16

Compilation

One way to improve the efficiency of interpretation is to convert the abstract representation into another representation that can be easier and faster to execute. This process, when this other representation is low-level enough, is called *compilation*. A compiler is a program that takes surface syntax and produces a low-level representation that can be executed efficiently. While the abstract representation is highly structured — it is a tree structure, hence recursive — a low-level representation is usual flat, like an array of instructions.

The low-level representation can often be understood as the instruction set of a dedicated *virtual machine*. An example of this is the Java Virtual Machine (JVM), a virtual machine optimized designed to execute a low-level representation that the Java compile produces (basically, the content of `.class` files).

Here's an example of a simple virtual machine that can be used to convert a simple arithmetic surface syntax (which could be taken to be in an abstract representation such as we've been using). It consists of a stack that holds integers, and the instructions of the virtual machine include:

- `n`: which is an instruction to push integer n on the stack;
- `+`: which is an instruction to pop two integers off the stack, add them, and push the result on the stack;
- `*`: which is an instruction to pop two integers off the stack, multiply then, and push the result on the stack.

Given this, here is a simple transformation function that takes an arithmetic expression and transforms it into a sequence of instructions for the stack

machine:

$$\begin{aligned}
[\![n]\!] &= \mathsf{n} \\
[\![\texttt{(+ } e_1 \ e_2\texttt{)}]\!] &= [\![e_1]\!]\ [\![e_2]\!]\ \oplus \\
[\![\texttt{(* } e_1 \ e_2\texttt{)}]\!] &= [\![e_1]\!]\ [\![e_2]\!]\ \otimes
\end{aligned}$$

Transforming `(+ 1 (* 3 4))`, for example, yields:

$$\begin{aligned}
[\![\texttt{(+ 1 (* 3 4))}]\!] &= [\![1]\!]\ [\![(*34)]\!]\ \oplus \\
&= 1\ [\![(*34)]\!]\ \oplus \\
&= 1\ [\![3]\!]\ [\![4]\!]\ \otimes\ \oplus \\
&= 1\ 3\ 4\ \otimes\ \oplus
\end{aligned}$$

The sequence of instructions $1\ 3\ 4\ \otimes\ \oplus$ pushes 1, 3, and 4 on the stack, then pops 3 and 4 and multiples them together to put 12 on the stack, and then pops 12 and 1 to put 13 on the stack. The result of the evaluation is sitting on top of the stack: 13.

We introduce a virtual machine that we will use as a target for compilation of TFUNC, our typed functional language. It uses an array of instructions, with some occasional values (representing integers, Booleans, and closures) and primitive operations in the mix. I'm going to call the array `code`. Indices of entries in the code array are called *addresses*.

The machine has two special registers (variables):

- PC: holds the address of the instruction to execute next;
- ENV: holds an environment

For the time being, you can think of the environment as the same kind of structure we've been using in the abstract representation. We will see soon that we can simply think of it as a linked list of values.

The machine has three stacks: a value stack (for storing values), an address stack (for storing addresses), and an environment stack (for storing environments). Instructions manipulate the registers and the stacks.

The machine uses two kinds of values: integers, and closures. A closure is represented as a pair $\langle a, e\rangle$ where a is an address, and e is an environment. A Boolean will be represented by value 1 for true and value 0 for false. We will not handle vectors for now.

Here are the instructions:

- STOP: stops execution and returns the value on top of the stack
- PUSH i: push integer i on the value stack
- PUSH-ADDR a: push address a on the address stack
- JUMP: pop an address off the address stack and set it as the new PC.
- JUMP-TRUE: pop a value off the value stack and an address off the address stack, and set the address as the new PC if the value is different than 0
- PUSH-CLOSURE: pop an address off the address stack, create a closure from it and the ENV register, and push that closure on the value stack
- POP-CLOSURE: pop a closure $\langle a, e \rangle$ off the value stack, and push a to the address stack and set ENV to e
- PUSH-ENV: push the environment in ENV on the environment stack
- POP-ENV: pop an environment off the environment stack and put it in ENV
- POP-TO-ENV s: pop a value off the value stack and add it to the environment in ENV associated to identifier s
- LOOKUP s: look up the value of identifier s in ENV and push it on the value stack
- PRIMCALL n p: pop n values off the value stack, call primitive operation p (from a bank of available operations) and push the result on the value stack
- NOP: do nothing

After every instruction, PC is incremented to point to the next memory location, unless PC is set by a jump.

It looks complicated, but in the end, it is a fairly simple machine that is well suited to our abstract representation. Here is a simple sequence of instructions to compute `(+ 1 (* 3 4))`, where each line describes the address of the instruction.

```
0000  PUSH 1
0001  PUSH 3
0002  PUSH 4
0003  PRIMCALL 2 oper_times
0004  PRIMCALL 2 oper_plus
0005  STOP
```

Simple enough.

Here is a more complicated example that computes the sum of all the numbers from 0 to 200000 using essentially the following process:

```
(let ((sum_iter (fun s (n result)
                  (if (= n 0) result (s (+ n -1) (+ n result))))))
  (sum_iter 200000 0))
```

The recursion is implemented via a loop — I don't bother creating a closure, because I know exactly where I'm supposed to jump (address 2) and which environment should exist (the original environment, which is saved right at the beginning). The "recursive call" is just jumping to address 2 after having restore the environment to the original one.

```
0000  PUSH 0
0001  PUSH 200000
# start of loop
0002  PUSH-ENV                # save environment
0002  POP-TO-ENV n
0003  POP-TO-ENV result
0004  LOOKUP n
0005  PUSH 0
0006  PRIMCALL 2 oper_equal   # is (= n 0)
0007  PUSH-ADDR 0018
0008  JUMP-TRUE
0009  LOOKUP n                # nope
0010  LOOKUP result
0011  PRIMCALL 2 oper_plus    # (+ n result)
0012  LOOKUP n
0013  PUSH -1
0014  PRIMCALL 2 oper_plus    # (+ n -1)
0015  POP-ENV                 # restore environment
```

```
0016  PUSH-ADDR 0002
0017  JUMP                    # loop
# we're done
0018  LOOKUP result
0019  STOP
```

This is tight code written by hand. Question is, can we create such code directly from the abstract representation? We can, but it won't look as tight as the above.

Here's a fairly simple compilation function, that converts abstract representation into a sequence of virtual machine instructions. The invariant we maintain is that every expression translates into a sequence of virtual machines instructions that when executed leaves everything as it was before execution, with the addition of the result of the evaluation sitting on top of the value stack.

The compilation is given by a function $\mathcal{C}[\![-]\!]$ that returns virtual machine instructions.

$$\mathcal{C}[\![\texttt{EInteger}(i)]\!] = \mathsf{PUSH}\ i$$

$$\mathcal{C}[\![\texttt{EBoolean}(b)]\!] = \begin{cases} \mathsf{PUSH}\ 1 & \text{if } b \text{ is true} \\ \mathsf{PUSH}\ 0 & \text{if } b \text{ is false} \end{cases}$$

$$\mathcal{C}[\![\texttt{EId}(s)]\!] = \mathsf{LOOKUP}\ s$$

$$
\begin{aligned}
\mathcal{C}[\![\texttt{EIf}(\mathit{cond}, \mathit{then}, \mathit{else})]\!] = & \mathcal{C}[\![\mathit{cond}]\!] \\
& \textsf{PUSH-ADDR } @\mathit{thenpart} \\
& \textsf{JUMP-TRUE} \\
& \mathcal{C}[\![\mathit{else}]\!] \\
& \textsf{PUSH-ADDR } @\mathit{donepart} \\
& \textsf{JUMP} \\
& \mathit{thenpart}: \\
& \mathcal{C}[\![\mathit{then}]\!] \\
& \mathit{donepart}: \\
& \textsf{NOP}
\end{aligned}
$$

Here, I'm using $@XYZ$ as "the address of label XYZ", instead of hard-wiring addresses into the translation. When compiling, we need to compute the address corresponding to those labels, based on the location in the code array where the instructions are generated. Most of the time, it's pretty straightforward to compute. (It's also possible to generate code which uses those labels, and do a pass on the code after it's been generated to convert those labels into actual addresses. Of course, if we do it this way, then we need to make sure that every expression we translate generates fresh labels that don't clash with labels in other compiled expressions.)

$$
\begin{aligned}
\mathcal{C}[\![\texttt{EApply}(f, e_1, \ldots, e_k)]\!] = & \textsf{PUSH-ENV} \\
& \mathcal{C}[\![e_n]\!] \\
& \ldots \\
& \mathcal{C}[\![e_1]\!] \\
& \mathcal{C}[\![f]\!] \\
& \textsf{PUSH-ADDR } @\mathit{return} \\
& \textsf{POP-CLOSURE} \\
& \textsf{JUMP} \\
& \mathit{return}: \\
& \textsf{POP-ENV}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}[\![\texttt{EFunction}([p_1, \ldots, p_n], body)]\!] = & \text{PUSH-ADDR } @after \\
& \text{JUMP} \\
& fun: \\
& \text{POP-TO-ENV } p_1 \\
& \ldots \\
& \text{POP-TO-ENV } p_n \\
& \mathcal{C}[\![body]\!] \\
& \text{JUMP} \\
& after: \\
& \texttt{PUSH-ADDR } @fun \\
& \texttt{PUSH-CLOSURE}
\end{aligned}
$$

For recursive functions, things are similar, except that the function itself is put in the environment before the body of the function is executed, to reflect the way evaluation works in the interpreter:

$$
\begin{aligned}
\mathcal{C}[\![\texttt{ERecFunction}(self, [p_1, \ldots, p_n], body)]\!] = & \text{PUSH-ADDR } @after \\
& \text{JUMP} \\
& fun: \\
& \text{PUSH-ADDR } @fun \\
& \text{PUSH-CLOSURE} \\
& \text{POP-TO-ENV } self \\
& \text{POP-TO-ENV } p_1 \\
& \ldots \\
& \text{POP-TO-ENV } p_n \\
& \mathcal{C}[\![body]\!] \\
& \text{JUMP} \\
& after: \\
& \text{PUSH-ADDR } @fun \\
& \text{PUSH-CLOSURE}
\end{aligned}
$$

Let's see what this produces when we apply it to our `(+ 1 (+ 2 3))` example:

```
8 : Op_PUSH_ENV()
9 : Op_PUSH_ENV()
10 : Op_PUSH(3)
11 : Op_PUSH(2)
12 : Op_LOOKUP(+)
13 : Op_PUSH_ADDR(16)
14 : Op_POP_CLOSURE()
15 : Op_JUMP()
16 : Op_POP_ENV()
17 : Op_PUSH(1)
18 : Op_LOOKUP(+)
19 : Op_PUSH_ADDR(22)
20 : Op_POP_CLOSURE()
21 : Op_JUMP()
22 : Op_POP_ENV()
23 : Op_STOP()
```

It works. It's not great, but it works.

If we compare this code to the hand-written code from earlier, we note that the hand-written code calls primitive operations directly. Here, we call the closure + that's in the initial environment, which holds the address of some pre-compiled code that does the actual call to the primitive `oper_plus` operation. That precompiled code is pretty straightforward.

So one way we could improve things is to recognize when we're making a call to a primitive operation. If we could do this, then the code would look like:

```
8 : Op_PUSH_ENV()
9 : Op_PUSH_ENV()
10 : Op_PUSH(3)
11 : Op_PUSH(2)
12 : Op_PRIMCALL(2,+)
16 : Op_POP_ENV()
17 : Op_PUSH(1)
18 : Op_PRIMCALL(2,+)
22 : Op_POP_ENV()
23 : Op_STOP()
```

Which is already much better. And we would not even need the PUSH-ENV and POP-ENV since we are not invoking closures. But in order to do this, we

need to *analyze* the abstract represetnation to determine whether an identifier such as `+` refers to the built-in `+` provided by the initial environment, as opposed to an identifier bound by the user to some other value, for instance, like in:

```
(let ((+ (fun (a b) a)))
   (+ 10 20))
```

We will cover this kind of analysis later. Similarly, the hand-written code doesn't use a closure to represent `sum_iter` because all the call site of this function are known—the function is used only in the body of the `let`, and does not escape. An analysis can be performed to determine this situation.

In general, much of the serious work of compilation is at the level of these kind of *optimizations* that we can use to improve the quality of the generated code by analyzing the abstract representation and taking advantage of what we can infer from the analysis.

For the sake of completeness, consider our more complex example:

```
(let ((sum_iter (fun s (n result)
                    (if (= n 0) result (s (+ n -1) (+ result n))))))
   (sum_iter 200000 0))
```

Here is the result of our compilation function. Compare it to the hand-written code.

```
10 : Op_PUSH_ADDR(14)
11 : Op_PUSH_CLOSURE()
12 : Op_PUSH_ADDR(55)
13 : Op_JUMP()
14 : Op_PUSH_ADDR(14)
15 : Op_PUSH_CLOSURE()
16 : Op_POP_TO_ENV(s)
17 : Op_POP_TO_ENV(n)
18 : Op_POP_TO_ENV(result)
19 : Op_PUSH_ENV()
20 : Op_PUSH(0)
21 : Op_LOOKUP(n)
22 : Op_LOOKUP(=)
23 : Op_PUSH_ADDR(26)
24 : Op_POP_CLOSURE()
```

```
25 : Op_JUMP()
26 : Op_POP_ENV()
27 : Op_PUSH_ADDR(53)
28 : Op_JUMP_TRUE()
29 : Op_PUSH_ENV()
30 : Op_PUSH_ENV()
31 : Op_LOOKUP(n)
32 : Op_LOOKUP(result)
33 : Op_LOOKUP(+)
34 : Op_PUSH_ADDR(37)
35 : Op_POP_CLOSURE()
36 : Op_JUMP()
37 : Op_POP_ENV()
38 : Op_PUSH_ENV()
39 : Op_PUSH(-1)
40 : Op_LOOKUP(n)
41 : Op_LOOKUP(+)
42 : Op_PUSH_ADDR(45)
43 : Op_POP_CLOSURE()
44 : Op_JUMP()
45 : Op_POP_ENV()
46 : Op_LOOKUP(s)
47 : Op_PUSH_ADDR(50)
48 : Op_POP_CLOSURE()
49 : Op_JUMP()
50 : Op_POP_ENV()
51 : Op_PUSH_ADDR(54)
52 : Op_JUMP()
53 : Op_LOOKUP(result)
54 : Op_JUMP()
55 : Op_POP_TO_ENV(sum_iter)
56 : Op_PUSH_ENV()
57 : Op_PUSH(0)
58 : Op_PUSH(200000)
59 : Op_LOOKUP(sum_iter)
60 : Op_PUSH_ADDR(63)
61 : Op_POP_CLOSURE()
62 : Op_JUMP()
```

```
63 : Op_POP_ENV()
64 : Op_STOP()
```