Some Shortcomings of, and Possible Improvements to, the Java Virtual Machine.

Nick Benton. MSR Cambridge.

nick@microsoft.com

DRAFT OF 02/06/99 4:59 PM

Introduction

Whilst working on MLJ, an optimizing compiler for Standard ML which emits Java bytecodes, we inevitably discovered a number of irksome, inefficient or ugly aspects of the JVM design, and of its implementations. This short note attempts to summarize just a few of these flaws. Although my perspective is that of someone who was trying to use the JVM in a way that its designers did not originally envisage, most of the issues are ones which if addressed would improve the JVM as a target for Java itself (or at least a better version of Java).

The list is in no particular order and includes both trivial matters and significant ones about which there is much more to be said. I'd be happy to discuss any of these subjects in more detail.

Tail calls.

The lack of fully general tail calls in (as far as I know) all current implementations of the JVM is a severe problem for compiling any declarative language to Java bytecodes. We were able to compile uses of tail recursion which correspond to simple loops using the **goto** bytecode, and other transformations often made calls which were not obviously tail calls in the source into jumps in the compiled code, but some programs would still run out of stack space quite unnecessarily on reasonably-sized inputs. This tends to happen on programs written in a non-trivially higher order style, which is certainly something which also occurs in object-oriented programming, particularly with the rise of slightly more abstract patterns which encapsulate control flow. For example, the following slightly contrived Java code, which I found on the web as an example of the "visitor pattern", traverses a linked list and will fail on large lists if tail calls are not implemented properly:

```
interface List {

       void accept(Visitor v);
}

class Nil implements List {
       public void accept(Visitor v) {
        v.visit(this);
       }
}

class Cons implements List {
       int head;
       List tail;
       public void accept(Visitor v) {
              v.visit(this);
       }

       public Cons(int x,List y) {
              head = x;
              tail = y;
       }
}

interface Visitor {
       void visit(Nil x);
       void visit(Cons x);
}

class SumVisitor implements Visitor {
       int sum = 0;
       public void visit(Nil x) {}
       public void visit(Cons x) {
        sum += x.head;

        // Tail Call
        x.tail.accept(this);
       }
}
```

Java programmers have also reported problems serializing large objects because of a lack of proper tail-calls, though this is also caused by the fact that more general (non-tail) deep recursion is not always well supported. See comments under Stack Size later.

There really is no way to program around a lack of proper tail calls in the underlying implementation without making the code grossly larger and slower. (Top-level interpretive loop, trampoline.)

I had some email conversations last year with James Gosling and Guy Steele from Sun about getting JVMs to support tail calls. They were both very aware of the problem and keen to fix it. They had two problems; both caused by freezing an inferior design choice too early. The first was simply that with the VM spec being so widely propagated and implemented by so many independent groups, making almost any change was practically impossible, particularly given the holy mantra of "Write Once, Run Anywhere". So actually *requiring* implementations to do tail call elimination was politically impossible. The second was the not entirely trivial interaction of tail call elimination with the (ugly) security model of Java 1.1 which again, couldn't be completely dropped for backwards compatibility reasons.

Despite these difficulties, Sun certainly want to support general tail calls and hope to do so in future JVMs. Steele is strongly in favour, as, I think, is Bill Joy. Gosling supports the idea of "changing the VM spec to strongly encourage VM implementations to do tail recursion elimination when they can, but to not require it". I believe that "when they can" means something like "between methods coming from classes with the same **ClassLoader**".

Security Managers

The Java 1.1 security model includes methods in the **SecurityManager** class for explicitly examining the runtime stack (**classDepth**, **inClass**, **classLoaderDepth**, **inClassLoader** and **getClassContext**). This, as several people have observed, is a bad way to implement security policies because it is far too low-level. It also inhibits optimizations, such as tail call elimination (see above). Capability or permission based security is much better, and has been adopted for Java 1.2 in which all the above methods (except, oddly, the last) are deprecated. But of course, they're still there for backward compatibility.

Stack Size

Even if tail calls were implemented properly, there would still be a problem with many JVMs (e.g. most versions of Sun's) in that they have a (by default relatively small) limit on frame stack size. By limiting call depth to a few thousand this often causes naturally recursive code, such as that for serialization, to run out of stack space. Microsoft's JVM and the most recent version of Sun's seem to have a larger fixed limit (using the underlying C stack?), at least for the main thread, but no obvious way of increasing the default. Any modern programming language implementation, especially one with multiple threads, should be able to expand stacks as necessary up to the limit of available memory.
A more radical approach is simply to allocate activation records on the heap and have them be collected like normal objects. This is particularly handy for languages with first-class continuations, such as Scheme and the New Jersey implementation of SML, and can help with the implementation of closures (see below). Appel and Shao have claimed (not, I admit, completely convincingly) that with a modern generational collector, heap allocated frames can be as efficient as stack allocated ones, at least for SML http://flint.cs.yale.edu/flint/publications/stack.html.

Local variables

The JVM has a slightly odd model of per-frame storage. There are local variables (directly addressable by number) and an evaluation stack (accessible from the top). This is a very naïve design – being, like so much else, a direct translation of constructs in the Java source language. It does not map onto real processor architectures at all well, as a good JIT will unpick all the stack and local usage and do its own job of register and memory allocation anyway. Mostly in the interests of keeping code compact, we ended up

putting quite a lot of effort into deciding what to keep in locals and what to leave on the stack. This turned out to be complex and not helped by the fact that the stack-manipulation instructions in the JVM are not designed to be general-purpose, so some permutations are tricky to achieve.

The other problem we had with locals is that they caused space leaks on some JVMs. When something is used by being popped from the stack, it's gone as far as the GC is concerned. When something is stored in a local but not live, we have to explicitly write a null over it (at least, if it might point to a large object and we are about to do something expensive). Of course, this is a waste of instructions if the JIT detects the dead variable itself, but not all do. Microsoft seems better than Sun/Symantec at doing this.

The real point here is that the stack/local distinction is just unnecessary – if there were stack-pointer relative addressing instructions then there'd be no need for locals at all and one could generate much neater code. (Alternatively, one could just have three-address code with no stack and again leave actual allocation up to the JIT, but stack-based code tends to be a bit more compact. But it's really having both which seems messy.)

Pointless instruction variants

There are several instructions that exist in several variants for different types, such as **iload**/**aload**, **istore**/**astore**, **ireturn**/**areturn**. Given that the sizes of the operands are the same, there is simply no need to distinguish these instructions as they do exactly the same thing operationally. So if you're not verifying, then it doesn't matter which one you use. And if you *are* verifying, then the verifier has to be able to work out (from method signatures and uses of type-specific opcodes) what type will actually be on the stack at that point, not least so it can check you've used the right instruction! Therefore, nothing is gained by complicating the instruction set in this way. (There is possibly a case for having multiple mnemonics for one bytecode in an assembler, though.)

Initializers

Many things to do with initializers are a bit tricky. One of these is the separation of allocation from initialization, which was presumably done so that it was easy for one initialization method to call another. Javac only ever seems to produce the sequence **new**; **dup**; **invoke** <**init**>. It would have been better to combine all that into one alternative call instruction.

One of the things the verifier enforces is that a chain of initializers be called on every newly allocated object before it may be used. This adds complexity and potentially makes allocation unnecessarily expensive (though it *may* be that a good JIT inlines the whole chain in most cases). With a few tweaks to the standard library, I believe one could be far more liberal in the treatment of uninitialized objects. Given that the **new** instruction already fills fields with zero or null as appropriate (according to the JLS), insisting on the calling of initializers is not necessary for type-safety at the bytecode level. Of course, there are security-critical classes in the standard libraries which should never be used before a certain amount of initialization has been done, but that is not a reason to impose a blanket requirement that a chain of initializers be called for every heap-allocated object.

It may well be true that it's good object-oriented *language* design to insist on initializers being called, so that all Java objects do get explicitly initialized on creation, but that *still* doesn't seem to imply that this should be enforced by the verifier.

I have not found much in the way of discussion of this in the literature. Allen Goldberg's "A specification of Java loading and bytecode verification" http://www.kestrel.edu/HTML/people/goldberg/Bytecode.html says

> The **new** instruction initializes the fields of the object with default values for each type. Thus, type safety is assured, even if **<init>** is not called. Nevertheless, the security of the JVM is dependent upon executing a proper initialization sequence, since user-defined classes such as extensions to the **classLoader**, must meet security-critical interface requirements, that are *at least partially satisfied* by insuring [sic] proper initialization. [my italics]

which I don't find entirely convincing. The other situation in which knowing that initializers have been called can be used to ensure safety properties is when one uses native methods. For example, a **Window** object might include a field holding a handle to an underlying native window manager object. If that

weren't correctly set up on creation, then subsequent calls to native methods could dump core or do other bad things.
I think there should be a way to deal with these essentially application-level interface requirements just using the more fundamental abstraction/hiding facilities already provided by the object model. I haven't thought all of this through (comments welcome!) but I think the right thing to do is to have object creation happen "internally" via a static method call, to which the usual modifiers can be applied, rather than "externally" via the **new** instruction. Then the question is whether visibility and finality restrictions are sufficient to allow the authors of security-critical code to guarantee the appropriate invariants.
In both the security-critical class case and the native code case, the problem seems to be inheritance – the standard library defines some class which users can extend, and something could go wrong if an instance of the new subclass were created without calling some initialization code in the superclass. If it weren't for inheritance, there'd be no problem – the only way to create a **Window** would be to call the creation method which was written to do the right thing. So I think the answer is that in these few critical cases, one simply writes the code in a different style, using delegation rather than inheritance. Exactly how delegation is used would be a matter of style, but for the **Window** example, one might choose to make the block of native pointers in every **Component** be an instance of a final class **ComponentHandles** (or something) with a protected creation method. Then a subclass of **Window** which didn't call the superclass initializer might not be very useful, but the worst that could happen would be a **nullPointerException** because its **ComponentHandles** field would never have got filled in.

No parameterized classes or methods (generics)

Parametric polymorphism is A Very Good Thing and is not supported in either Java or the JVM. We could have made good use of parameterized classes and methods in the JVM when compiling SML. Many Java programmers have complained about the lack of generics in Java and there are now a fair number of proposals for adding them. Some of these involve extending the Java language but translating the extensions into the current bytecodes, whereas others also involve changes to the JVM. Parametric polymorphism should clearly have been part of the basic type system of the JVM in the first place – compiling it out causes problems with libraries and debuggers, and increases code size and run-times.
References include

- Andrew C. Myers, Joseph A. Bank, Barbara Liskov "Parameterized Types for Java**"** http://www.pmg.lcs.mit.edu/papers/popl97/popl97.html. This proposal is being implemented under the name PolyJ http://www.pmg.lcs.mit.edu/polyj/.
- Kim B. Bruce "Increasing Java's Expressiveness with `ThisType` and Match-Bounded Polymorphism" ftp://ftp.cs.williams.edu/pub/kim/LOOMJava.ps.gz
- Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler "Making the future safe for the past: Adding Genericity to the Java Programming Language". Also implemented, see the GJ project homepage http://www.cs.bell-labs.com/~wadler/pizza/gj/ for more info.

It seems very likely that some form of parameterized class will be added to Java in some future release – Sun are well aware of the issues, Bracha and Stoutamire (cited above) work at Javasoft, and Gosling cites this and fixing the floating point problems as the most-wished-for changes. (It's in the top 25 in the JDC buglist.)
Parameterized classes, if included in the underlying type systems, add no runtime cost. In fact, they improve efficiency by removing either spurious dynamic type checks and casts or code duplication. Compared with a language like SML, parameterization in the JVM would have to be slightly less universal because of the built-in distinction between primitive and reference types. One could only sensibly quantify over reference types, and there are advantages to allowing *bounded* quantification over subclasses of a particular class, or over classes implementing a particular interface. (Note that Haskell includes unboxed types, which may not be abstracted on, and has a form of bounded quantification in its type class mechanism.)

One silly flaw in Java, which would be fixed by having parametric polymorphism, is the problem of array subtyping:

Array subtyping is wrong

The type system of Java makes the array type constructor covariant in its argument type, so if B extends A then B[ ] is assignment convertible to A[ ]. But, as is well-known, this is just plain wrong: arrays are invariant in their element type because reading is covariant and writing is contravariant. For example, the following passes Java's static typechecker and verifier:

```
static void update(A[] arg) {
      arg[0] = new A();
}
…
B b = new B();
B[] arr = {b};
update(arr);
B b2 = arr[0];
…
```

The update method is fine: it takes an array of As and puts an A into it. The main body makes an array of Bs and passes it to a method expecting an array of As. Since B extends A, this is allowed. But then we try to get a B out of our array of Bs, which should be OK, but we get an A instead, which is *not* a B. Oh dear. To avoid this unsoundness, Java has to insert a runtime type test on all array update operations (so the above actually throws a runtime exception).

The reason Java's designers did this foolish thing is that they wished to be able to write methods such as a general sort routine that is polymorphic in the element type. But the correct way to do such a thing is to have parametric polymorphism in the language in the first place.

Dynamic type checks and casts are ugly and inefficient

Java has two instructions relating to checking and casting types at runtime. The **instanceof** instruction returns a flag saying whether an object is of a particular type. The **checkcast** instruction casts the object to a given type, throwing an exception if it cannot be so coerced. (Note that if successful, this is actually a noop, the "coercion" only happens at the level of the verifier.) Particularly in the absence of polymorphism, one often wishes to perform what amounts to a "**typecase**": finding out which of a number of classes an object belongs to and doing something different (and type-specific) with it in each case. Because doing ordinary control flow with exceptions is awkward (can't leave things on the stack) and slow, one ends up needing to use *both* of these instructions repeatedly to get the desired effect:

```
      dup
      instanceof Class1
      ifeq l2
      checkcast Class1
      .. code to deal with first case
l2:   dup
      instanceof Class2
      ifeq l3
      checkcast Class2
      .. code to deal with second case
l3:   .. etc.
```

which is clearly pretty dismal. Two possible improvements suggest themselves. One is to leave the instructions as they are, but to make the verifier or JITter sophisticated enough to detect that if execution drops through the first branch, then the object on the top of the stack is necessarily an instance of Class1, so the **checkcast** instruction is redundant. I have read that Sun's HotSpot is hardwired to detect and optimize exactly this situation. A more elegant solution is to replace or augment the two instructions with a proper **typecase** instruction using a constant table of class names and instruction numbers similar to that used for a **tableswitch**.

No closures or function types

A predictable criticism from a functional programmer, I suppose! This could be regarded as a criticism of the Java language, rather than the JVM, since one *can* compile closures using classes and objects, as we did for MLJ. Java's designers belatedly recognized the need for some kind of lexically-scoped first-class function and so introduced inner classes to the language, which are compiled by translation into ordinary classes and objects.

Nevertheless, for reasons of both efficiency and modularity, it would be preferable to have some form of first-class function built into the JVM and its type system. Compiling functions using objects involves many classes, since one typically requires one abstract class (or interface) for each function *type* (declaring, for example, an appropriate **apply** method) plus one subclass (or implementing class) for every closure of that type appearing in the program. The classes/interfaces corresponding to function types are wasteful and hard to reuse, unless their names are chosen as a uniform mangling of the underlying function type. If the VM understood those names as actual types, then one could do without the class/interface altogether. Similarly, the overhead of a class definition for each first-class function is rather excessive. (MLJ works hard to share these classes between several functions, but this uses a global analysis.)

Smalltalk includes closures (called blocks), which are used to express basic control structures such as conditionals as well as for "real"' higher-order programming. The code for a block lives inline in the code for the method in which it is defined. An activation record for an invocation of a block includes a pointer to the activation record of its defining method (or block, in the case of nested blocks): this gives both read and write access to free variables from the enclosing scope. Since a block may outlive a call to its defining method, activation records cannot always be strictly stack-allocated (a simple-minded implementation keeps *all* frames on the garbage-collected heap). A slightly bizarre feature of blocks, necessitated by their use for general control structures, is that an invocation of a block may not only return its own value, but can directly cause a return from its defining method. This is ugly, to say the least, since it is then possible to attempt to return more than once from the same method call! (This situation is trapped and causes a runtime error.) Blocks are proper first-class objects, being instances of the class BlockContext – questions of argument and result types do not arise, as Smalltalk does not have static types.

Java's inner classes allow the definition of a new class within another class or method. The class may be anonymous in the case where one immediately creates a new instance of it. Methods within the inner class may access (static or non-static) fields of the enclosing class, but can only access the values of **final** (immutable) local variables of an enclosing method. This restriction allows inner classes to be translated into normal classes without requiring frames to be heap-allocated because the values of immutable variables may be copied. Inner classes are quite heavyweight compared to functions but only 30% of their uses in Sun's standard library have more than one method.

Microsoft's VJ++ adds delegates to the language and VM. Delegates are lighter-weight than inner classes, both syntactically and in terms of implementation – they essentially make a single method (static or non-static) into a first-class object. One defines a new delegate class per function type and can then create instances of that class from arbitrary methods of the appropriate type. A delegate object contains a reference to the class or instance whose method it objectifies, and so has access to fields; it has no other free variables or local state.

How could one improve support for first-class functions in the JVM? The "morally correct" thing to do is to adopt a mechanism like that of Smalltalk, with a classical static chain and stack frames being stored on the heap (although without the possibility of closures causing a non-local return from their enclosing

methods). It is possible to optimize an implementation so *most* frames are stack allocated, and heap allocation is only used when a closure which refers to free variables from its enclosing context outlives that context. Notwithstanding the other advantages of keeping stack frames on the heap (continuations and deep stacks) however, I believe something slightly less radical could also be a useful enhancement to the JVM. For implementing ML-like languages, it is not necessary to mutate values held in enclosing frames, as the only mutable values are **ref**s and arrays, which generally live on the heap. Hence, when a closure is created, the values of its free variables may freely be *copied* into the closure object. For Java-like languages, it is not clear that being able to create closures able to access local variables of their enclosing context by reference is often useful. So the following alternative to delegates seems to make sense:

1. Add function types to the type system of the JVM, so that separate named class definitions for delegates, interfaces or abstract base classes containing a single method are not required. Function types should be treated similarly to array types – they are reference types coercible to and from **Object** but may not be subclassed, implement interfaces, etc. They should be subtyped according to the usual contra-co rule.
2. Allow functions to be created as instances of special "closure classes". These have a single constructor, which always simply bundles its arguments into instance fields, a single virtual method with a fixed (and therefore redundant) name and no static fields. Instances of closure classes have function types, rather than standard reference types.

In a Java-like language, I certainly wouldn't imagine closure classes appearing explicitly. They would be generated by the compiler from anonymous lambda abstractions in the source. Having both function types and parametric polymorphism is, of course, particularly useful (the standard imperative example being a polymorphic array sort taking, for any A,an array of As and a comparison function for As).

No way of getting the carry bit for integer arithmetic

Standard ML specifies that arithmetic overflow should raise an exception. Java does not. This is certainly wrong in terms of the general philosophy of the Java language, but the worst thing is that there is no way to check for overflow in the bytecodes even if you want to. Doing it by hand just with the standard arithmetic operators is of course possible, but *extremely* inefficient. This was the only significant place in which we were forced to depart from the official definition of SML in our compiler.

Floating point doesn't quite match IEEE 754

Not really my area, but the JVM is apparently missing error flags, rounding modes and the double extended type. Limitations of Java's floating point support have been the subject of much discussion and it looks as though there will be some changes. See http://java.sun.com/people/jag/FP.html for some of Gosling's views on the issue. There is a working/pressure group called Java Grande which is heavily pushing Sun to make changes to Java to make it more suitable for scientific computing http://www.javagrande.org. See also http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf for an amusing and slightly scattergun rant on the subject.

Note: The recently-released Java 1.2 does introduce some support for extended precision formats and a new modifier bit on methods, though I haven't yet found a coherent account of exactly what changes have been made.

Performance

Although modern JITs are much faster than the first interpreters, JVM performance is still pretty poor. For first-order code which doesn't do very much allocation, our compiler produces code which (on x86, anyway) is competitive with SML/NJ, a popular native code compiler for SML. For code which does a significant amount of heap allocation we do much worse, despite doing transformations which mean we often allocate fewer objects. Allocation and garbage collection in JVMs is still some way from the state of the art (though Sun has been promising great things for their HotSpot VM for some time).

We also found that programs which make use of exceptions as a control-flow mechanism run very slowly. Just to illustrate, here are some performance numbers for our system – note the relatively poor results for Knuth-Bendix and Logic, particularly on the MS JVM. Both of these benchmarks do a lot of allocation and throw and catch many exceptions.

Benchmark	MLJ 0.1m + JIT:		Moscow ML 1.43	SML/NJ 110
	Microsoft build 3154	Sun JDK 1.2		
BoyerMoore	3.0	♣ 5.8	2.1	0.6
FFT	♦ 15.3	12.2	441.6	28.6
KnuthBendix	63.1	♥ 29.7	10.1	2.4
Life	4.3	6.5	38.3	1.7
Logic	136.3	40.3	55.3	11.9
Mandelbrot	25.8	10.0	322.6	41.9
Nfib	0.8	0.9	8.2	1.3
Quicksort	5.8	16.6	22.4	0.9
Raytrace	7.1	10.5	28.5	7.9

♣ JIT turned off manually because bug in Symantec JIT causes crash

♥ Internally detected bug in Symantec JIT causes JIT to be disabled automatically

♦ Microsoft JIT bug causes incorrect results

Reliability

As indicated by the notes on the previous table, we also found significant bugs in JVMs, particularly in JITs. Whilst recent interpreters are fairly reliable, there is still no JIT that runs all our standard tests correctly.
The main problem seems to be that Java compilers only produce very simple, stylized code sequences. A compiler which produces valid bytecodes which do not look like those from a Java compiler reveals many JIT bugs which have not been found by testing processes which only use the output of Java compilers.

Method size limit

There is a limit of 64K on the size of methods (or at least, the length of branches). Whilst Java programmers might be unlikely to write methods this large, optimizing compilers certainly produce them. It is surprisingly tricky to prevent a compiler from generating large methods, since the optimizing transformations take place on an intermediate representation from which one cannot easily calculate the size of the resulting bytecode. Furthermore, the intermediate representation may transiently represent a large method, which then becomes smaller as further transformations are done. Hence one is reduced to a very ugly bytecode to bytecode transformation in the backend to split large methods.

Initialization of static data

There is no good way to initialize static data, particularly arrays. One has to emit code that inserts all the elements into the array one by one. This is slow, wastes space and can easily run into the method size limit when the array is large, as for example in the tables for some automatically generated parsers. The alternative is to use serialization, which amounts to looping round reading the bytes from a file and putting them into the array. At least for arrays of primitive types, it would be nice to have a way of just blitting the data in.

Ridiculous subroutines

The JVM has very peculiar instructions for "subroutines". These are not at all general purpose, but are just used to avoid code duplication when certain specific source constructs (**try**/**finally**) are compiled. The **jsr** instruction puts a return address on stack. The **astore** instruction is overloaded to allow that to be stored in a local variable (**aload** is not so overloaded). The **ret** instruction returns to the address stored in a local variable.

Subroutines cannot be usefully recursive because of the requirement for per-method stack depth to be statically limited. Furthermore, the verification requirements are rather strict. This odd mechanism has provided plenty of material for theoreticians, because it is extremely awkward to verify, but experiments show that it is used so infrequently that it serves no useful purpose. It should be either removed or replaced with something more general.

No true multidimensional arrays

Arrays of arrays are not a real substitute. Although there is a bytecode to create an array of arrays all at once, it has to be indexed twice to access elements, the internal arrays are first-class and one cannot tell if they are all the same size. This prevents a compiler or JIT, unless it is particularly clever, from organizing the layout efficiently (even deciding between row and column major order) or detecting that it can omit some bounds checks.

No general immutable values

There are plenty of applications for a general facility to define classes or types of immutable values with structural equality. Apart from advantages at the language level, including these in the VM would allow compilers and JITs great freedom because they are known not to be **null**, can be copied and may be represented either boxed or unboxed. Adding such a general facility would address a number of more specific criticisms of the JVM, such as its lack of efficient built-in support for complex numbers. Indeed, Gosling's proposal for improved Java numerics includes immutable value types.

One could add immutable values in a typical object-oriented style using a special kind of class definition. Alternatively, one could actually add proper products (tuples) to the type system. (Of course, tuple types are just the thin end of the type-theorists wedge – building in coproduct types (tagged unions), of which enumerations are just a special case, would be great too…)

No multiple return values

One often wants to return more than one result from a method. Of course, this can be achieved by creating a class with several instance variables of the appropriate type and then returning a new instance of this class, which is unpacked by the caller. But this involves creating a new class and, at least if implemented straightforwardly, is inefficient.
The simplest solution is just to add multiple returns as a primitive notion. Alternatively, if one had built-in tuples (see general immutable values above) then one might use those and leave it up to the JIT to decide how to return them (stack, heap, registers) in each case.

Throws declarations

The Java language requires that methods explicitly list the exceptions which they might throw. This information is stored in compiled class files and checked by the compiler, but is not actually used by the verifier and so is not strictly required to be accurate. I don't recall if our experiments indicated that any JIT made use of the information. I don't think so, though clearly, if one thought the information was accurate then it could in theory be exploited by a JIT. This is a slightly odd situation and it is rather wasteful to include this information in code sent over the network if it gets ignored at the other end. I think the original intention was that the verifier *would* eventually check exception declarations but since it didn't at first, it now probably never will.

There's nothing actually *wrong* with exception declarations as currently stand, but they are an example of the uneasy conflation of object code with module signatures. They also show how, even with an all new from the ground up project, early design decisions (which one expects to reconsider later) become effectively frozen very quickly.

Non-uniform tests and branches

The JVM has a number of comparison and conditional branch instructions. For the basic integer type there is a family of binary comparison-and-branch instructions **ificmp<cond>** which take two integers from the stack and branch according to whether one is less than, equal, etc. the other. There is also a family of **if<cond>** instructions which branch if an integer comparison with zero succeeds. For non-integer primitive types (long, double, float), there are binary comparison operators which compare two values from the top of the stack and push an integer result of –1,0 or 1 which can then be tested with **if<cond>**. For reference types, there is no simple comparison, just two **ifacmp** compare-and-branch instructions. This seems a not unreasonable design, but we found it slightly awkward to make use of the ificmp instructions in compiling non-trivial conditional expressions without overcomplicating our code generator. This is because the integer case is treated differently from the other base types. So we tended to use the more uniform if<cond> form, but then the fact that there is no icmp instruction was sometimes annoying: using subtraction isn't quite the same as it complicates use of logical operators on the results of multiple tests. We would have preferred to have icmp and acmp available and could have managed without ificmp.
Of course, this is a rather trivial criticism, but a little extra uniformity here would have saved us a day or two of surprisingly annoying fiddling with the code generator.

Acknowledgments

Thanks to Luca Cardelli, Andy Gordon, Andrew Kennedy, Simon Peyton Jones and Don Syme for useful discussions and comments.