Check for updates

Java Intermediate Bytecodes

ACM SIGPLAN Workshop on Intermediate Representations (IR '95)

James Gosling <jag@eng.sun.com>
Sun Microsystems Laboratories

Java† is a programming language loosely related to C++. Java originated in a project to produce a software development environment for small distributed embedded systems. Programs needed to be small, fast, "safe" and portable. These needs led to a design that is rather different from standard practice. In particular, the form of compiled programs is machine independent bytecodes. But we needed to manipulate programs is ways usually associated with higher level, more abstract intermediate representations. This lets us build systems that are safer, less fragile, more portable, and yet show little performance penalty while still being simple.

Introduction

The project that produced Java started in 1991. Its goal was to produce a software environment for small distributed embedded systems. The requirements imposed on this system were based on the need to cope with heterogeneous networks and to build long-lived reliable systems. In particular, it was necessary for compiled software to be shipped around the network and executed on whatever CPU it landed on. Once the code gets to the client CPU, the code needs to adapt to whatever versions of the classes it uses that happen to be there, and the receiving system has to have a reasonable amount of faith that the code is safe to run. All while being as fast and small as possible.

Originally, we intended to be politically correct and just use C++. But a number of serious problems arose as a consequence of the requirements on the system. Many of the problems could be addressed with some compiler technology, in particular, by using a different intermediate representation for compiled programs. This paper covers our solutions to these problems.

The solution we chose was to compile to a byte coded machine independent instruction set that bears a certain resemblance to things like the UCSD Pascal P-Codes[Bowles78]. While compact and amenable to interpretation, such a

IR '95, 1/95,
San Francisco, California, USA
Copyright © 1995 ACM

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

†. Java used to be known as "Oak" when it was just an internal project.

representation is, in general, unsuitable for higher level, more abstract, manipulation. We applied a number of twists: there is an unusual amount of type information, there are restrictions on the use of the operand stack, and there is a heavy reliance on symbolic references and on-the-fly code rewriting.

A Code Sample

To give you a taste of what compiled code looks like, consider this source fragment:

```
class vector {
  int arr[];
  int sum() {
    int la[] = arr;
    int S = 0;
    for (int i=la.length; --i>=0;)
      S += la[i];
    return S;
  }
}
```

It compiles to:

```
    aload_0        Load this
    getfield #10   Load this.arr
    astore_1       Store in la
    iconst_0
    istore_2       Store 0 in S
    aload_1        Load la
    arraylength    Get its length
    istore_3       Store in i
A:  iinc 3 -1      Subtract 1 from i
    iload_3        Load i
    iflt B         Exit loop if <0
    iload_2        Load S
    aload_1        Load la
    iload_3        Load i
    iaload         Load la[i]
    iadd           add in S
    istore_2       store to S
    goto A         do it again
B:  iload_2        Load S
    ireturn        Return it
```

Type information

This example is pretty straightforward. One of the slightly odd things about it is that there is somewhat more type information than is strictly necessary. This type information is often encoded in the opcode. For example, there are both aload and iload opcodes whose implementations are identical, except that one is used to load a pointer, the other is used to load an integer. Similarly, the getfield opcode has a symbol table reference. There is type information in the symbol table.

In most stack based instruction sets, you can do pretty much anything with the stack and local variables. In the Java bytecode there is an important restriction: conceptually, at any point in the program each slot in the stack and each local variable has a type. This collection of type information is called the *type state* of the execution frame. The important property is that this type can be determined *statically* by induction. As you read through a block of instructions, each instruction pops and pushes values of particular types. Instruction definitions are required to have the following inductive property:

> *Given only the type state before the execution of the instruction, the type state afterwards is determined.*

Given a straight-line block of code, starting with a known stack state, the type state of each slot in the stack is known. For example:

```
iload_1        Load integer variable, stack type state=I
iconst_5       Load integer constant, stack type state=II
```

`iadd` *Add two integers producing an integer, stack type state=I*

A number of stack based codes, like Smalltalk [Goldberg83] and PostScript [Adobe85] do not have this property. For example the definition of the PostScript **add** operator explicitly states “If both operands are integers and the result is within integer range, the result is an integer, otherwise the result is a real”. In many situations, this dynamic type behavior is considered to be an advantage, but in our situation, it is not. The most important ingredient in a simple implementation is a simple specification.

In conjunction with this we require that:

> *When there are two execution paths into the same point, they must arrive there with exactly the same type state.*

This means, for example, that bytecode generators cannot write loops that iterate through arrays, loading each element of the array onto the stack, effectively copying the array onto the stack. Why? Because the flow path into the top of the loop will have a different type state than the branch back to the top.

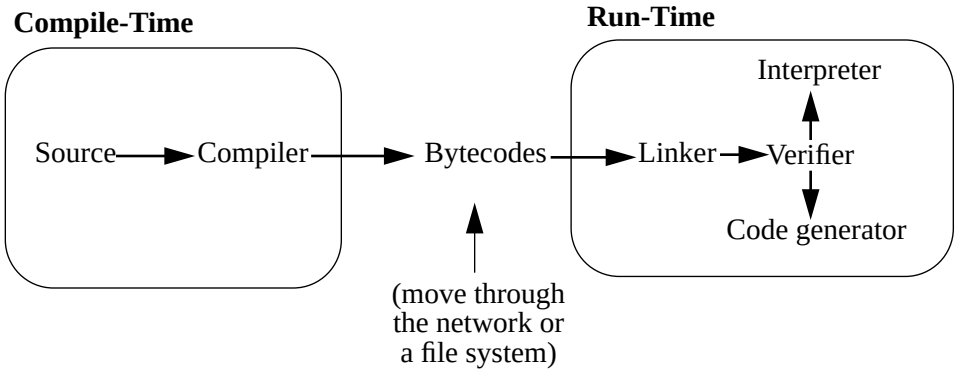
Since all paths to a point are required to arrive with the same type state, then the type state from *any* incoming path can be used to do further manipulations.

These restrictions have a number of important consequences.

Checkable

The most important consequence is that there are a number of properties that can be checked statically. The simplest is operand stack overflow and underflow: the length of the stack portion of the type state is the depth that the operand stack will have when that chunk of code is executed. For interpreters that are willing to trust the compiler that generated the bytecode, the interpreter can skip doing stack checks on each bytecode execution. In other situations, like PostScript, stacks have to be checked. In our case, we couldn’t afford to have every bytecode check the stack depth as it executes, but on the other hand we couldn’t afford to trust that loaded code came from a correct compiler.

The last phase of the bytecode loader is the *verifier*. It traverses the byte codes, constructs the type state information, and verifies the types of the parameters to all the opcodes.



The verifier acts as a sort of gatekeeper: imported code is not allowed to execute by any means until after it has passed the verifier's tests. Once the verifier is done, a number of important properties are known:

- There are no operand stack overflows or underflows.
- The types of the parameters of all opcodes are known to always be correct.
- No illegal data conversions are done, like converting integers to pointers.
- Object field accesses are known to be legal (i.e. private/public/... are rechecked by the verifier).

Knowing these properties makes the interpreter much faster: it doesn't have to check anything. There are no operand type checks and no stack overflow checks. The interpreter can do this without compromising reliability.

These properties also provide a foundation for the security of the system: pointers can be treated essentially as capabilities: Applications cannot forge them, they cannot get around them, and all the access restrictions are enforced. So in higher level software, you can trust that a private variable really is private, that no evil piece of application code is doing some magic with casts to extract a credit card number from the billing software.

Fragile superclasses

One of the big problems with using C++ in a commercial situation is sometimes called the *fragile base class problem*. Say company A sells a library which defines class CA, then company B builds a product which uses that class. In doing so they define a class CB, which is a subclass of CA. The way that most C++ compilers are implemented, the code generated for CB will have integers hardwired into it that reference the contents of an instance at fixed offsets. If company A releases a new version of CA which changes the number of instance variables or methods, then B will have to recompile CB to correct all the offsets that are now different.

This becomes a nightmare when you take the customer into account: they have a copy of the software that they bought from B. When CA is a shared library that is used in many products, the end user is likely to get new versions

of it independent of new versions of CB. So if they upgrade the library, existing applications which use it will break. They would have to go back to B and get a new release.

In practice, this doesn't happen because software developers don't let it happen: C++ style object oriented programming is essentially never used in public interfaces to widely shared libraries. And when it is used, it is used very carefully.

This problem essentially defeats the whole "software IC" reusability model in object oriented programming. By fixing this, as we have in Java, the software IC model can be made to really work.

The way that Java gets around the fragile base class problem is simply to use symbolic references. For example, in the code fragment at the beginning of the paper the getfield opcode doesn't contain an offset into the object, it contains an index into the symbol table. This is a pretty standard technique. But, as usual, there's a twist: Java is essentially C++, so it is known that once a system starts executing (assuming that classes can't be dynamically unloaded and reloaded, which we can get around...) the offset into the object doesn't change. When the getfield opcode is executed the interpreter looks up the symbol, discovers it's offset, then rewrites the instruction stream to be a quick getfield opcode with the exact offset. This can be executed very quickly. This technique of rewriting symbolic references is used pervasively.

Portability

One of the obvious benefits of using a bytecode like Java's is that compiled programs are portable: so long as the interpreter is present, programs can execute on any kind of CPU. One of the requirements for making programs portable is nailing down, in the language specification, all those little grey areas in language specifications that are often left "implementation specific". Things like evaluation order and "what does *int* mean?". In general, we opted for semantics that we could make small and fast, which is generally consistent with the C/C++ tradition. We avoided doing things like defining arithmetic to be infinite precision. There are a number of hardware trends that we exploited and cast into the definition of the system. For example, many systems have very loose definitions of floating point arithmetic. But implementation of the IEEE 754/854 specifications for floating point semantics have become almost universal in modern hardware, so we explicitly specify that the language floating point semantics follow the IEEE specification.

Translation to machine code

The statically determinable type state enables a very powerful performance technique: simple on the fly translation of bytecodes into efficient machine code. Because the types of all arguments are statically determinable in a simple way, the bytecodes can be simply translated into machine code: no dynamic type checks or sophisticated inferences have to be done. Translation is just matter of reading the "iadd" bytecode and emitting an "add ra, rb, rc" instruction.

Many other systems, such as Smalltalk [Duetsch84] and Self [Chambers92] do on the fly compilation of code fragments. They are very sophisticated

systems that put a lot of effort into caching or deducing type information and compiled code fragments. The cost of this sophistication is complexity. In contrast, our goal was to be as *simple* as possible. The language we compile has much more static information and this static information is carried through to the byte codes. We also use a restricted bytecode set to allow a lot of information (like type codes) to be trivially re-derived rather than carried along. What is interesting here is not sophisticated technology, it is a set of simplifying choices.

The representation of the operand stack is a tricky issue. Just being straightforward and representing it directly at runtime isn't very efficient. Rather, what we do is to think of compiling as watching what the interpreter would do, and taking notes. There is a simple data structure that represents the state of a stack slot:

```
class SlotState {
  int RegisterNumber;
  int IntValue;
};
```

The value at a particular level in the stack is the sum of a register and an integer. More complex state representations are possible, this one is just barely complex enough to be instructive.

The byte code to machine code translator iterates over the bytecodes doing bytecode specific processing on each. Many of the bytecodes generate no instructions, but simply manipulate the description of the stack state.

For example, the `pop` opcode merely decrements the reference count on the register (there's a fictitious register used to represent no-register-needed) and decrements the stack pointer.

The "integer constant" opcode pushes a new `SlotState` onto the stack that has an `IntValue` taken from the instruction stream and no register.

The "load local" opcode has two cases: if the local variable is in a register, it pushes a new `SlotState` that refers to that register and increments its reference count. If it is not in a register it has to convert it to a form representable by a `SlotState`: it allocates a register and emits a "load" instruction.

The "integer add" opcode looks at the two `SlotState` descriptions and emits something appropriate.

So the sequence "load local; load constant; integer add" usually ends up emitting one instruction, since the first two bytecodes generate nothing and just manipulate the stack slot state and eventually get folded into the integer add instruction.

This very simple approach works because the stack state is statically deterministic.

Another way to think of this is that the operand stack is a source of names. One traditional intermediate representation used by compilers is as a *three*

address code [Aho86] where the program is reduced to “a sequence of statements of the general form:

x := y *op* z

Where x, y and z are names, constants, or compiler-generated temporaries. *Op* stands for an operator”. Bytecodes are a compressed form of the same thing, where the operands are implicitly derived from the stack.

The tyranny of the instruction set

There are a small number of instruction set architectures, like x86 and SPARC, and a set of implementations of each. These implementations form successive generations, perpetuating the instruction set architecture for a long time in order to preserve the usefulness of existing software. A computer is useless, after all, if there are no applications that run on it. There are two problems with this process.

One problem is that the consistancy of instruction sets from one generation to the next is really fiction. As each of these instruction set architectures go from one generation to the next the performance trade-offs change dramatically. The fastest sequence to accomplish a task in one generation may be the worst in the next.

The other problem in successive generations is that compatibility makes the CPUs much more complex and usually slower. In general, a group of electrical engineers designing a new hot chip will be able to get much better performance if they aren’t constrained by the instruction set architecture. Using an architecture neutral bytecode technique like that in Java, combined with moderatly reasonable on the fly machine code generation that is targeted at exactly the machine being executed on, can lead to net better performance when compared to a sophisticated optimizer that is trying to target an architecture family. If the CPU were designed without constraints, all the performance bottlenecks of backward compatibility would be eliminated. In other words, a reasonably competent code generator targeted precisely to the CPU at hand, coupled with a CPU whose performance was optimized for the implementation technology without historic compatibility constraints would beat a brilliant code generator targeted at a family of CPU chips whose designers had to expend chip area on compatibility rather than performance. Unfortunately, this is an essentially untestable conjecture.

Performance

The current system running interpreted on a SparcStation 10 (roughly equivalent to a 486/50) will execute the empty for loop:

```
for(i = 900000; --i>=0;);
```

in 1 second (i.e. we’re getting about 900K trips around the loop per second). Add a call to an empty method and we get about 300K trips per second. After running this through the machine code translator, the numbers get better by a factor of almost 10, making the performance essentially indistinguishable from C.

Conclusions

When you add the deterministic stack type-state restriction to a fairly conventional bytecode intermediate representation it becomes possible to use such a low-level representation in situations where higher-level, more abstract representations are often used. This allows the bytecoded program to be compact. It can be directly interpreted efficiently or translated to machine code or analyzed statically. The implementation of the these manipulations can be simple, fast and small.

Bibliography

[Adobe85] Adobe Systems Incorporated, “PostScript Language Reference Manual”, *Addison Wesley.*

[Aho86] Aho, Alfred V., Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, *Addison Wesley.*

[Bowles78] Bowles, Kenneth L, “UCSD Pascal”, *Byte*. 46 (May)

[Chambers92] Chambers, Craig, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA ‘89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[Duetsch84] Deutsch, L. Peter and Alan Schiffman, “Efficient Implementation of the Smalltalk-80 System.” *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984

[Goldberg83] Goldberg, Adele and David Robson, “Smalltalk-80: The Language and its Implementation”, *Addison Wesley.*