Lecture Notes on
Programs as Data: The JVM

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 9
September 21, 2010

1 Introduction

A recurring theme in computer science is to view programs as data. For example, a *compiler* has to read a program as a string of characters and translate it into some internal form, a process called *parsing*. You learn about parsing in Assignment 3. Another instance are first-class functions, which you will study in great depth in 15–150, a class dedicated to functional programming. When you learn about computer systems in 15–213 you will see how programs are represented as *machine code* in binary form.

In this lecture we will take a look at a *virtual machine*. In general, when a program is read by a compiler, it will be translated to some lower-level form that can be executed. For C and C0, this is usually machine code. For example, the `cc0` compiler you have been using in this course translates the input file to a file in the C language, and then a C compiler (`gcc`) translates that into code that can be executed directly by the machine. In contrast, Java implementations typically translate into some intermediate form called *byte code* which is saved in a class file. Byte code is then *interpreted* by a virtual machine called the JVM (for Java Virtual Machine). So the program that actually runs on the machine hardware is the JVM which interprets byte code and performs the requested computations.

Using a virtual machine has one big drawback, which is that it will be slower than directly executing a binary on the machine. But it also has a number of important advantages. One is *portability*: as long as we have an implementation of the virtual machine on our target computing platform,

we can run the byte code there. So we need a virtual machine implementation for each computing platform, but only one compiler. A second advantage is *safety*: when we execute binary code, we give away control over the actions of the machine. When we interpret byte code, we can decide at each step if we want to permit an action or not, possibly terminating execution if the byte code would do something undesirable like reformatting the hard disk or crashing the computer. The combination of these two advantages led the designers of Java to create an abstract machine. The intent was for Java to be used for mobile code, embedded in web pages or downloaded from the Internet, which may not be trusted or simply be faulty. Therefore safety was one of the overriding concerns in the design.

As a side remark, at the time the C language was designed, machines were slow and memory was scarce compared to today. Therefore, *efficiency* was a principal design concern. As a result, C sacrificed safety in a number of crucial places, a decision we still pay for today. Any time you download a security patch for some program, chances are a virus or worm or other malware was found that takes advantage of the lack of safety in C in order to attack your machine. The most gaping hole is that C does not check if array accesses are in bounds. So by assigning to $A[k]$ where k is greater than the size of the array, you may be able to write to some arbitrary place in memory and, for example, install malicious code. In 15–213 *Computer Systems* you will learn precisely how these kind of attacks work, because you will carry out some of your own!

In C0, we spent considerable time and effort to trim down the C language so that it would permit a safe implementation. This makes it marginally slower than C on some programs, but it means you will not have to try to debug programs that crash unpredictably. We will introduce you to all the unsafe features of C, when the course switches to C later in the semester, and teach you programming practices that avoid these kinds of behavior. But it is very difficult, even for experienced teams of programmers, as the large number of security-relevant bugs in today's commercial software attests. One might ask why program in C at all? One reason is that many of you, as practicing programmers, will have to deal with large amounts of legacy code that is written in C or C++. As such, you should be able to understand, write, and work with these languages. The other reason is that there are low-level systems-oriented programs such as operating systems kernels, device drivers, garbage collectors, networking software, etc. that are difficult to write in safe languages and are usually written in a combination of C and machine code. But don't lose hope: research in programming language has made great strides of the last two decades, and

there is an ongoing effort at Carnegie Mellon to build an operating system based on a safe language that is a cousin of C. So perhaps we won't be tied to an unsafe language and a flood of security patches forever.

2 A Stack Machine

The JVM is a *stack machine*, which is why we are introducing it as this point in the course, just after we have covered stacks. This means that the evaluation of expressions uses a stack, called the *operand stack*. It is written from left to right, with the rightmost element denoting the *top* of the stack.

We begin with a simple example, evaluating an expression without variables:

$$(3 + 4) * 5/2$$

In the table below we show the *instruction* on left, in textual form, and the operand stack *after* the instruction on the right. We write '$\cdot$' for the empty stack.

Instruction	Operand Stack
	$\cdot$
`bipush 3`	3
`bipush 4`	3, 4
`iadd`	7
`bipush 5`	7, 5
`imul`	35
`bipush 2`	35, 2
`idiv`	17

The process of going from the expression to the instructions is what a compiler would normally do. Here we just write the instructions by hand, in effect simulating the compiler. The important part is that executing the instructions will compute the correct answer for the expression. We always start with the empty stack and end up with the answer as the only item on the stack.

In the JVM instructions are represented as bytes. This means we only have 256 different instructions. Some of these instructions require more than one byte. For example, the `bipush` instruction requires a second byte for the number to push onto the stack. The following is an excerpt from the JVM reference, listing only the instructions needed above.

```
0x10 bipush <c>    S     -> S,c
```

```
0x60 iadd          S,x,y -> S,x+y
0x68 imul          S,x,y -> S,x*y
0x6C idiv          S,x,y -> S,x/y
```

On the right-hand side we see the effect of the operation on the stack S. Using these code we can translate the program into code.

Code	Instruction	Operand Stack
		.
0x10 0x03	bipush 3	3
0x10 0x04	bipush 4	3, 4
0x60	iadd	7
0x10 0x05	bipush 5	7, 5
0x68	imul	35
0x10 0x02	bipush 2	35, 2
0x6C	idiv	17

In a binary file that contains this program we would just see the bytes

```
10 03 10 04 60 10 05 68 10 02 6C
```

and it would be up to the JVM implementation to interpret them appropriately. The file format we use is essentially this, except we don't use binary but represent the hexadecimal numbers as strings separated by whitespace, literally as written in the display above.

3 Local Variables

Next, we add the ability to handle function arguments and local variables to the machine. For that purpose, a function has an array V containing *local variables*. We can push the value of a local variable onto the operand stack with the `iload` instruction, and we can pop the value from the top of the stack and store it in a local variable with the `istore` instruction. Initially, when a function is called, its arguments $x_1, \ldots, x_n$ are stored as local variables $V[1], \ldots, V[n]$. The variable 0 is unused in our version of the machine; for Java it holds the reference to `this`, the object on which a method is invoked.

So assume we want to implement

```
int mid (int lower, int upper) {
  int mid = lower + (upper - lower)/2;
  return mid;
}
```

Here is a summary of the instructions we need

```
0x15 iload <i>     S -> S,V[i]
0x36 istore <i>    S,v -> S        (V[i] = v)
0x64 isub          S,x,y -> S,x-y
0xB1 ireturn       .,v -> .
```

Notice that for `ireturn`, there must be exactly one element on the stack. Using these instructions, we obtain the following code for our little function. We indicate the operand stack on the right, using symbolic expressions to denote the corresponding runtime values. The operand stack is not part of the code; we just write it out as an aid to composing the program.

```
// V[1] = lower
// V[2] = upper
// V[3] = mid
                       // .
0x15 0x01 iload 1      // lower
0x15 0x02 iload 2      // lower, upper
0x15 0x01 iload 1      // lower, upper, lower
0x64      isub         // lower, upper-lower
0x10 0x02 bipush 2     // lower, upper-lower, 2
0x6C      idiv         // lower, (upper-lower)/2
0x60      iadd         // lower+(upper-lower)/2
0x36 0x03 istore 3     // .
0x15 0x03 iload 3      // lower+(upper-lower)/2
0xB1      ireturn      // .
```

We can optimize this piece of code, simply removing the last `istore 3` and `iload 3`, but we translated the original literally to clarify the relationship between the function and its translation.

4 Implementing JVM00

We now turn to implementing a tiny fragment of the JVM, which we call JVM00. You may consult the complete code in the file jvm00.c0. Many additional constructs in the JVM are concerned with different forms of data, such as floating point numbers, or arrays, or objects. Method calls take some thought and introduce a second form of stack, the *call stack* into the machine.

A function is represented as a struct

```
struct func {
  int max_local;                /* use V[0..max_local) */
  int max_pc;                   /* use P[0..max_pc) */
  int[] program;                /* program P */
};
```

where `max_local` is the number of local variables, `max_pc` is the number of bytes in the program, and `program` is an array of ints holding the bytecode.

It makes it easier to read files, so we assume our bytecode file format starts with two bytes, the first giving `max_pc` and the second `max_local`. The rest of the file contains the bytecode of the instructions.

We write a C0 function

```
int exec (func f, int arg1, int arg2);
```

limiting ourselves to functions of two arguments, for simplicity. More generally, we would pass the number of arguments and an array with the argument themselves of the appropriate length.

We begin the function by naming the program P, allocating an array V of local variables, allocating a new empty stack S, initializing a program counter pc to 0 and loading the arguments into the local variable array. The program counter pc always holds the next instruction to execute and therefore starts at 0.

```
int exec (func f, int arg1, int arg2) {
  int[] P = f->program;
  int[] V = alloc_array(int, f->max_local);
  stack S = s_new();
  int pc = 0;
  V[1] = arg1;
  V[2] = arg2;
  ... ?? ...
}
```

Now we go into a (potentially infinite) loop, reading, decoding, and executing instructions from the program array P. We explicitly return from the `exec` function when we find an `ireturn` instruction, escaping from the loop.

```
while (true) {
  int inst = P[pc];
  if (inst == 0x60) { push(pop(S)+pop(S),S) ; pc+=1; } // iadd
  else if (inst == 0x68) { push(pop(S)*pop(S),S) ; pc+=1; } // imul
  ... ?? ...
  else assert(false, "unrecognized instruction");
}
```

More instructions will be added at the noted place. If we don't recognize the instruction at all, we issue an error message. Let's look at the code for the `iadd` instruction (`0x60`) in detail.

Executing the code `push(pop(S)+pop(S),S)` will pop two operands from the stack, which is supposed to have the form S, x, y. In C0, expressions are evaluated from left to right[1], so we first pop y then x. The stack is now S, so we push $y+x$ back onto S, which now is $S, y+x$. This is correct, because $x + y = y + x$ in modular arithmetic. We now have to increment the program counter by 1 in order to continue execution with the next instruction on the next iteration of the loop. This is accomplished by `pc+=1`.

Multiplication works the same way. Subtraction and division are slightly tricky because the arguments are popped from the stack in the "wrong order". We can compensate for subtraction, again exploiting laws of modular arithmetic, but for division we need to bind the values to temporary variables. This solution is also portable to C, since it doesn't rely on the order of evaluation of expressions.

```
while (true) {
  int inst = P[pc];
  if (inst == 0x00) { pc+=1; }
  else if (inst == 0x60) { push(pop(S)+pop(S),S) ; pc+=1; } // iadd
  else if (inst == 0x64) { push(-(pop(S)-pop(S)),S) ; pc+=1; } // isub
  else if (inst == 0x68) { push(pop(S)*pop(S),S) ; pc+=1; } // imul
  else if (inst == 0x6C) {
    int y = pop(S); int x = pop(S); push(x/y,S); pc+=1; } // idiv
  ... ?? ...
}
```

We now show the instructions individually. A `ireturn` pops the return value from the stack (which should be empty afterwards, but we do not

[1] This is not true for C, which does not guarantee a particular evaluation order for expressions. So some of the instructions would have to be written slightly differently to be correct in C.

check this). We then return this value from the exec function.

```
if (inst == 0xB1) { return pop(S); } // ireturn
```

The bipush <c> instruction consists of two bytes. We have to push the *next* byte in the program array onto the operand stack. Then we have to increase the program counter by 2 to find the next instruction.

```
if (inst == 0x10) { push(P[pc+1],S); pc+=2; } // bipush <c>
```

The iload <i> instruction also consists of two bytes. It pushes the contents of $V[i]$ onto the operand stack, where i is the next byte in the program array. The istore <i> instruction does the opposite, popping an element from the stack and storing it in $V[i]$.

```
if (inst == 0x15) { push(V[P[pc+1]],S); pc+=2; } // iload <i>
else if (inst == 0x36) { V[P[pc+1]] = pop(S); pc+=2; } // istore <i>
```

The exec function is now the following:

```
int exec (func f, int arg1, int arg2) {
  int[] P = f->program;
  int[] V = alloc_array(int, f->max_local);
  stack S = s_new();
  int pc = 0;
  V[1] = arg1;
  V[2] = arg2;
  while (true) {
    int inst = P[pc];
    if (inst == 0x60) { push(pop(S)+pop(S),S) ; pc+=1; } // iadd
    else if (inst == 0x64) { push(-(pop(S)-pop(S)),S) ; pc+=1; } // isub
    else if (inst == 0x68) { push(pop(S)*pop(S),S) ; pc+=1; } // imul
    else if (inst == 0x6C) {
      int y = pop(S); int x = pop(S); push(x/y,S); pc+=1; } // idiv
    else if (inst == 0xB1) { return pop(S); }               // ireturn
    else if (inst == 0x10) { push(P[pc+1],S); pc+=2; }      // bipush <const>
    else if (inst == 0x15) { push(V[P[pc+1]],S); pc+=2; }   // iload <index>
    else if (inst == 0x36) { V[P[pc+1]] = pop(S); pc+=2; }  // istore <index>
    else assert(false, "unrecognized instruction");
  }
  assert(false, "should never reach this spot");
  return 0;
}
```

We should never exit the loop (unless we are returning from `exec`), so we should never reach the statement that follows it. Nevertheless, the C0 compiler insists that every function that returns a value must have an explicit return statement at its end.[2] We therefore add a `return` statement, but guard it with an `assert` statement that will always raise an error.

We can now read and run the following file `mid.hx`, which computes the midpoint. The first two bytes are the number of bytes in the program and the number of local variables.

```
10 04

15 01
15 02
15 01
64
10 02
6C
60
36 03
15 03
B1
```

You are invited the check out the code at the course web pages and hand-translate your own programs and try it out.

5 Conditional Branches and Jumps

We introduce a few more byte code instructions so we can represent more interesting programs. For example, say, we want to compile the following (incorrect!) integer square root function from the first lecture:

```
int f (int n) {
  int i = 0; int k = 0;
  while (k <= n) {
    k = k + 2*i + 1;
    i++;
  }
  return i-1;
}
```

[2] Technically, it is at the end of every control-flow path, a notion we have not introduced.

The question is how to represent the `while` loop.

At the level of the source code it promotes good programming discipline to use looping constructs such as `while` or `for`. In the byte code, though, these are mapped into conditional branches and jumps. Both of them take an *offset* as an argument rather than an absolute address of the next instruction. This makes it possible to move the code for a function to a different address without having to change its actual bytes. We represent offsets here by a single byte; in the real JVM it is two.

For the `goto` instruction, the stack is unaffected, we simply jump to the address calculated from the current address and the offset.

```
0xA7 goto <o>
```

In our implementation, we insert into our `exec` function:

```
if (inst == 0xA7) { pc += offset(P[pc+1]); }  // goto <offset>
```

The function `offset` computes a negative offset if bit 7 of $P[pc+1]$ is a 1.

```
int offset(int o) {
  if (o > 127) return o-256;
  else return o;
}
```

This interpretation is necessary so offsets can be both positive (forward jumps) and negative (backwards jumps).

There are six different conditional branch instructions, all of which have the form `if_icmp<d> <o>`, where d is as in the following table and o is the offset for the jump to be taken in case the condition is true.

```
eq  ==
ne  !=
lt  <
ge  >=
gt  >
le  <=
```

For all these instructions, the arguments x and y will be popped from the stack, but unlike the binary operators, no result is pushed back. Instead, we compare x and y with the indicated operator and jump by the given offset if the condition is true. If it is false, we continue execution with the next instruction. We show only two cases.

```
if (inst == 0x9F) {       // if_icmpeq <o>
  if (pop(S) == pop(S)) { pc += offset(P[pc+1]); } else { pc+=2; }
} else if (inst == 0xA3) { // if_cmpgt <o>
  if (pop(S) < pop(S)) { pc += offset(P[pc+1]); } else { pc+=2; }
}
```

Note the use of less-than (<) instead of greater-than (>) in the code for `if_cmpgt`, which compensates for the order in which elements are popped from the stack.

Below is the code for the integer square root, where we have also labeled each instruction with its address in the program array. This makes it easier to compute the offsets for the conditional branches and jumps. Addresses are given not in hex but decimal. They have to be converted to hex for the program code.

```
00  0x10 0x00  bipush 0
02  0x36 0x02  istore 2  // i = V[2] = 0
04  0x10 0x00  bipush 0
06  0x36 0x03  istore 3  // k = V[3] = 0
// <loop>
08  0x15 0x03  iload 3   // k = V[3]
10  0x15 0x01  iload 1   // n = V[1]
12  0xA3 0x18  if_icmpgt 24 // if k > n goto <finish>
14  0x15 0x03  iload 3   // k
16  0x15 0x02  iload 2   // k, i
18  0x10 0x02  bipush 2  // k, i, 2
20  0x68       imul      // k, i*2
21  0x10 0x01  bipush 1  // k, i*2, 1
23  0x60       iadd      // k, i*2+1
24  0x60       iadd      // k+i*2+1
25  0x36 0x03  istore 3  // .
27  0x15 0x02  iload 2   // i
29  0x10 0x01  bipush 1  // i, 1
31  0x60       iadd      // i+1
32  0x36 0x02  istore 2  // .
34  0xA7 0xE6  goto -26  // goto <loop>
// <finish>
36  0x15 0x02  iload 2   // i
38  0x10 0x01  bipush 1  // i, 1
40  0x64       isub      // i-1
41  0xB1       ireturn   // return i-1
```

You are invited to read the binary hex file resulting from this in isqrt.hx, which also has bytes `0x2A` (for 42 instructions) and `0x04` (for 4 local variables) at the beginning.

6 Byte Code Verification

So far, we have not written any invariants in the code. What *is* the data structure invariant for code? How do we establish it?

We can try to derive this from the program that interprets the bytecode. First, we would like to check that there is valid instruction at every address we can reach when the program is executed. This is slightly complicated by forward and backward conditional branches and jumps, but overall not too difficult to check. We also want to check that all local variables used are less that `max_local`, so that references $V[i]$ will always be in bounds. Further, we check that when a function returns, there is exactly one value on the stack. This more difficult to check, again due to conditional branches and jumps, because the stack grows and shrinks. As part of this we should also verify that at any given instruction there are enough items on the stack to execute the instruction, for example, at least two for `iadd`.

These and a few other checks are performed by JVM *byte code verification*. The most important one we omitted here is *type checking*. It is not relevant for the JVM00 because all data elements are integers. After byte code verification, a number of runtime checks can be avoided because we have verified statically that they can not occur. Realistic byte code verification is far from trivial, but we see here that it just establishes a data structure invariant for the JVM byte code interpreter.

It is important to recognize that there are limits to what can be done with bytecode verification before the code is executed. For example, we can not check in general if division might try to divide by 0, or if the program will terminate. There is a lot of research in the area of programming languages concerned with pushing the boundaries of static verification, including here at Carnegie Mellon University. Perhaps future instances of this course will benefit from this research by checking your C0 program invariants, at least to some extent, and pointing out bugs before you ever run your program just like the parser and type checker do.

7 JVM00 Instruction Reference

```
<c> = constant
<i> = local variable index
<o> = branch offset
<f> = function address

// operand stack and locals
0x00 nop           S   -> S
0x59 dup           S,v -> S,v,v
0x57 pop           S,v -> S
0x5F swap          S,v1,v2 -> S,v2,v1
0x10 bipush <c>    S -> S,c
0x15 iload <i>     S -> S,V[i]
0x36 istore <i>    S,v -> S; V[i] = v;
0x84 iinc <i>,<c>

// arithmetic
0x60 iadd          S,x,y -> S,x+y
0x64 isub          S,x,y -> S,x-y
0x68 imul          S,x,y -> S,x*y
0x6C idiv          S,x,y -> S,x/y
0x70 irem          S,x,y -> S,x%y
0x74 ineg          S,x -> S,-x
0x78 ishl          S,x,y -> S,x<<y
0x7a ishr          S,x,y -> S,x>>y
0x7C iushr         S,x,y -> S,x>>>y
0x7E iand          S,x,y -> S,x&y
0x80 ior           S,x,y -> S,x|y
0x82 ixor          S,x,y -> S,x^y

// control
0xB1 ireturn       .,x -> .
0xA7 goto <o>
0x9F if_icmpeq <o> S,x,y -> S
0xA0 if_icmpne <o> S,x,y -> S
0xA1 if_icmplt <o> S,x,y -> S
0xA2 if_icmpge <o> S,x,y -> S
0xA3 if_icmpgt <o> S,x,y -> S
0xA4 if_icmple <o> S,x,y -> S
```