Analyzing Evaluator 28

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

August 10, 2010

To load the analying evaluator, say:

```
(load "~cs61a/lib/analyze.scm")
```

1 Overview: Separating Analysis from Execution

`Eval` takes two arguments, an expression and an environment. Of those, the expression argument is (obviously!) the same every time we revisit the same expression, whereas the environment will be different each time. For example, when we compute `(fact 3)` we evaluate the body of fact in an environment in which num has the value 3. That body includes a recursive call to compute `(fact 2)`, in which we evaluate the same body, but now in an environment with num bound to 2.

Our plan is to look at the evaluation process, find those parts which depend only on `exp` and not on `env`, and do those only once. The procedure that does this work is called `analyze`.

What is the result of `analyze`? It has to be something that can be combined somehow with an environment in order to return a value. The solution is that `analyze` returns a procedure that takes only `env` as an argument, and does the rest of the evaluation.

Instead of

```
(eval exp env) ==> value
```

we now have

```
1. (analyze exp) ==> exp-procedure
2. (exp-procedure env) ==> value
```

When we evaluate the same expression again, we only have to repeat step 2. What we're doing is akin to memoization, in that we remember the result of a computation to avoid having to repeat it. The difference is that now we're remembering something that's only part of the solution to the overall problem, instead of a complete solution.

We can duplicate the effect of the original eval this way:

```
(define (eval exp env)
  ((analyze exp) env))
```

2 Motivation

Suppose we've defined the factorial function as follows:

```
(define (fact num)
  (if (= num 0)
      1
      (* num (fact (- num 1)))))
```

What happens when we compute `(fact 3)`?

```
eval (fact 3)
  self-evaluating? ==> #f    if-alternative ==> (* num (fact (- num 1)))
  variable? ==> #f             eval (* num (fact (- num 1)))
  quoted? ==> #f                 self-evaluating? ==> #f
  assignment? ==> #f             ...
  definition? ==> #f             list-of-values (num (fact (- num 1)))
  if? ==> #f                     ...
  lambda? ==> #f                 eval (fact (- num 1))
  begin? ==> #f                    ...
  cond? ==> #f                     apply <procedure fact> (2)
  application? ==> #t                eval (if (= num 0) ...)
  eval fact
    self-evaluating? ==> #f
    variable? ==> #t
    lookup-variable-value ==> <procedure fact>
    list-of-values (3)
      eval 3 ==> 3
    apply <procedure fact> (3)
      eval (if (= num 0) ...)
        self-evaluating? ==> #f
        variable? ==> #f
        quoted? ==> #f
        assignment? ==> #f
        definition? ==> #f
        if? ==> #t
          eval-if (if (= num 0) ...)
            if-predicate ==> (= num 0)
              eval (= num 0)
                self-evaluating? ==> #f
                ...
```

Four separate times, the evaluator has to examine the procedure body, decide that it's an `if` expression, pull out its component parts, and evaluate those parts (which in turn involves deciding what type of expression each part is).

This is one reason why interpreted languages are so much slower than compiled languages: The interpreter does the syntactic analysis of the program over and over again. The compiler does the analysis once, and the compiled program can just do the part of the computation that depends on the actual values of variables.

3 The Implementation Details.

Analyze has a structure similar to that of the original eval:

```
(define (eval exp env)                (define (analyze exp)
(cond ((self-evaluating? exp)           (cond ((self-evaluating? exp)
       exp)                                    (analyze-self-eval exp))
      ((variable? exp)                        ((variable? exp)
       (lookup-var-val exp env))               (analyze-var exp))
      ...                                     ...
      ((foo? exp) (eval-foo exp env))         ((foo? exp) (analyze-foo exp))
      ...))                                   ...))
```

The difference is that the procedures such as `eval-if` that take an expression **and** an environment as arguments have been replaced by procedures such as `analyze-if` that take **only** the expression as argument.

How do these analysis procedures work? As an intermediate step in our understanding, here is a version of `analyze-if` that exactly follows the structure of `eval-if` and doesn't save any time:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
(define (analyze-if exp)
  (lambda (env)
    (if (true? (eval (if-predicate exp) env))
        (eval (if-consequent exp) env)
        (eval (if-alternative exp) env))))
```

This version of `analyze-if` returns a procedure with `env` as its argument, whose body is exactly the same as the body of the original `eval-if`. Therefore, if we do:

```
((analyze-if some-if-expression) some-environment)
```

the result will be the same as if we'd said

```
(eval-if some-if-expression some-environment)
```

in the original metacircular evaluator.

But we'd like to improve on this first version of `analyze-if` because it doesn't really avoid any work. Each time we call the procedure that `analyze-if` returns, it will do all of the work that the original `eval-if` did.

The first version of `analyze-if` contains three calls to `eval`. Each of those calls does an analysis of an expression and then a computation of the value in the given environment. What we'd like to do is split each of those `eval` calls into its two separate parts, and do the first part only once, not every time:

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

In this final version, the procedure returned by `analyze-if` doesn't contain any analysis steps. All of the components were already analyzed before we call that procedure, so no further analysis is needed.

The biggest gain in efficiency comes from the way in which `lambda` expressions are handled. In the original metacircular evaluator, leaving out some of the data abstraction for clarity here, we have:

```
(define (eval-lambda exp env)
  (list 'procedure exp env))
```

The evaluator does essentially nothing for a lambda expression except to remember the procedure's text and the environment in which it was created. But in the analyzing evaluator we *analyze* the body of the procedure; what is stored as the representation of the procedure *does not include its text*! Instead, the evaluator represents a procedure in the metacircular Scheme as a procedure in the underlying Scheme, along with the formal parameters and the defining environment.

3.1 Question:

The analyzing evaluator turns an expression such as:

```
(if A B C)
```

into a procedure

```
(lambda (env)
  (if (A-execution-procedure env)
      (B-execution-procedure env)
      (C-execution-procedure env)))
```

This may seem like a step backward; we're trying to implement `if` and we end up with a procedure that does an `if`. Isn't this an infinite regress?

4 Conclusion[2]

The syntactic analysis of expressions is a large part of what a compiler does. In a sense, this analyzing evaluator is a compiler! It compiles Scheme into Scheme, so it's not a very useful compiler, but it's really not that much harder to compile into something else, such as the machine language of a particular computer.

A compiler whose structure is similar to this one is called a *recursive descent* compiler. Today, in practice, most compilers use a different technique (called a stack machine) because it's possible to automate the writing of a parser that way. (I mentioned this earlier as an example of data-directed programming.) But if you're writing a parser by hand, it's easiest to use recursive descent.

[2]Fun Fact. This lecture is the last technical section, so it's a conclusion in a meta sense too!