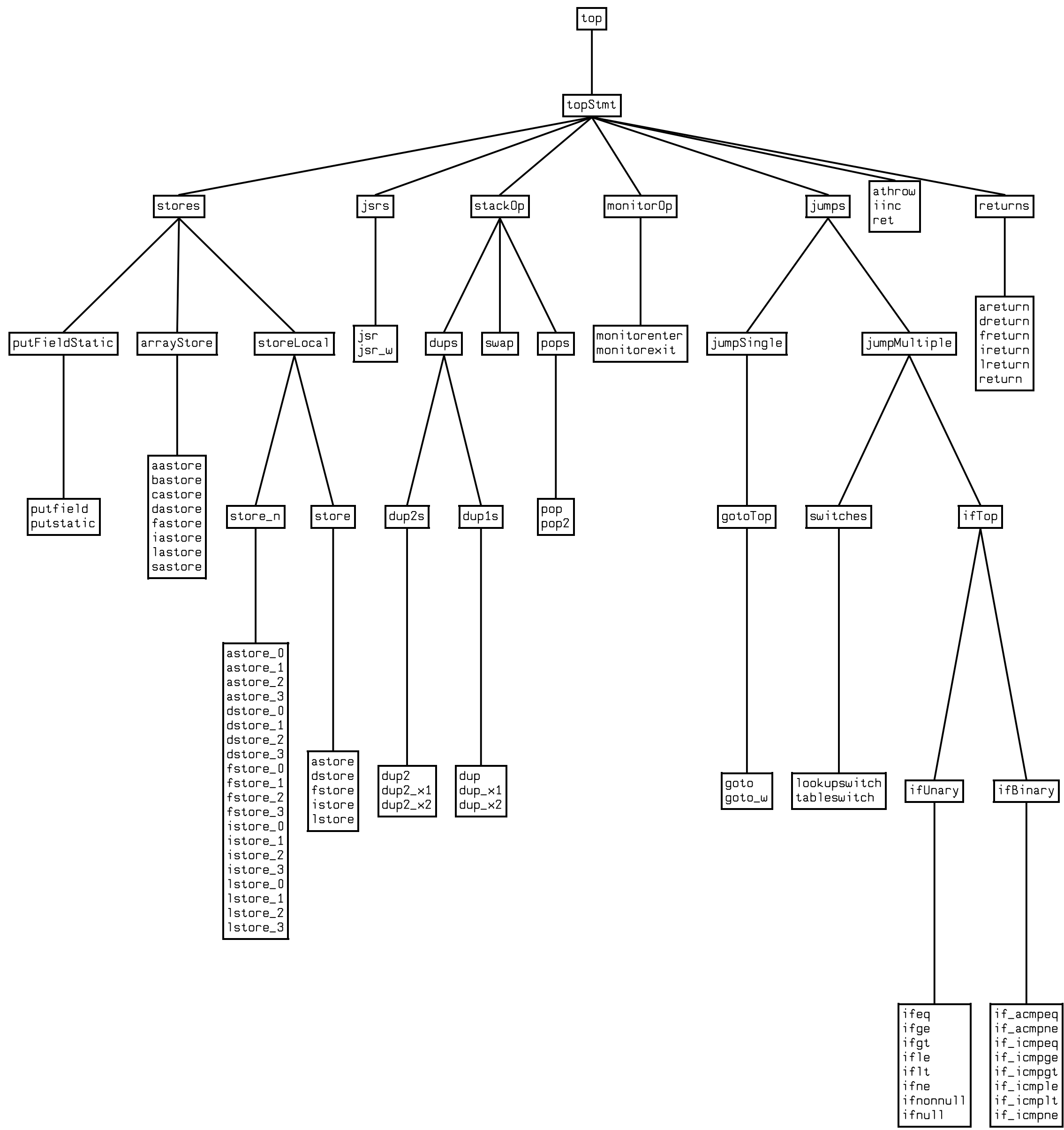
23. David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications*, p. 227-241, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

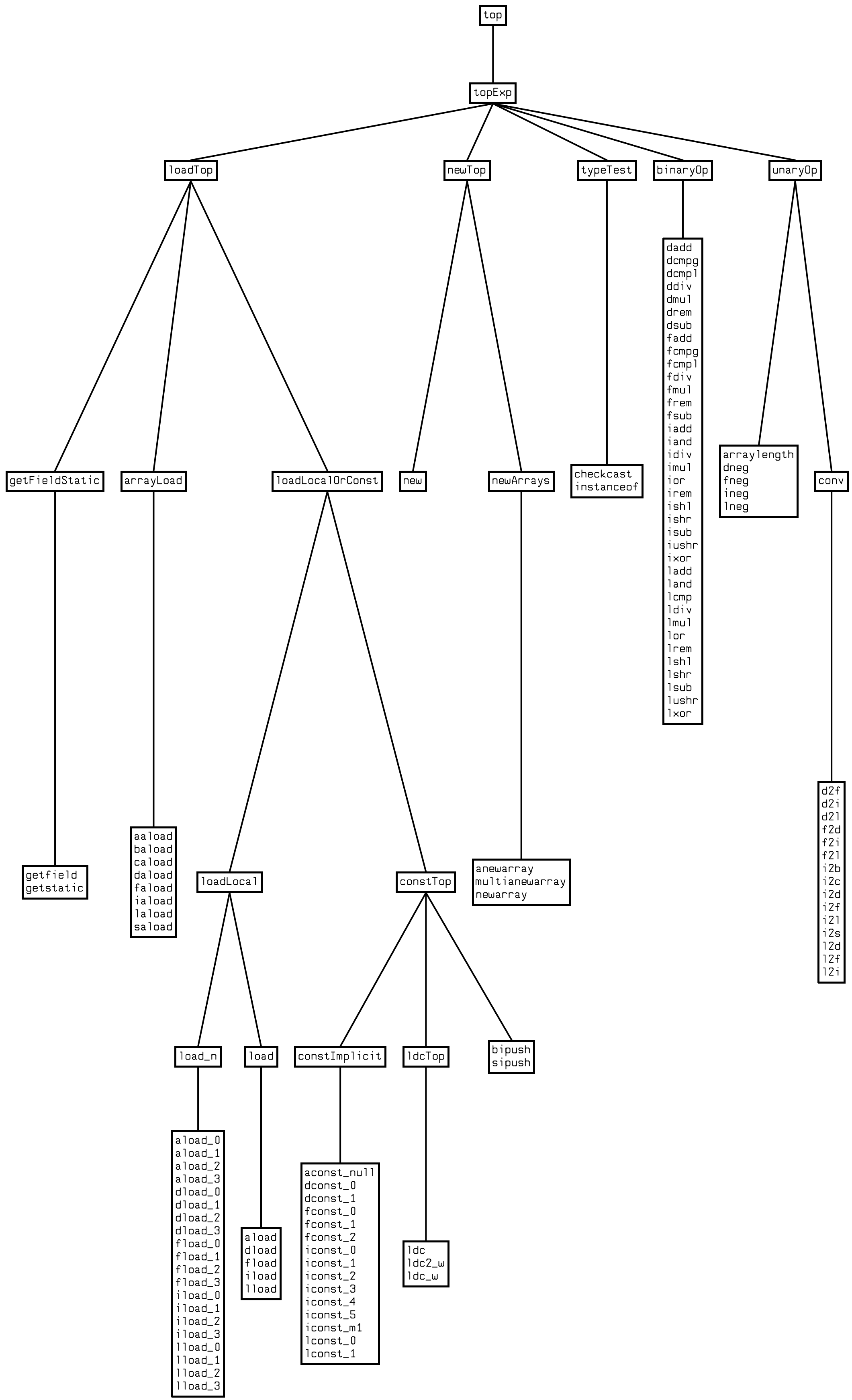
Appendix

Pep classifies the 200 Java bytecodes into an inheritance hierarchy. The classification, although developed specifically to organize the behavior needed for translation of Java bytecodes, can also be a useful aid in understanding the bytecode set and, perhaps with modifications, be used for other computations over bytecodes.

The following three figures show the full bytecode set. Only the leaves represent actual (concrete) bytecodes. The internal nodes represent abstractions of the concrete bytecodes. Most of the behavior needed to translate bytecodes is defined in the abstract nodes and therefore applies to all the concrete bytecodes inheriting the code. The most fundamental distinction that Pep makes is between bytecodes that always, sometimes, and never produce a value when evaluated. Thus, the "top" abstract bytecode, from which all concrete bytecodes inherit, has three direct children: topStmt (which never produces a value), topEither (which may or may not produce a value), and topExp (which always produces a value). The topEither part of the hierarchy is shown below. To allow the figures to have a reasonable font size, the topStmt and topExp nodes and their children are shown separately on the following two pages.




top
topStmt
stores
jsrs
stackOp
monitorOp
jumps
athrow
iinc
ret
returns
putFieldStatic
arrayStore
storeLocal
jsr
jsr_w
dups
swap
pops
monitorenter
monitorexit
jumpSingle
jumpMultiple
areturn
dreturn
freturn
ireturn
lreturn
return
putfield
putstatic
aastore
bastore
castore
dastore
fastore
iastore
lastore
sastore
store_n
store
dup2s
dup1s
pop
pop2
gotoTop
switches
ifTop
astore_0
astore_1
astore_2
astore_3
dstore_0
dstore_1
dstore_2
dstore_3
fstore_0
fstore_1
fstore_2
fstore_3
istore_0
istore_1
istore_2
istore_3
lstore_0
lstore_1
lstore_2
lstore_3
astore
dstore
fstore
istore
lstore
dup2
dup2_x1
dup2_x2
dup
dup_x1
dup_x2
goto
goto_w
lookupswitch
tableswitch
ifUnary
ifBinary
ifeq
ifge
ifgt
ifle
iflt
ifne
ifnonnull
ifnull
if_acmpeq
if_acmpne
if_icmpeq
if_icmpge
if_icmpgt
if_icmple
if_icmplt
if_icmpne


top
topExp
loadTop
newTop
typeTest
binaryOp
unaryOp
dadd dcmpg dcmpl ddiv dmul drem dsub fadd fcmpg fcmpl fdiv fmul frem fsub iadd iand idiv imul ior irem ishl ishr isub iushr ixor ladd land lcmp ldiv lmul lor lrem lshl lshr lsub lushr lxor
getFieldStatic
arrayLoad
loadLocalOrConst
new
newArrays
checkcast instanceof
arraylength dneg fneg ineg lneg
conv
d2f d2i d2l f2d f2i f2l i2b i2c i2d i2f i2l i2s l2d l2f l2i
getfield getstatic
aaload baload caload daload faload iaload laload saload
loadLocal
constTop
anewarray multianewarray newarray
load_n
load
constImplicit
ldcTop
bipush sipush
aload_0 aload_1 aload_2 aload_3 dload_0 dload_1 dload_2 dload_3 fload_0 fload_1 fload_2 fload_3 iload_0 iload_1 iload_2 iload_3 lload_0 lload_1 lload_2 lload_3
aload dload fload iload lload
aconst_null dconst_0 dconst_1 fconst_0 fconst_1 fconst_2 iconst_0 iconst_1 iconst_2 iconst_3 iconst_4 iconst_5 iconst_m1 lconst_0 lconst_1
ldc ldc2_w ldc_w