Lecture Notes on
Programs as Data: The C0VM

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 21
June 19, 2014

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*. Another instance are first-class functions, which you will study in great depth in 15–150, a course 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 in turn 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.

In this lecture we explore how to apply the same principles to develop a virtual machine to implement C0. We call this the C0VM and in Assignment 9 of this course you will have the opportunity to implement it. The `cc0` compiler has an option (`-b`) to produce bytecode appropriate for the C0VM. This will give you insight not only into programs-as-data, but also into how C0 is executed, its *operational semantics*.

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. You have been introduced to all the unsafe features of C, when the course switched to C, and we taught 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.

Implementation of a virtual machine is actually one of the applications where even today C is usually the language of choice. That's because C gives you control over the memory layout of data, and also permits the kind of optimizations that are crucial to make a virtual machine efficient. Here, we don't care so much about efficiency, being mostly interested in correctness and clarity, but we still use C to implement the C0VM.

2 A Stack Machine

The C0VM is a *stack machine*. 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 *virtual machine instruction* on left, in textual form, and the operand stack *after* the instruction on the right has been executed. 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 translation of expressions to instructions is what a compiler would normally do. Here we just write the instructions by hand, in effect simulat-

ing 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 C0VM, instructions are represented as bytes. This means we only have at most 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 C0VM reference, listing only the instructions needed above.

```
0x10 bipush <b>     S      -> S,b
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
		.
`10 03`	`bipush 3`	3
`10 04`	`bipush 4`	3, 4
`60`	`iadd`	7
`10 05`	`bipush 5`	7, 5
`68`	`imul`	35
`10 02`	`bipush 2`	35, 2
`6C`	`idiv`	17

In the figure above, and in the rest of these notes, we always show bytecode in hexadecimal form, without the `0x` prefix. 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 C0VM 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 Compiling to Bytecode

The `cc0` compiler provides an option `-b` to generate bytecode. You can use this to experiment with different programs to see what they translate to.

For the simple arithmetic expression from the previous section we could create a file `ex1.c0`: We compile it with

```
% cc0 -b ex1.c0
```

which will write a file `ex1.bc0`. In the current version of the compiler, this has the following content: We will explain various parts of this file later on.

It consists of a sequence of bytes, each represented by two hexadecimal digits. In order to make the bytecode readable, it also includes comments. Each comment starts with `#` and extends to the end of the line. Comments are completely ignored by the virtual machine and are there only for you to read.

We focus on the section starting with `#<main>`. The first three lines

```
#<main>
00 00             # number of arguments = 0
00 00             # number of local variables = 0
00 0C             # code length = 12 bytes
```

tell the virtual machine that the function `main` takes no arguments, uses no local variables, and its code has a total length of 12 bytes (`0x0C` in hex). The next few lines embody exactly the code we wrote by hand. The comments first show the virtual machine instruction and then the expression in the source code that was translated to the corresponding byte code.

```
10 03    # bipush 3        # 3
10 04    # bipush 4        # 4
60       # iadd            # (3 + 4)
10 05    # bipush 5        # 5
68       # imul            # ((3 + 4) * 5)
10 02    # bipush 2        # 2
6C       # idiv            # (((3 + 4) * 5) / 2)
B0       # return          #
```

The `return` instruction at the end means that the function returns the value that is currently the only one on the stack. When this function is executed, this will be the value of the expression shown on the previous line, `(((3 + 4) * 5) / 2)`.

As we proceed through increasingly complex language constructs, you should experiment yourself, writing C0 programs, compiling them to byte code, and testing your understanding by checking that it is as expected (or at least correct).

4 Local Variables

So far, the only part of the *runtime system* that we needed was the local operand stack. 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 `vload` instruction, and we can pop the value from the top of the stack and store it in a local variable with the `vstore` instruction. Initially, when a function is called, its arguments $x_0, \ldots, x_{n-1}$ are stored as local variables $V[0], \ldots, V[n-1]$.

Assume we want to implement the function `mid`.

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

Here is a summary of the instructions we need

```
0x15 vload <i>     S -> S,v        (v = V[i])
0x36 vstore <i>    S,v -> S        (V[i] = v)
0x64 isub          S,x,y -> S,x-y
0xB0 return        .,v -> .
```

Notice that for `return`, 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 reading the program.

```
#<mid>
00 02             # number of arguments = 2
00 03             # number of local variables = 3
00 10             # code length = 16 bytes
15 00    # vload 0         # lower
15 01    # vload 1         # lower, upper
15 00    # vload 0         # lower, uppper, lower
64       # isub            # lower, (upper - lower)
10 02    # bipush 2        # lower, (upper - lower), 2
6C       # idiv            # lower, ((upper - lower) / 2)
60       # iadd            # (lower + ((upper - lower) / 2))
36 02    # vstore 2        # mid = (lower + ((upper - lower) / 2));
```

```
15 02    # vload 2          # mid
B0       # return           #
```

We can optimize this piece of code, simply removing the last `vstore 2` and `vload 2`, but we translated the original literally to clarify the relationship between the function and its translation.

5 Constants

So far, the `bipush <b>` instruction is the only way to introduce a constant into the computation. Here, b is a *signed byte*, so that its possible values are $-128 \leq b < 128$. What if the computation requires a larger constant?

The solution for the C0VM and similar machines is not to include the constant directly as arguments to instructions, but store them separately in the byte code file, giving each of them an index that can be referenced from instructions. Each segment of the byte code file is called a *pool*. For example, we have a pool of integer constants. The instruction to refer to an integer is `ildc` (integer load constant).

```
0x13 ildc <c1,c2>  S -> S, x:w32     (x = int_pool[(c1<<8)|c2])
```

The index into the constant pool is a 16-bit unsigned quantity, given in two bytes with the most significant byte first. This means we can have at most $2^{16} - 1 = 65,535$ different constants in a byte code file.

As an example, consider a function that is part of a linear congruential pseudorandom number generator. It generates the next pseudorandom number in a sequence from the previous number. There are three constants in this file that require more than one byte to represent: `1664252`, `1013904223`, and `0xdeadbeef`. Each of them is assigned an index in the integer pool. The constants are then pushed onto the stack with the `ildc` instruction. The comments denote the ith integer in the constant pool by $c[i]$.

There are other pools in this file. The *string pool* contains string constants. The *function pool* contains the information on each of the functions, as explained in the next section. The *native pool* contains references to "native" functions, that is, library functions not defined in this file.

6 Function Calls

As already explained, the function pool contains the information on each function which is the number of arguments, the number of local variables,

the code length, and then the byte code for the function itself. Each function is assigned a 16-bit unsigned index into this pool. The `main` function always has index 0. We call a function with the `invokestatic` instruction.

```
0xB8 invokestatic <c1,c2> S, v1, v2, ..., vn -> S, v
```

We find the function g at `function_pool[c1<<8|c2]`, which must take n arguments. After $g(v_1, \ldots, v_n)$ returns, its value will be on the stack instead of the arguments.

Execution of the function will start with the first instruction and terminate with a return (which does not need to be the last byte code in the function). So the description of functions themselves is not particularly tricky, but the implementation of function calls is.

Let's collect the kind of information we already know about the runtime system of the virtual machine. We have a number of *pools* which come from the byte code file. These pools are *constant* in that they never change when the program executes.

Then we have the *operand stack* which expands and shrinks within each function's operation, and the *local variable array* which holds function arguments and the local variables needed to execute the function body.

In order to correctly implement function calls and returns we need one further runtime structure, the *call stack*. The call stack is a stack of so-called *frames*. We now analyze what the role of the frames is and what they need to contain.

Consider the situation where a function f is executing and calls a function g with n arguments. At this point, we assume that f has pushed the arguments onto the operand stack. Now we need take the following steps:

1. Create a new local variable array V_g for the function g.
2. Pop the arguments from f's operand stack S_f and store them in g's local variable array $V_g[0..n)$.
3. Push a frame containing V_f, S_f, and the next program counter pc_f on the call stack.
4. Create a new (empty) operand stack S_g for g.
5. Start executing the code for g.

When the called function g returns, its return value is the only value on its operand stack S_g. We need to do the following

1. Pop the last frame from the call stack. This frame holds V_f, S_f, and pc_f (the return address).
2. Take the return value from S_g and push it onto S_f.
3. Restore the local variable array V_f.
4. Deallocate any structs no longer required.
5. Continue with the execution of f at pc_f.

Concretely, we suggest that a frame from the call stack contain the following information:

1. An array of local variables V.
2. The operand stack S.
3. A pointer to the function body.
4. The return address which specifies where to continue execution.

We recommend that you simulate the behavior of the machine on a simple function call sequence to make sure you understand the role of the call stack.

7 Conditionals

The C0VM does not have if-then-else or conditional expressions. Like machine code and other virtual machines, it has *conditional branches* that jump to another location in the code if a condition is satisfied and otherwise continue with the next instruction in sequence.

```
0x9F if_cmpeq <o1,o2>   S, v1, v2 -> S        (pc = pc+(o1<<8|o2) if v1 == v2)
0xA0 if_cmpne <o1,o2>   S, v1, v2 -> S        (pc = pc+(o1<<8|o2) if v1 != v2)
0xA1 if_icmplt <o1,o2>  S, x:w32, y:w32 -> S  (pc = pc+(o1<<8|o2) if x < y)
0xA2 if_icmpge <o1,o2>  S, x:w32, y:w32 -> S  (pc = pc+(o1<<8|o2) if x >= y)
0xA3 if_icmpgt <o1,o2>  S, x:w32, y:w32 -> S  (pc = pc+(o1<<8|o2) if x > y)
0xA4 if_icmple <o1,o2>  S, x:w32, y:w32 -> S  (pc = pc+(o1<<8|o2) if x <= y)
0xA7 goto <o1,o2>       S -> S                (pc = pc+(o1<<8|o2))
```

As part of the test, the arguments are popped from the operand stack. Each of the branching instructions takes two bytes are arguments which describe

a *signed* 16-bit offset. If that is positive we jump forward, if it is negative we jump backward in the program.

As an example, we compile the following loop, adding up odd numbers to obtain perfect squares. The compiler currently produces somewhat ideosyncratic code, so what we show below has been edited to make the correspondence to the source code more immediate.

```
#<main>
00 00             # number of arguments = 0
00 02             # number of local variables = 2
00 23             # code length = 35 bytes
10 00    # bipush 0        # 0
36 00    # vstore 0        # sum = 0;
10 01    # bipush 1        # 1
36 01    # vstore 1        # i = 1;
# <00:loop>
15 01    # vload 1         # i
10 64    # bipush 100      # 100
A2 00 14 # if_icmpge 20    # if (i >= 100) goto <01:endloop>
15 00    # vload 0         # sum
15 01    # vload 1         # i
60       # iadd            #
36 00    # vstore 0        # sum += i;
15 01    # vload 1         # i
10 02    # bipush 2        # 2
60       # iadd            #
36 01    # vstore 1        # i += 2;
A7 FF EB # goto -21        # goto <00:loop>
# <01:endloop>
15 00    # vload 0         # sum
B0       # return          #
```

The compiler has embedded symbolic labels in this code, like `<00:loop>` and `<01:endloop>` which are the targets of jumps or conditional branches. In the actual byte code, they are turned into relative offsets. For example, if we count forward 20 bytes, starting from `A2` (the byte code of `if_icmpge`, the negation of the test `i < 100` in the source) we land at `<01:endloop>` which labels the `vload 0` instruction just before the return. Similarly, if we count backwards 21 bytes from `A7` (which is a `goto`), we land at `<00:loop>` which starts with `vload 1`.

8 The Heap

In C0, structs and arrays can only be allocated on the system heap. The virtual machine must therefore also provide a heap in its runtime system. If you implement this in C, the simplest way to do this is to use the runtime heap of the C language to implement the heap of the C0VM byte code that you are interpreting. One can use a garbage collector for C such as `libgc` in order to manage this memory. We can also sidestep this difficulty by assuming that the C0 code we interpret does not run out of memory.

We have two instructions to allocate memory.

```
0xBB new <s>       S -> S, a:*          (*a is now allocated, size <s>)
0xBC newarray <s>  S, n:w32 -> S, a:*   (a[0..n) now allocated)
```

The `new` instructions takes a size s as an argument, which is the size (in bytes) of the memory to be allocated. The call returns the address of the allocated memory. It can also fail with an exception, in case there is insufficient memory available, but it will never return `NULL`. `newarray` also takes the number n of elements from the operand stack, so that the total size of allocated space is $n * s$ bytes.

For a pointer to a struct, we can compute the address of a field by using the `aaddf` instruction. It takes an unsigned byte offset f as an argument, pops the address a from the stack, adds the offset, and pushes the resulting address $a + f$ back onto the stack. If a is null, and error is signaled, because the address computation would be invalid.

```
0x62 aaddf <f>      S, a:* -> S, (a+f):*   (a != NULL; f field offset)
```

To access memory at an address we have computed we have the `mload` and `mstore` family of instructions. They vary, depending on the size of data that are loaded from or stored to memory.

```
0x2E imload    S, a:* -> S, x:w32   (x = *a, a != NULL, load 4 bytes)
0x2F amload    S, a:* -> S, b:*     (b = *a, a != NULL, load address)
0x4E imstore   S, a:*, x:w32 -> S   (*a = x, a != NULL, store 4 bytes)
0x4F amstore   S, a:*, b:* -> S     (*a = b, a != NULL, store address)
```

They all consume an address from the operand stack. `imload` reads a 4-byte value from the given memory address and pushes it on the operand stack. `imstore` pops a 4-byte value from the operand stack and stores it at the given address. The `amload` and `amstore` versions load and store and address, respectively. There are also `cmload` and `cmstore` explained in the next section for single-byte loads and stores.

As an example, consider the following struct declaration and function.

```
struct point {
  int x;
  int y;
};
typedef struct point* point;

point reflect(point p) {
  point q = alloc(struct point);
  q->x = p->y;
  q->y = p->x;
  return q;
}
```

The `reflect` function is compiled to the following code. When reading this code, recall that `q->x`, for example, stands for `(*q).x`. In the comments, the compiler writes the address of the x field in the struct pointed to by q as `&q->x`, in analogy with C's address-of operator `&`.

```
#<reflect>
00 01             # number of arguments = 1
00 02             # number of local variables = 2
00 1B             # code length = 27 bytes
BB 08    # new 8           # alloc(struct point)
36 01    # vstore 1        # q = alloc(struct point);
15 01    # vload 1         # q
62 00    # aaddf 0         # &q->x
15 00    # vload 0         # p
62 04    # aaddf 4         # &p->y
2E       # imload          # p->y
4E       # imstore         # q->x = p->y;
15 01    # vload 1         # q
62 04    # aaddf 4         # &q->y
15 00    # vload 0         # p
62 00    # aaddf 0         # &p->x
2E       # imload          # p->x
4E       # imstore         # q->y = p->x;
15 01    # vload 1         # q
B0       # return          #
```

We see that in this example, the size of a `struct point` is 8 bytes, 4 each for the x and y fields. You should scrutinize this code carefully to make sure you understands how structs work.

Array accesses are similar, except that the address computation takes an index i from the stack. The size of the array elements is stored in the runtime structure, so it is not passed as an explicit argument. Instead, the byte code interpreter must retrieve the size from memory. The following is our sample program.

```
int main() {
  int[] A = alloc_array(int, 100);
  for (int i = 0; i < 100; i++)
    A[i] = i;
  return A[99];
}
```

Showing only the loop, we have the code below (again slightly edited). Notice the use of `aadds` to consume A and i from the stack, pushing `&A[i]` onto the stack.

```
# <00:loop>
15 01    # vload 1         # i
10 64    # bipush 100      # 100
9F 00 15 # if_cmpge 21     # if (i >= 100) goto <01:endloop>
15 00    # vload 0         # A
15 01    # vload 1         # i
63       # aadds           # &A[i]
15 01    # vload 1         # i
4E       # imstore         # A[i] = i;
15 01    # vload 1         # i
10 01    # bipush 1        # 1
60       # iadd            #
36 01    # vstore 1        # i += 1;
A7 FF EA # goto -22        # goto <00:loop>
# <01:endloop>
```

There is a further subtlety regarding booleans and characters stored in memory, as explained in the next section.

9 Characters and Strings

Characters in C0 are ASCII characters in the range from $0 \leq c < 128$. Strings are sequences of non-NULL characters. While C0 does not prescribe the representation, we follow the convention of C to represent them as an array of characters, terminated by `'\0'` (NUL). Arrays (and therefore strings) are manipulated via their addresses, and therefore add to the types we denote by `a:*`.

But what about constant strings appearing in the program? For them, we introduce the *string pool* as another section of the byte code file. This pool consists of a sequence of strings, each of them terminated by `'\0'`, represented as the byte `0x00`. Consider the program There are two string constants, `"Hello "` and `"World!\n"`. In the byte code file below they are stored in the string pool at index positions 0 and 7.

```
C0 C0 FF EE       # magic number
00 05             # version 2, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 0F             # string pool total size
# string pool
48 65 6C 6C 6F 20 00  # "Hello "
57 6F 72 6C 64 21 0A 00  # "World!\n"
```

In the byte code program, we access these strings by pushing their *address* onto the stack using the `aldc` instruction.

```
0x14 aldc <c1,c2>  S -> S, a:*   (a = &string_pool[(c1<<8)|c2])
```

We can see its use in the byte code for the main function.

```
#<main>
00 00             # number of arguments = 0
00 02             # number of local variables = 2
00 1B             # code length = 27 bytes
14 00 00 # aldc 0          # s[0] = "Hello "
36 00    # vstore 0        # h = "Hello ";
15 00    # vload 0         # h
14 00 07 # aldc 7          # s[7] = "World!\n"
B7 00 00 # invokenative 0  # string_join(h, "World!\n")
```

```
36 01    # vstore 1         # hw = string_join(h, "World!\n");
15 01    # vload 1          # hw
B7 00 01 # invokenative 1   # print(hw)
57       # pop              # (ignore result)
15 01    # vload 1          # hw
B7 00 02 # invokenative 2   # string_length(hw)
B0       # return           #
```

Another noteworthy aspect of the code is the use of native functions with index 0, 1, and 2. For each of these, the *native pool* contains the number of arguments and an internal index.

```
00 03             # native count
# native pool
00 02 00 4E       # string_join
00 01 00 06       # print
00 01 00 4F       # string_length
```

There is a further subtle point regarding the memory load and store instructions and their interaction with strings. As we can see from the string pool representation, a character takes only one byte of memory. The operand stack and local variable array maintains all primitive types as 4-byte quantities. We need to mediate this difference when loading or storing characters. Booleans similarly take only one byte, where 0 stands for `false` and 1 for `true`. For this purpose, the C0VM has variants of the `mload` and `mstore` instructions that load and store only a single byte.

```
0x34 cmload   S, a:* -> S, x:w32   (x = (w32)(*a), a != NULL, load 1 byte)
0x55 cmstore  S, a:*, x:w32 -> S   (*a = x & 0x7f, a != NULL, store 1 byte)
```

As part of the load operation we have to convert the byte to a four-byte quantity to be pushed onto the stack; when writing we have to mask out the upper bits. Because characters c in C0 are in the range $0 \leq c < 128$ and booleans are represented by just 0 (for `false`) and 1 (for `true`), we exploit and enforce that all bytes represent 7-bit unsigned quantities.

10 Byte Code Verification

So far, we have not discussed any invariants to be satisfied by the information stored in the byte code file. What *are* the invariants for code, encoded as data? How do we establish them?

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 `num_vars`, 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 is 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 *byte code verification* of the Java Virtual Machine (JVM). The most important one we omitted here is *type checking*. It is not relevant for the C0VM because we simplified the file format by eliminating type information. 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 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.

11 Implementing the C0VM

For some information, tips, and hints for implementing the C0VM in C we refer the reader to the Assignment 9 writeup and starter code.

12 C0VM Instruction Reference

13 C0VM File Format Reference