Notes on Compilation

Programming Languages

November 1 & 8, 2016

Optimizing Interpretation

We've now essentially seen everything that pertains to interpretation. Sure, there are features that may be difficult to implement, but the basic infrastructure of interpretation the way we have it — parse surface syntax into abstract representation, possibly performing front-end transformations along the way, and then recursively evaluate the abstract representation to obtain a result — can handle most scenarios.

What we'll focus from now is not so much features of the language to be interpreted, but rather addressing the efficiency aspects of interpretation. To put it bluntly, our interpreters work, but they are abysmally slow. There are two ways to improve that:

1. Optimizing the abstract representation;

2. Translating the abstract representation into something that is easier and faster to execute — this leads to compilation.

Let's look at optimizing the abstract representation. There are some low-lying fruit there.

For instance, last lecture, we saw the CPS transformation that takes a program and transforms it into one where every function call, instead of returning a value, calls a continuation with the resulting value. A continuation is a function that captures the "rest of the computation".

I gave you a mathematical definition of the transformation, and my sample code added a method `cps()` to every Expression node in the abstract representation that performed the given transformation. (While I defined the CPS transformation on the surface syntax, it should be clear that it could be performed on the abstract representation.)

One issue is that the transformation that I defined is supremely inefficient. Consider the expression `(+ 1 (* 3 4))`. If we apply the CPS transformation I defined to this expression, in the context of a continuation `K`, the result is basically the following (where I use `let` to clean up the presentation slightly):

```
(let ((f *))
  (let ((a 3))
    (let ((b 4))
      (f a b (function (x)
               (let ((f +))
                 (let ((a 1))
                   (let ((b x))
                     (f a b K)))))))))
```

That's silly. And expensive to interpret: every `let` requires a binding to be added to the environment, every identifier requires a lookup in said environment. If we basically perform a substitution for all the bindings that define a constant value or another identifier, we obtain the much more reasonable expression:

```
(* 3 4 (function (x) (+ 1 x K)))
```

which is the example I gave last time. So an easy way to improve our interpreter is to make sure our CPS translation is less "naive", or to perform some simple substitutions on the result of the transformation. The sample code for this lecture uses the former approach, using a slightly more clever CPS transformation:

```
class ECall (Exp):

    ...

    def cps (self,k):
        vars_e = [ " _x{}_".format(fresh()) for e in self._args ]
        var_f = " _x{}_".format(fresh())
        args = []
        cont_args = []
        cont_exps = []
        if self._fun.is_basic:
            fun = self._fun
        else:
            fun = EId(var_f)
            cont_args.append(var_f)
            cont_exps.append(self._fun)
        for (var,e) in zip(vars_e,self._args):
            if e.is_basic:
                args.append(e)
            else:
                args.append(EId(var))
                cont_args.append(var)
```

```
                cont_exps.append(e)
        result = ECall(fun,args + [k])
        for (var,exp) in reversed(zip(cont_args,cont_exps)):
            result = exp.cps(EFunction([var],result))
        return result
```

Basically, every Expression node now has a field `is_basic` capturing whether the expression node represents a basic expression (a value or an identifier). Such basic expressions are used directly in the resulting CPS transformed code instead of being passed through a continuation that ultimately puts them in their final spot.

And indeed:

```
# defining exp to be (+ 1 (* 3 4))
>>> exp = ECall(EId("+"),[EValue(VInteger(1)),
                          ECall(EId("*"),[EValue(VInteger(3)),
                                          EValue(VInteger(4))])])
# the CPS transformation produces the code above
>>> print exp.cps(EId("K"))
ECall(EId(*),[EValue(3),
              EValue(4),
              EFunction([ _x41_],ECall(EId(+),[EValue(1),
                                               EId( _x41_),
                                               EId(K)]))])
```

Here are some timing results for an interpreter with the above improved CPS transformation, to give us a baseline.

```
Lecture 8 - REF Language (defun, define, CPS)
#quit to quit, #abs to see abstract representation
ref/cps> (let ((sum (function s (x) (if (zero? x) 0 (+ x (s (- x 1))))))
           (sum 200000))
Eval time: 9.305s
20000100000
```

I'm using the recursive `sum` function called with 200000 to provide a large enough running time so that we can spot differences and improvements.

Another potential target for improvement is identifier lookup, at least if we use an array-based environment of the kind I've been using in my sample code.[1]. Right now, in our

[1]If we use a dictionary-based environment, then the Python implementation of dictionaries is doing the heavy lifting of identifier lookup, and we don't need what I'm talking about in this section. This is another case of a feature of the object language (identifier lookup) modeled by a feature of the metalanguage. If we did not want to rely on that feature of the metalanguage, then we can use the technique described here.

interpreter, whenever we evaluate an identifer, we need to look it up in the environment. In the course of an evaluation, we lookup (search for) identifiers in the environment frequently. For instance, in the course of evaluating `(sum 200000)` above, we perform 3,800,011 identifier evaluations — that is 3,800,011 times we search for an identifier in the environment. That's a bit silly, because it turns out that we can predict exactly where in the environment any identifier will be. That's the whole point of the static binding discipline we've been using: we can tell statically, by looking at the code, which binding is going to hold the value of the identifier.

We can perform an analysis, *without executing the code*, that will tell us the shape that the environment will take when we evaluate every expression in our code. We can use this information to transform every `EId(n)` the abstract representation into an `EIdIndex(i)`, which takes an integer, and simply pulls out the ith element of the environment, without having to search through the environment. Here is the code for `EIdIndex`:

```
class EIdIndex (Exp):
    # identifier without lookup

    def __init__ (self,index):
        self._index = index
        self.expForm = "EIdIndex"
        self.is_basic = True

    def eval (self,env):
        return eval_iter(self,env)
```

and the corresponding code in `eval_iter()` that defines evaluation for `EIdIndex` nodes:

```
def eval_iter (exp,env):
    current_exp = exp
    current_env = env

    ...

        elif current_exp.expForm == "EId":

            for (id,v) in reversed(current_env):
                if current_exp._id == id:
                    return v
            raise Exception("Error: unknown identifier {}".format(
   current_exp._id))

        elif current_exp.expForm == "EIdIndex":

            return current_env[current_exp._index][1]
```

```
    ...
```

You can see the contrast with `EId` – evaluation for `EIdIndex` should be faster.

Here's the idea. We're going to define a new Expression node method, called *compile_env()*, that transforms the current node into a new node in which every `EId` has been replaced by a corresponding `EIdIndex`. We need to supply enough information for `compile_env()` to be able to determine the index of every identifier in the environment. We're going to do so by building, as we're translating and digging down into subexpressions, a *symbol table* that basically corresponds to the environment *as it will look when the expression is eventually evaluated*. That symbol table only contains the identifiers themselves, and not the values, of course. (The values bound to the identifiers will only be known when the code evaluates, and we're not evaluating here, just looking at the source code to try to predict the shape of the environment.)

The initial symbol table just is a list of the identifiers in the initial environment. Here are the definitions for `compile_env()` for each Expression node:

```
class EValue (Exp):
    ...

    def compile_env (self,symtable):
        return self


class EPrimCall (Exp):
    ...

    def compile_env (self,symtable):
        exps = [ exp.compile_env(symtable) for exp in self._exps ]
        return EPrimCall(self._prim,exps)


class EIf (Exp):
    ...

    def compile_env (self,symtable):
        condexp = self._cond.compile_env(symtable)
        thenexp = self._then.compile_env(symtable)
        elseexp = self._else.compile_env(symtable)
        return EIf(condexp,thenexp,elseexp)
```

```
class EId (Exp):
    ...

    def compile_env (self,symtable):
        for (index,name) in reversed(list(enumerate(symtable))):
            if name == self._id:
                return EIdIndex(index)
        raise Exception("Error: unbound identifier {}".format(self._id))


class ECall (Exp):
    ...

    def compile_env (self,symtable):
        fun = self._fun.compile_env(symtable)
        args = [ exp.compile_env(symtable) for exp in self._args ]
        return ECall(fun,args)


class EFunction (Exp):
    ...

    def compile_env (self,symtable):
        rec_name = [ self._name ] if self._name else []
        body = self._body.compile_env(symtable+rec_name+self._params)
        return EFunction(self._params,body,self._name)
```

To see how this is used, here is how to invoke `compile_env()` in the shell to transform the code before evaluating it:

```
def shell_compenv ():
    ...

    env = initial_env_compenv()
    symtable = [ name for (name,_) in env ]

    ...

            result = parse(inp)

            if result["result"] == "expression":
                exp = result["expr"]
                with Timer() as timer:
```

```
                cps_exp = exp.cps(EFunction(["x"],EId("x")))
                comp_exp = cps_exp.compile_env(symtable)
                v = comp_exp.eval(env)
                print "Eval time: {}s".format(round(timer.time(),3))
            print v
    ...
```

We get the expression from the parser, we transform it to continuation-passing style, we transform it via `compile_env()`, and then we evaluate it.

The results are promising:

```
Lecture 8 - REF Compiled Language (defun, define, CPS, compiled env)
#quit to quit, #abs to see abstract representation
ref/env> (let ((sum (function s (x) (if (zero? x) 0 (+ x (s (- x 1)))))))
           (sum 200000))
Eval time: 7.76s
20000100000
```

So we shaved off nearly 1.5 seconds, so about 15% of the evaluation time. Pretty good for something easy to implement.

This is by way of example of how we can improve the efficiency of our interpreter by doing analyses that let us precompute or reduce the amount of work that the interpreter needs to do. This is especially important when evaluating function definitions: functions are parsed and transformed and stored in the environment in this transformed way, and we gain the benefit of the transformation every time we invoke the function.

Other obvious improvements include simplifying expressions involving constants, such as simplifying `(* 2 3)` to 6, or `((function (x) (+ x 1)) a)` to `(+ a 1)`. But rather than looking at such transformations now, let's postpone them until we see compilation.

Introduction to Compilation

The other to improve the efficiency of interpretation is to convert the abstract representation into another representation that is even 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.

The low-level representation used by compiler is often called an *abstract (or virtual) machine*. There are two main classes of abstract machines used: stack machines, and register machines. One difference between abstract machines and the abstraction representations we've been using in our interpreters is that code for abstract machines is generally not recursive.

Stack machines are easier to understand, at least at a high level. Here's an example of a simple stack 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). The stack machine consists of a stack that holds integers, and the instructions of the stack machines 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:

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

The sequence of stack machine instructions $\mathsf{1}\ \mathsf{3}\ \mathsf{4}\ \otimes\ \oplus$ 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.

The other kind of abstract machine is a *register machine*. The idea is modeled after CPUs, which execute instructions stored in an array (the memory) and uses registers to store temporary values. This is the representation we will use as the target of our compilation. Next time, we will define a way to translate our abstract representation into the register machine I'll presently introduce.

The register machine 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`. Position in the array are called *addresses*.

The registers of the machine are

- `PC`: holds the address of the instruction to execute next;
- `ADDR`: holds an address

- `ENV`: holds an environment (just an array of values)
- `ARGS`: holds an array of values
- `RESULT`: holds a value

Here are the instructions:

- `RETURN`: stops execution and returns the value in `RESULT`
- `LOAD`: puts the value at address `PC` $+ 1$ in `RESULT`
- `LOAD-ADDR`: puts the integer at `PC` $+ 1$ in `ADDR`
- `LOOKUP`: puts the value in `ENV` at index given by the integer at `PC` $+ 1$ in `RESULT`
- `CLEAR-ARGS`: clears the `ARGS` array
- `PUSH-ARGS`: adds the content of `RESULT` to the end of `ARGS`
- `PRIM-CALL`: calls the primitive operation at `PC` $+ 1$ with arguments given by `ARGS` and puts the result in `RESULT`
- `PUSH-ENV`: adds the content of `RESULT` to the end of `ENV`
- `PUSH-ENV-ARGS`: appends the content of `ARGS` to the end of `ENV`
- `JUMP`: sets `PC` to the content of `ADDR`
- `JUMP-TRUE`: sets `PC` to the content of `ADDR` if `RESULT` contains Boolean value `true`
- `LOAD-FUN`: creates a closure[2] value out the `ADDR` and `ENV` and puts it in `RESULT`
- `LOAD-ADDR-ENV`: pulls out the address and environment in a closure in `RESULT` and puts them in `ADDR` and `ENV`, respectively

After every instruction, `PC` is incremented to point to the next memory location (or the one after if the instruction uses the value at `PC` $+ 1$), unless a jump is involved.

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))`:

[2] In this register machine, a closure is a pair of an address and an environment. Contrast with the definition of a closure in our abstract representation which is a function (parameters and body) and an environment.

```
0000  LOAD
0001  3
0002  PUSH-ARGS
0003  LOAD
0004  4
0005  PUSH-ARGS    # ARGS is now [3,4]
0006  PRIM-CALL
0007  oper_times   # RESULT is now 12
0008  CLEAR_ARGS
0009  PUSH_ARGS
0010  LOAD
0011  1
0012  PUSH_ARGS    # ARGS is now [12,1]
0013  PRIM-CALL
0014  oper_plus    # RESULT IS now 13
0015  RETURN
```

Simple enough.

Here is a more complicated example that computes:

```
(let ((sum_iter (function s (n result)
                  (if (zero? n) result (s (- n 1) (+ n result))))))
  (sum_iter 200000 0))
```

The recursive call is mediated by a closure with address `0008` and an empty environment. That closure is created at address `0008`, and is immediately stored in `ENV` at position 0. The "recursive call" to that closure happens at addresses 51-54, which loads the closure and jumps to the closure's address.

```
0000  LOAD
0001  200000
0002  PUSH-ARGS
0003  LOAD
0004  0
0005  PUSH-ARGS
0006  LOAD-ADDR
0007  8

0008  LOAD-FUN
0009  PUSH-ENV
0010  PUSH-ENV-ARGS
0011  CLEAR-ARGS
0012  LOOKUP
```

```
0013  1
0014  PUSH-ARGS
0015  PRIM-CALL
0016  oper_zero
0017  LOAD-ADDR
0018  55
0019  JUMP-TRUE
0020  CLEAR-ARGS
0021  LOOKUP
0022  1
0023  PUSH-ARGS
0024  LOAD
0025  1
0026  PUSH-ARGS
0027  PRIM-CALL
0028  oper_minus
0029  CLEAR-ARGS
0030  PUSH-ARGS
0031  PUSH-ENV-ARGS
0032  CLEAR-ARGS
0033  LOOKUP
0034  1
0035  PUSH-ARGS
0036  LOOKUP
0037  2
0038  PUSH-ARGS
0039  PRIM-CALL
0040  oper_plus
0041  CLEAR-ARGS
0042  PUSH-ARGS
0043  PUSH-ENV-ARGS
0044  CLEAR-ARGS
0045  LOOKUP
0046  3
0047  PUSH-ARGS
0048  LOOKUP
0049  4
0050  PUSH-ARGS
0051  LOOKUP
0052  0
0053  LOAD-ADDR-ENV
0054  JUMP
```

```
0055  LOOKUP
0056  2
0057  RETURN
```

I claim this is fast to execute. Let's see. I've implemented a simple `execute()` function to execute code in this abstract machine — see the sample code. A function `test()` basically runs the code above, except that the integer at address `0001` is taken from the argument to `test`:

```
>>> test(200000)
Eval time: 2.51s
20000100000
```

Bam. Faster.

A Simple Compiler

Let's look at a simple example. Consider the code `(+ 1 (+ 2 3))`. What is the best code we can write directly in the register machine to compute this? One approach is the following, which respects the fact that evaluation order has you evaluating `(+ 2 3)` before evaluating `(+ 1 ...)`:

```
LOAD
2
PUSH-ARGS
LOAD
3
PUSH-ARGS
PRIM-CALL
oper_plus
PUSH-ENV
CLEAR-ARGS
LOAD
1
PUSH-ARGS
LOOKUP
0
PUSH-ARGS
PRIM-CALL
oper_plus
RETURN
```

(Since the addresses are not important here, I'm dropping them.)

Question: how can we compile `(+ 1 (+ 2 3))` such that we produce the above, or something similar? As a first step, note that we're compiling a CPS translation of this expression:

```
(+ 2 3 (function (x) (+ 1 x *DONE*)))
```

where `*DONE*` is a pre-defined continuation that simply returns its argument. And in fact, we are compiling the abstract representation, so:

```
ECall(EId("+"),[EValue(VInteger 2),
                EValue(VInteger 3),
                EFunction(["x"],ECall(EId("+"),
                                      [EValue(VInteger(1)),
                                       EId("x"),
                                       EId("*DONE*")]))])
```

and in fact, we are compiling a form where every `EId` is converted to an `EIdIndex` with an index into the environment:

```
ECall(EIdIndex(0),[EValue(VInteger 2),
                   EValue(VInteger 3),
                   EFunction(["x"],ECall(EIdIndex(0),
                                         [EValue(VInteger(1)),
                                          EIdIndex(2),
                                          EIdIndex(1)]))])
```

(This assumes an initial environment with only `+` at index 0 and `*DONE*` at index 1.)

Here's a fairly simple compilation function, that converts abstract representation into register machine code. The invariant is that values (`EValue`, `EIdIndex`, and `EFunction`) translate to machine code that finishes with the value in the `RESULT` register. Note that the CPS transformation ensures that the arguments to function calls are either values or identifiers, and that `ECall` and `EIf` never actually return a value, but always call their continuations with their result.

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

$$\mathcal{C}[\![\texttt{EValue}(v)]\!] = \texttt{LOAD}\ v$$

$$\mathcal{C}[\![\texttt{EIdIndex}(i)]\!] = \texttt{LOOKUP}\ i$$

$$
\begin{aligned}
\mathcal{C}[\![\texttt{EFunction}(\mathit{params}, \mathit{body}, \texttt{None})]\!] = \; & \texttt{LOAD-ADDR}\ @\mathit{after} \\
& \texttt{JUMP} \\
& \#\mathit{fun} \\
& \texttt{PUSH-ENV-ARGS} \\
& \mathcal{C}[\![\mathit{body}]\!] \\
& \#\mathit{after} \\
& \texttt{LOAD-ADDR}\ @\mathit{fun} \\
& \texttt{LOAD-FUN}
\end{aligned}
$$

Here, I'm using $@XYZ$ as "the address of $\#XYZ$", instead of hard-wiring addresses into the translation. A pass called *assemble* in the sample code I gave you converts these labels into actual addresses. Of course, every time this compilation function is invoked, a fresh pair of labels for *fun* and *after* need to be used so as not to clash with other compiled functions.

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:

$$
\begin{aligned}
\mathcal{C}[\![\texttt{EFunction}(\mathit{params}, \mathit{body}, \mathit{name})]\!] = \; & \texttt{LOAD-ADDR}\ @\mathit{after} \\
& \texttt{JUMP} \\
& \#\mathit{fun} \\
& \texttt{LOAD-FUN} \\
& \texttt{PUSH-ENV} \\
& \texttt{PUSH-ENV-ARGS} \\
& \mathcal{C}[\![\mathit{body}]\!] \\
& \#\mathit{after} \\
& \texttt{LOAD-ADDR}\ @\mathit{fun} \\
& \texttt{LOAD-FUN}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}[\![\texttt{ECall}(f, [e_1, \ldots, e_k])]\!] = &\texttt{CLEAR-ARGS}\\
&\mathcal{C}[\![e_1]\!]\\
&\texttt{PUSH-ARGS}\\
&\ldots\\
&\mathcal{C}[\![e_k]\!]\\
&\texttt{PUSH-ARGS}\\
&\mathcal{C}[\![f]\!]\\
&\texttt{LOAD-ADDR-ENV}\\
&\texttt{JUMP}
\end{aligned}
$$

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

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

```
 CLEAR-ARGS
 LOAD
 2
 PUSH-ARGS
 LOAD
 3
 PUSH-ARGS
 LOAD-ADDR
 @A6
 JUMP
#B7
 PUSH-ENV-ARGS
 CLEAR-ARGS
 LOAD
 1
 PUSH-ARGS
 LOOKUP
```

```
 4
 PUSH-ARGS
 LOOKUP
 0
 PUSH-ARGS
 LOOKUP
 1
 LOAD-ADDR-ENV
 JUMP
#A6
 LOAD-ADDR
 @B7
 LOAD-FUN
 PUSH-ARGS
 LOOKUP
 1
 LOAD-ADDR-ENV
 JUMP
```

(Again, this is using `@A6` as the address of label `#A6`, rather than explicit addresses, so that the code is easier to understand.)

This doesn't look so great. That's a lot of code. And if we run some benchmarking, we see that we're not doing much better than the interpreted version with indices into the environment instead of identifier lookup. Here is the baseline, running the sum of all numbers up to 200000 in the interpreter that compiles all `EId`s to `EIdIndex`es:

```
Lecture 8 - REF Compiled Language (defun, define)
#quit to quit, #abs to see abstract representation
ref/env> (let ((sum (function s (n result)
                        (if (zero? n)
                            result
                            (s (- n 1) (+ result n))))))
            (sum 200000 0))
Eval time: 6.395s
20000100000
```

Here is the version that compiles and then executes:

```
Lecture 9 - REF Compiled Language (defun, define)
#quit to quit, #abs to see abstract representation and code
ref/comp> (let ((sum (function s (n result)
                         (if (zero? n)
```

```
                                result
                                (s (- n 1) (+ result n))))))
                (sum 200000 0))
Eval time: 5.846s
20000100000
```

Better, but not much.

If we compare this code with the code we wrote by hand, we note that in our hand-written code, we called primitive operations directly. Here, we call the closure `+` that's in the environment, which holds the address of some pre-compiled code that does the actual call to the primitive `oper_plus` operation. (The pre-compiled code was written by hand.)

So one way we could improve things is to recognize when we're making a call to a primitive operation. We can do so by recognizing when the index of an identifier in function call position is the index corresponding to a primitive function. We can figure that out because we know what the initial environment looks like.

If we assume that $oper[i]$ holds the primitive operation function that corresponds to index i, then we can write a special case of $\mathcal{C}[\![-]\!]$ when the function call is an index to a primitive operation (the previous compilation function applies in all other cases):

$$
\begin{aligned}
\mathcal{C}[\![\texttt{ECall(EIdIndex(}i_p\texttt{)}, [e_1, \ldots, e_k])]\!] = \; & \texttt{CLEAR-ARGS} \\
& \mathcal{C}[\![e_1]\!] \\
& \texttt{PUSH-ARGS} \\
& \ldots \\
& \mathcal{C}[\![e_{k-1}]\!] \\
& \texttt{PUSH-ARGS} \\
& \texttt{PRIM-CALL } oper[i_p] \\
& \texttt{CLEAR-ARGS} \\
& \texttt{PUSH-ARGS} \\
& \mathcal{C}[\![e_k]\!] \\
& \texttt{LOAD-ADDR-ENV} \\
& \texttt{JUMP}
\end{aligned}
$$

With this addition to the compilation process, the code now generated for `(+ 1 (+ 2 3))` becomes:

```
CLEAR-ARGS
LOAD
2
PUSH-ARGS
LOAD
```

```
 3
 PUSH-ARGS
 PRIM-CALL
 oper_plus
 CLEAR-ARGS
 PUSH-ARGS
 LOAD-ADDR
 @A66
 JUMP
#B67
 PUSH-ENV-ARGS
 CLEAR-ARGS
 LOAD
 1
 PUSH-ARGS
 LOOKUP
 4
 PUSH-ARGS
 PRIM-CALL
 oper_plus
 CLEAR-ARGS
 PUSH-ARGS
 LOOKUP
 3
 LOAD-ADDR-ENV
 JUMP
#A66
 LOAD-ADDR
 @B67
 LOAD-FUN
 LOAD-ADDR-ENV
 JUMP
```

This looks much better. If we execute this code, we are faster, but still not great.

```
Lecture 9 - REF Compiled Language (defun, define)
#quit to quit, #abs to see abstract representation and code
ref/comp> (let ((sum (function s (n result)
                          (if (zero? n)
                              result
                              (s (- n 1) (+ result n))))))
             (sum 200000 0))
Eval time: 5.169s
```

```
20000100000
```

Again, let's compare to our hand-written code. The one difference that remains is that in this code we create a function (corresponding to continuation `(function (x) (+ 1 x *DONE))`) just to call it immediately. We can save a lot of work by never creating the function in the first place! We can identify that special case and compile it differently. (the previous compilation functions apply in all other cases):

$$
\begin{aligned}
\mathcal{C}[\![\texttt{ECall}(\texttt{EIdIndex}(i_p), [e_1, \ldots, e_{k-1}, \texttt{EFunction([}p\texttt{]}, body)])]\!] = \; & \texttt{CLEAR-ARGS} \\
& \mathcal{C}[\![e_1]\!] \\
& \texttt{PUSH-ARGS} \\
& \ldots \\
& \mathcal{C}[\![e_{k-1}]\!] \\
& \texttt{PUSH-ARGS} \\
& \texttt{PRIM-CALL}\ oper[i_p] \\
& \texttt{PUSH-ENV} \\
& \mathcal{C}[\![body]\!]
\end{aligned}
$$

The code produced now is quite close to the one we wrote by hand:

```
CLEAR-ARGS
LOAD
2
PUSH-ARGS
LOAD
3
PUSH-ARGS
PRIM-CALL
oper_plus
PUSH-ENV
CLEAR-ARGS
LOAD
1
PUSH-ARGS
LOOKUP
4
PUSH-ARGS
PRIM-CALL
oper_plus
CLEAR-ARGS
PUSH-ARGS
```

```
LOOKUP
3
LOAD-ADDR-ENV
JUMP
```

And the benchmarks show a running time that is a third of the interpreted one, which is as good as we might expect given that we are running the compiled code using Python!

```
Lecture 9 - REF Compiled Language (defun, define)
#quit to quit, #abs to see abstract representation and code
ref/comp> (let ((sum (function s (n result)
                          (if (zero? n)
                              result
                              (s (- n 1) (+ result n))))))
             (sum 200000 0))
Eval time: 2.696s
20000100000
```