MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1576 4/96

Supporting dynamic languages on the Java virtual machine

Olin Shivers
`shivers@ai.mit.edu`

Abstract

In this note, I propose two extensions to the Java virtual machine (or VM) to allow dynamic languages such as Dylan, Scheme and Smalltalk to be efficiently implemented on the VM. These extensions do not affect the performance of pure Java programs on the machine. The first extension allows for efficient encoding of dynamic data; the second allows for efficient encoding of language-specific computational elements.

This report also appeared in the proceedings of the Dynamics Objects Workshop, May, 1996.

This publication can be retrieved by anonymous ftp from publications.ai.mit.edu.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Defense Advanced Research Projects Agency of the Department of Defense under Rome Laboratory contract F30602-94-C-0252

1 Introduction

Java is easily the highest-profile language development effort going today. By riding the tidal wave of Internet growth, Java is being propagated out to systems world-wide. This presents a tempting opportunity to language implementors: if a programming language can be compiled for the Java VM, then it can be executed on millions of computers. A port to the Java VM is a port to practically every important hardware platform in existence. This has, for example, drawn the attention of DARPA, where program managers have expressed interest in having the results of programming language research targeted to the Java VM as a means of doing instantaneous technology transfer to the commercial sector.

However, the Java VM is not a good target machine for dynamic languages, for reasons involving efficient representations of both data and computation in dynamic languages. In this note, we'll consider some of the difficulties imposed by the current VM specification, and examine two extensions to alleviate these problems.

2 The data representation problem

Dynamic languages typically require special representations for their data values. Consider, for example, the requirements imposed by Scheme's polymorphism and dynamic type system. Scheme's polymorphism requires a "uniform representation" of data. For example, the car and cdr slots of a cons cell can contain any value of any type, so all values of all types must be able to fit into the storage allocated for the slots of the cons cell. Similarly, the formal parameter of a procedure can be bound to any value of any type. This means that subroutine linkage conventions must be able to accept any value in the expected place, whether it is a character, integer, array, or other value.

Scheme implementations solve the one-size-fits-all problem by representing all values with exactly one machine word:

- If the value is smaller than a single word (*e.g.*, a character or boolean), it is padded to fit into a single word. Some of the word's bits are used to represent the type of the value. Let us call these values "immediate descriptors."

- If the value is larger than a single word (*e.g.*, a cons cell or double-precision floating-point number), it is "boxed"—that is, the actual data for the value is stored in memory, and the value is represented by the address of that region of memory. The bit-encoding of the address needs to be distinguishable from bit-encoding of immediate descriptors; beyond this simple discrimination, more complex type information can be stored in the header of the associated block of memory.

Unfortunately, the Java virtual machine has been designed as a target for a monomorphic, statically-typed object-oriented language. To the degree that polymorphism is supported by the virtual machine, it must be realised by the class hierarchy, which allows us to build objects of different run-time "types."

For example, suppose we wish to compile Scheme code for the Java virtual machine. The different types of Scheme values can be represented as sub-classes of the Java `Object` class: we can represent small integers as Java `Int` objects; define a `Procedure` class for Scheme procedures, with `Closure` and `Primop` sub-classes; and so forth. The VM allows one to perform run-time tests to query if an object is of a given sub-class, so we can perform the run-time type tests that Scheme needs as desired.

The difficulty with this representation is that small objects, such as characters, small integers, and booleans, must be represented as general Java objects. The Java virtual machine uses a boxed representation to be able to uniformly describe objects of different classes: an object is represented by a pointer to a chunk of memory containing a description of its class and its state variables. As with Scheme data, boxing Java object data allows us to represent all Java objects uniformly, with exactly one machine word.

The overhead of boxing is acceptable for multi-word objects, but is quite expensive for objects that fit within a single word. Implementations of dynamic languages typically use clever encodings to represent these values as immediate descriptors—allowing them to express the value in a single machine word, without requiring any extra information to be stored in memory.

Immediate descriptors and pointers to boxed values are frequently discriminated by using a single low-order bit of the descriptor. Since backing store for boxed objects is typically allocated on word-size boundaries, on a byte-addressable machine the low-order bit is unused by pointers, and so can be used for this low-level discrimination.

For example, some Scheme implementations represent immediate data (such as characters, 30-bit integers, the empty list, and booleans) as 32-bit patterns whose least-significant bit is zero. The next bit is used to discriminate 30-bit integers from other immediate descriptors: if it is zero, the 32-bit pattern represents a 30-bit integer; if it is one, the next six bits are used to provide the type information for the descriptor, leaving three bytes from the descriptor for the actual data.

Note that making the small-integer type tag be the bits "00" means that we can add and subtract these values without having to convert to the raw machine representation and back; multiplication requires one normalising shift before the multiply; division requires a shift after the divide.

Scheme, Lisp, Dylan, and Smalltalk implementors have exploited these sorts of representational devices for decades to achieve good performance from their systems.

Without immediate-descriptor representations, the boxing/unboxing costs can swamp the actual cost of computing with small values. For example, adding two small integers would involve two memory fetches to unbox the addends, an add instruction, and then an allocation to box the new value. String and character processing will have similar overheads. In short, while our large, composite data structures remain fairly efficient, it becomes much more expensive to compute with the primitive scalar data values.

2.1 Adding immediate descriptors to the Java VM

A simple extension to the Java VM will allow us to reap the benefits of immediate descriptors. We can do this by making two, simple, backwards-compatible changes to the VM:

- All pointers to boxed Java objects must have a low bit of one. This is not an onerous restriction: if boxed objects are allocated in word units, on a byte-addressable machine the low two bits of the pointer are unused.

- A new class, `ImmediateDescriptor`, is introduced. `ImmediateDescriptor` is a direct sub-class of the `Object` class, and has exactly 31 bits of state (we might also wish to define a `LongImmediateDescriptor` class with 63 bits of state). The `ImmediateDescriptor` class is a final class (meaning that it cannot be sub-classed), so if a Java object is known to be of class `ImmediateDescriptor`, it requires no more than 31 bits to represent its value.

  An `ImmediateDescriptor` object is represented as a machine word with low bit 0. It requires no backing storage, and can reliably be distinguished from boxed objects by its low bit.

  An `ImmediateDescriptor` can be cast to an even 32-bit integer; such a cast is merely a compile-time change of viewpoint, requiring no run-time computation. Similarly, an integer can be cast to an `ImmediateDescriptor`; the Java VM implements this cast by shifting the integer left one bit.

Implementations of dynamic languages can now use immediate descriptors to efficiently represent small data values without requiring the allocation of backing store.

Introducing immediate descriptors into the VM should have almost no impact on programs that do not use them. It is worth examining, however, the task of method lookup on Java objects for performance impact. Method lookup of standard Java objects involves indirecting through the pointer that represents the boxed object. Such an indirection is not well-defined for descriptors that are not addresses, such as immediate descriptors. So the Java machine must perform method lookup on objects of class `ImmediateDescriptor` using an alternate technique, for example, by checking in a special method table known to the VM.

If an object is known to be any sub-class of the `Object` class that isn't the `ImmediateDescriptor` class, then it is guaranteed to be boxed, and the standard method lookup can be employed. If an object is known to be an `ImmediateDescriptor`, then the special method-lookup technique can be used instead. However, if the object is of class `Object`, then it could be either boxed or immediate, and the VM must perform a run-time test on the low bit to determine how to look up its method. Fortunately, we can use a classic Lisp trick to optimise this case: since the two kinds of descriptor are distinguished by their low bit, the VM can simply assume the descriptor is a boxed object and perform the necessary memory load to start the method lookup. If the descriptor is actually an immediate descriptor, this will generate a memory-alignment exception on the underlying hardware. The VM can catch this exception and vector off to the custom code for doing method lookups on immediate descriptors. With this trick, supporting immediate descriptors in the VM imposes zero cost on operations that are performed on boxed Java objects.

3 The computational representation problem

Now let us turn to the problem of using the Java VM to represent the computations to be performed upon our data. The problem here is to be found in the design of the instruction set.

3.1 Encoding tension: trust *vs.* efficiency

The basic issue addressed by the design of the Java VM is the tension between trust and efficiency. We wish to describe our computation with a notation (that is, a language) that

- can be verified locally, and
- is efficiently executable.

The Java VM's instruction set fits these criteria. It fits the second criterion because it was carefully designed to efficiently encode programs expressed in Java. Now that we are trying to extend the coverage to other programming languages, we are having trouble meeting the efficiency criterion—the Java VM does not efficiently express Scheme or Dylan programs. What to do? If we changed the VM to be a generic RISC instruction set, we'd get our efficiency back, but we'd lose the safety criterion. How do you trust a program expressed in assembler? That is the tension.

The essence of the problem with the Java VM as a general-purpose instruction set is that it is a CISC: a high-level, highly encoded instruction set that is carefully tuned to the demands of the Java language. The individual building blocks—the instructions—for expressing a computation are complex, high-level, safe units. For example, the Java VM "hard-wires" method lookup into a single instruction.

The problem with CISC instruction sets is that they are brittle encodings. If a computation fits the instruction set exactly, things are good: it can be encoded compactly, and executed efficiently. But if a computation is just slightly different, there is no simple, efficient encoding. For example, if our object-oriented language does method lookup just a little bit differently from the Java semantics that are implemented by the VM's instruction set, we can't use the VM's class system or the VM's method-lookup instruction.

This is not the fault of the Java VM. It was tuned for implementing Java and cleverly exploits this constraint to achieve dense, efficient encodings of programs written in Java. The problem is that now the VM is being used for new purposes—as the target of different languages.

This limitation of CISC instruction sets has always been at the heart of the RISC philosophy: encoding programs at a lower-level means there is more room to maneuver when the compiler is mapping high-level programs down to the machine in ways that efficiently utilise the machine's resources. The lesson here is that this holds as much for the Java VM in the nineties as it did for the VAX 11/780 in the eighties.

3.2 Writeable control store and trust boundaries

Fahlman has made an interesting proposal [Fahlman96] to help "open up" the Java VM's instruction set in order to address this problem. The idea is to allow some of the opcode space in the VM's instruction set to be implemented by C routines that are dynamically linked into the VM. The implementors for a given language could therefore design and use the extra handful of instructions that efficiently express computations written in their language. This is essentially the VM analog of writeable control store—extensible microcode.

What we are doing here is playing with the trust boundaries to improve the efficiency of the whole system. The microcode extension allows us to define new safe building blocks using a dangerous implementation substrate. However, allowing for "microcode" written in C and delivered as raw machine code to be dynamically loaded into the VM requires us to decide why we are going to trust the microcode, and how we are going to verify programs that use these instructions.

This technique exploits the fact that there aren't very many language implementations—there are many more programs than there are languages. A handful of microcode libraries would need to be developed by the language community. Perhaps we would end up with one from Harlequin for doing Dylan (with Dylan-style method lookup in an instruction), one from Hewlett-Packard for doing Scheme (with closures, tail-recursion, and tagged arithmetic), and so forth. The Dylan implementors at CMU might either collaborate with the Harlequin team on a common instruction-set extension, or independently develop one tuned for their compiler technology. The source code for these few microcode libraries would be submitted to a Java consortium, who would check them and publish digitally-signed copies on the network. When an application is loaded by the Java VM, if the VM doesn't have the necessary microcode loaded, it fetches it from a server and verifies the consortium's digital signature. If the application is implemented using some experimental system which hasn't yet received the consortium's imprimatur, the user would have the option of loading in the microcode from other routes—perhaps he got a floppy disk from CMU with their Dylan release that he trusts.

To repeat, notice that the issue is: why should I trust your microcode, which comes written in some dangerous, unverifiable language, such as a SPARC a.out file or a chunk of C code? The example answers I have been giving rely on the fact that the scale of the problem is very containable, since it per language implementation, not per program. If the user will deal with the mechanics of authorising his computer to trust Hewlett-Packard's Scheme microcode, he can now run any Scheme application in the world that was produced with Hewlett-Packard's Scheme-to-Java-VM compiler.

This little extra degree of freedom means that language implementors such as a Dylan development team could design those few extra instructions that would make Dylan run well on the VM—exploiting the greater freedoms of a dangerous implementation substrate such as C.

Note that this technique is mostly concerned with describing programs, not data. It doesn't help us tune data representations. For example, to address the Scheme arithmetic boxing problem, we need the immediate-descriptor technique described earlier.

3.3 Compiler-oriented microcode

Fahlman's proposal provides an extensible method for breaking free of the limits imposed by the current VM's CISC, Java-focused design. However, there is a serious problem with the proposal as it stands: it is oriented towards interpreter implementations of the VM, not compiler (or other) implementations. This limits efficiency. The Java byte-codes are designed to allow them to be translated "on-the-fly" into efficient native code when they are loaded off the network. But our new instructions are described to the Java engine with C code or some routines that are delivered to the VM simply as a chunk of raw native machine code. How will the VM's byte-code translator be able to translate the use of such an instruction to native code?

A simple answer is just to translate each use of the instruction into a subroutine call to the machine-code subroutine that defines it. This technique is adequate for very large-granularity instructions, such as FFT (to choose an extreme example), or perhaps method lookup in some complex object-oriented language. The overhead of the subroutine call will be amortised by the time spent executing the instruction. This overhead is unacceptable, however, for "little" instructions, such as an overflow-checking add-with-trap instruction. Unfortunately, these instructions are also quite important for efficient implementations of dynamic languages.

This leads us to a compiler-oriented variant of Fahlman's microcode proposal. Let us define some lower-level machine underlying the VM—such as a generic RISC processor with operations described in a simple RTL language. For example, we could use an RTL representation similar to the one that is employed for the intermediate representation of the gcc compiler. Note that the RTL language is a dangerous language, not a restricted, safe one—it is the dangerous, trusted machinery we use to define our safe building blocks.

New instructions are defined with macros that expand from a use of the instruction to its implementation in RTL. If the instruction is simple, such as add-with-trap, then the macro expands into a small quantity of RTL which directly performs the operation *in toto*. If the instruction is complex, such as method lookup for an object-oriented language, then the macro expands into a subroutine call to a shared microcode routine that does the operation. The shared microcode routine is also described with the dangerous RTL sublanguage.

Now an application that uses instruction-set extensions has multiple implementation possibilities:

- A compiling implementation that runs on a stock microprocessor can expand instruction uses appearing in the byte-codes into RTL and then translate to native code using standard compiler technology. (The basic Java VM instructions can be specially handled by the translator for fast translation.) We trust the macro-expanders to generate safe uses of the dangerous RTL—that is the trust requirement, and it is also the little extra degree of freedom that gives us efficiency.

- A hardware Java engine would simply implement the RTL engine as well.

- A byte-code interpreter that does not translate to native code can either expand all the extended instructions to RTL, or macro-expand a canonical code sequence for a particular instruction (such as add-with-trap), producing a little subroutine for the interpreter to call on each use of that instruction.

We can limit our macro language as we please—it doesn't have to be Turing equivalent.

The main point of this proposal is that if we choose to make the Java VM extensible, we must be careful in choosing how we express these extensions. The extensions should be described in some form that is amenable to a spectrum of implementation strategies; a form which our computer systems can manipulate efficaciously. C source and processor-specific machine code do not satisfy these requirements.

This extension mechanism is clearly less well-developed than the immediate-descriptor technique presented in the first half of this note. Much detail needs to be developed, such as the exact design of the RTL language used for defining extended instructions, or the associated instruction meta-information used to describe to the byte-code verifier the static type constraints of the defined instructions.

4 Conclusion

Using the Java VM as a propagation vector for distributing dynamic languages out to a large audience is a tempting goal. We've examined two difficulties with targeting dynamic languages to the current Java VM: the overhead of boxing small data structures, and the mismatch between the VM's Java-tuned instruction set and the requirements of dynamic languages. The former problem can be addressed with a simple, backwards-compatible extension to the VM. It appears that this extension should have no performance impact on current Java programs. The latter problem can be addressed with the more speculative proposal of adding an extension mechanism to the VM that is amenable to compiler processing.

Bibliography

[ALIT] Peter Lee (editor). *Topics in Advanced Language Implementation.* MIT Press, 1991.

[Dylan] *Dylan: An Object-Oriented Dynamic Language.* Apple Computer, 1992.

[Fahlman96] Scott E. Fahlman. Email to the `java-vm@life-ai.mit.edu` mailing list, January 10, 1996. The `java-vm` mailing list is archived at the MIT AI Lab; this message can be located at `http://wilson.ai.mit.edu/java-vm?37`

[Java] Java: Programming for the Internet. Sun Microsystems, 1995. `http://java.sun.com/`

[Scheme] J. Rees and W. Clinger (editors). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* 21(12):37–79, December 1986.