

# Oak Intermediate Bytecodes

## **A summary of a paper for the ACM SIGPLAN Workshop on Intermediate Representations (IR '95)**

James Gosling <jag@eng.sun.com>  
100 Hamilton Avenue 3rd floor  
Palo Alto CA 94301

*Oak is a programming language loosely related to C++. It originated in a project to produce a software development environment for small distributed embedded systems. From this, a rather unique set of requirements arose which led to a design that is rather different from standard practice. In particular, the distributed form of compiled programs is machine independent bytecodes. The definition of the bytecode set has some twists that allow it to be used to derive information normally 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 and are still simple.*

The project that produced Oak started in 1991. 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 these issues; issues driving some of the language changes will be addressed in another paper.

The requirements were based on needs to cope with heterogeneous networks and 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 there it needs to adapt to whatever versions of the classes it uses 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 as possible.

The solution we settled on was to compile to a byte coded machine independent instruction set that bears a certain resemblance to things like the UCSD Pascal P-Codes. There are a number of twists though: 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 a[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. In the oak bytecode there is an important restriction: Conceptually, at any point in the program each slot in the stack has a type. The important property is that this type can be determined *statically*. As you read through a block of instructions, each instruction pops and pushes values of particular types. From the opcodes and parameter information the type can be determined. For a straight-line block of code, starting with a known stack state, the type state of each slot in the stack is known. A part of this restriction is that when there are two execution paths into the same point, they must arrive there with exactly the same stack 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.

This restriction has a number of important consequences.

## Checkable

The most important consequence is that there are a number of properties that can be checked statically. This simplest is operand stack overflow and

underflow: the length of the type state is the depth that the operand stack will have when that chunk of code is executed.

Another is that the types of the arguments to an opcode can be checked statically: you can tell that the operands to the integer add opcode (for example) will be integers.

## **Safety**

This checkability property is used to provide a measure of safety: bytecodes can be loaded from some foreign host and you can test that they satisfy certain criteria. This can be done with *no* runtime performance impact: when bytecodes are being interpreted, the interpreter doesn't have to do overflow checks on the operand stack because it can trust that the code doesn't do something bad. Similarly, the integer add instruction just adds two integers: no runtime type check is necessary.

## **Fragile superclasses**

One of the big problems with using C++ in a commercial situation is something 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 essentially defeats the whole "software IC" reusability model in object oriented programming. By fixing this, as we have in oak, the software IC model can be made to really work.

## **Symbolic references**

The way that oak gets around this 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: oak 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 such as this 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 this work is nailing down, in the language specification, all those little grey areas in language specs that are often left “implementation specific”. Things like evaluation order and “what does *int* mean?”.

## Translation to machine code

The statically determinable type signatures enable 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 translated into machine code pretty simply: no dynamic type checks or sophisticated inference have to be done. It really is pretty much a just matter of reading the “iadd” bytecode and spitting out an “add ra, rb, rc” instruction.

The representation of the operand stack is a tricky issue. Just being straightforward and representing it directly 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, and 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.

## **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 picture:

One is that it's really fiction: in all of these instruction set architectures as you go from one generation to the next the performance tradeoffs change dramatically. The fastest sequence to accomplish a task in one generation may be the worst in the next.

The Other is that this compatibility makes the cpus much more complex and usually slows them down. 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 instruction set architecture constraints. Using an architecture neutral bytecode technique like that in oak, combined with moderately 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. And if the CPU were designed with this in mind, all the performance bottlenecks of backward compatibility would be eliminated.

## **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 (ie. 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 them 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 and directly interpreted efficiently while at the same time allowing static analysis and machine code generation to occur.