Virtual Machines and Compilation

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 29, 2021

Table of Contents

Note: The material covered in this lecture will not be on the midterm.

Table of Contents

The Difference Between Interpreters and Compilers

An interpreter runs the program, while a compiler translates expressions to another language, usually assembly language or machine code.

What Do We Compile To?

- There are some languages which are normally compiled to the assembly language or machine code of real hardware architectures (e.g., the GCC and Clang compilers for C compile to architectures such as x86-64 and ARM).

What Do We Compile To?

- There are some languages which are normally compiled to the assembly language or machine code of real hardware architectures (e.g., the GCC and Clang compilers for C compile to architectures such as x86-64 and ARM).
- However, some compilers translate expressions to the assembly language or machine code of contrived architectures known as virtual machines.

What Do We Compile To?

- There are some languages which are normally compiled to the assembly language or machine code of real hardware architectures (e.g., the GCC and Clang compilers for C compile to architectures such as x86-64 and ARM).
- However, some compilers translate expressions to the assembly language or machine code of contrived architectures known as virtual machines.
- In fact, a lot of languages that are commonly thought of as "interpreted" (such as Python) actually use a VM.
  - We've been using a virtual machine (the Racket virtual machine) the whole time we've been using Scheme/Racket!

Table of Contents

A Note about Virtual Machines

- In this class, whenever we are discussing virtual machines, we are not discussing *system virtual machines* such as VMWare Workstation, Oracle VirtualBox, Parallels, etc.
    - They aim to emulate an entire computer system (not just the processor, but memory, storage, peripherals, etc.) for the purpose of running another operating system on top of the OS running the virtual machine.

A Note about Virtual Machines

- Instead, we will be discussing *process virtual machines*, which are used for providing a runtime environment for a single application or process. This runtime environment has its own contrived instruction set and memory allocator, but it does not strive to emulate a complete computer system.
  - In Unix parlance, each process gets its own virtual machine (for example, three separately-running Java programs run in their own JVM instances).

Examples of Virtual Machine Architectures

- Java Virtual Machine (JVM)
- Microsoft .NET Common Language Runtime (CLR)
- Python's virtual machine
- Ruby's virtual machine (Yet another Ruby VM – YARV)
- V8 JavaScript engine (used by Google Chrome)

Note that there are many others.

Why Virtual Machines?

- Increases *portability* (the ability for programs to run on a variety of operating systems and architectures).
- We can define a programming language's semantics by a virtual machine instead of piggybacking off another machine or language.
- Compiled code runs faster than interpreted code.
- We can exploit *just-in-time compilation* to further increase speed (more on this later in the course).

Bytecode

Bytecode is essentially a virtual machine's "machine code". Bytecode is a sequence of instructions. The virtual machine interprets instructions written in bytecode and executes them, translating it to the host environment's machine code.

There are many ways to design a virtual machine's architecture, but we will cover two approaches: *register machines* (covered in SICP Chapter 5) and *stack machines*, a very popular VM design choice.

Register Machines

Register machines consist of the following components:

- An unlimited amount of registers, which are each of a fixed size and are used to store the inputs and outputs of instructions.
    - Operators such as addition and equality operate on registers and store their results in registers.
- A stack, which is used for handling function calls
- A program counter
- Access to memory

Instructions Necessary in a Register Machine

- Assigning a value to a specific register
- Testing whether a register conforms to a specified predicate
- Conditional and unconditional branching
- Pushing and popping items to and from the stack
- Loading a value from a memory location to a specific register
- Storing a value from a register to a specific memory location

Example: GCD in Assembly for a Register Machine

This is GCD written in Scheme. Example is from Chapter 5 of *Structure and Interpretation of Computer Programs*:

```
; Note that gcd is tail-recursive
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Example: GCD in Assembly for a Register Machine

Example is from Chapter 5 of *Structure and Interpretation of Computer Programs*:

```
; Note that while this is in S-expression syntax,
; this is not Scheme code.
(controller
  test-b
  (test (op =) (reg b) (const 0)) ; rB = 0?
  (branch (label gcd-done))  ; if rB == 0, goto gcd-done
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label test-b))
  gcd-done)
```

Example: GCD in Assembly for a Register Machine

Here is a more natural assembly syntax:

```
test-b:
  EQ RB, 0
  BRANCH gcd-done
  REM RT, RA, RB
  ASSIGN RA, RB
  ASSIGN RB, RT
  GOTO test-b
gcd-done:
```

Stack Machines

Unlike register machines, where operators operate on registers, in a stack machine, operators operate on entries that have been placed onto a stack, similar to the postfix calculator you wrote in Lab 1.

```
; add two numbers
PUSH 5
PUSH 7
ADD ; pops 7 and 5, does 7+5, and pushes 12
```

Table of Contents

This lecture will cover the basics of compilation; there are entire courses devoted to compilation, such as CS 153.

Compiler Components

A compiler can be split into two components: the *front end*, which handles lexing and parsing to create an abstract syntax tree, and the *back end*, which transforms the abstract syntax tree into the target language, which is usually assembly, bytecode, or machine code.

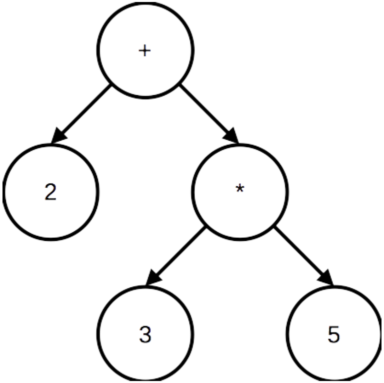
A compiler's backend will be written very similar to an interpreter's `eval` function, where it takes an expression (the AST) and the environment as input. However, since the goal of the compiler is not to execute the AST, but to `translate` it, we need to recreate the environment.

Let's begin with a simple example: the infix arithmetic language. Let's make a compiler for infix arithmetic expressions.

```
<multop>       ::= '*' | '/'
<addop>        ::= '+' | '-'
<number-expr> ::= ('-')?[0-9]+ | ('-')?[0-9]+ '.' [0-9]+
                  | <add-expr>
<add-expr>     ::= <number-expr> <addop> <number-expr>
                  | <mult-expr>
<mult-expr>    ::= <number-expr> <multop> <number-expr>
                  | <expo-expr>
<expo-expr>    ::= <number-expr> ^ <number-expr>
                  | <parens-expr>
<parens-expr> ::= '(' <number-expr> ')'
```

Assuming that we are using a stack VM, how would we convert expressions such as $2 + 3 * 5$ into bytecode?

The front-end of the compiler would covert the expression $2 + 3 * 5$ into the following abstract syntax tree:



How do we convert $2 + 3 * 5$ into bytecode for a stack machine?

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is +. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is +. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is `+`. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.
3. Visit the right node, which is `*`. We know it must have two children. We are going to push the results of the children onto the stack.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is `+`. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.
3. Visit the right node, which is `*`. We know it must have two children. We are going to push the results of the children onto the stack.
4. Visit the left node, which is 3. Output `PUSH 3`.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is `+`. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.
3. Visit the right node, which is `*`. We know it must have two children. We are going to push the results of the children onto the stack.
4. Visit the left node, which is 3. Output `PUSH 3`.
5. Visit the right node, which is 5. Output `PUSH 5`.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is +. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.
3. Visit the right node, which is *. We know it must have two children. We are going to push the results of the children onto the stack.
4. Visit the left node, which is 3. Output `PUSH 3`.
5. Visit the right node, which is 5. Output `PUSH 5`.
6. Perform multiplication. Output `MULT`.

Compiling $2 + 3 * 5$ to Bytecode for a Stack Machine

1. Visit top node, which is +. We know from the definition of addition that it has two children. We are going to push the results of each child onto the stack.
2. Visit the left node, which is 2. Output `PUSH 2`.
3. Visit the right node, which is ∗. We know it must have two children. We are going to push the results of the children onto the stack.
4. Visit the left node, which is 3. Output `PUSH 3`.
5. Visit the right node, which is 5. Output `PUSH 5`.
6. Perform multiplication. Output `MULT`.
7. Perform addition. Output `ADD`.

$2 + 3 * 5$ in Bytecode for a Stack Machine

```
PUSH 2
PUSH 3
PUSH 5
MULT
ADD
```

What if we add variables to our calculator?

```
x = 5
y = 10
z = x + y
a = x^2 + y^2
```

What if we add variables to our calculator?

```
x = 5
y = 10
z = x + y
a = x^2 + y^2
```

- The compiled code would maintain its own environment for managing its values for x, y, z, and a. The compiler would maintain a mapping between variables and memory locations, and the compiled code would refer to these memory locations, placing values onto the stack whenever necessary.

Table of Contents

Stack Frames

Internally, every time a program calls a function, the runtime environment creates a stack frame that contains that local function's variables.

The Stack versus the Heap

- The stack contains local variables, whereas the heap contains global variables and dynamically-allocated objects.

Perform whiteboard example in Java

Memory Management

Most modern programming languages provide automated memory management, where programmers are not responsible for deallocating unused objects that are allocated on the heap.

- This "modern" development goes all the way back to the original version of Lisp

Garbage collection makes automated memory management possible.