Brandeis University

COMPUTER SCIENCE (1)21B (SPRING TERM, 2017)

Structure and Interpretation of Computer Programs

Separating Syntactic Analysis from Execution: Steps Towards a Compiler

The basic metacircular evaluator needs to re-evaluate syntax over and over. For example, in

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

when we evaluate `(fact 100)`, the body is syntactically evaluated 100 times to discover—surprise, surprise!—it is an `if` expression.

While the execution of the `if` needs to take place that many times, surely the *syntax analysis* does not. We discuss here a way to analyze the syntax of a program *exactly once*.

The evaluator is now defined as

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

where procedure `analyze` takes an expression, and returns a *procedure* which, when given an environment, produces an answer. The code of the procedure involves no syntax analysis.

For example, `(analyze '10)` produces (essentially) the procedure derived from `(lambda (env) 10)`; analogously, `(analyze 'x)` yields the procedure derived from

```
(lambda (env) (lookup-variable-value 'x env))
```

Here's a brief way of remembering what the syntax analyzer does: let E be a quoted list structure representing a Scheme program you would like to evaluate, and let P be the procedure, in the *underlying* Scheme system, which results from evaluating `(analyze` E`)`.

Notice that P involves *no* call to `eval`, which is the only way that syntax analysis (dispatching on syntax) can occur.

Furthermore, for any environment *env*, we assert that `(eval` E ***env***`)` and `(`P ***env***`)` evaluate to the same thing.

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp)
         (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp))))
```

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application
        (fproc env)
        (map (lambda (aproc) (aproc env))
             aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            args
            (procedure-environment proc))))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))

(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

```
(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)))))

(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

To understand what the analyzer does, have another look at `analyze-if`:

```
(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)))))
```

The returned value is a procedure derived from evaluating

```
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

in an environment where `pproc`, `cproc` and `aproc` are defined.

What would the code for this procedure (as well as those for other syntactic forms) look like if the `lambda`-expressions for `pproc`, `cproc` and `aproc` were instead *substituted*? **This gives us a compiler:**

```
(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)))))
```

Note: `(pp ...)` is the expression Scheme evaluates to produce subsequent output.

```
(pp (analyze '(if p 0 1)))
(lambda
 (env)
 (if
  (true?
   ((lambda (env) (lookup-variable-value 'p env)) env))
  ((lambda (env) 0) env)
  ((lambda (env) 1) env)))
;No value
```

Notice the redundancies, which we clean up with procedure `clean`: we rewrite occurrences of `((lambda (env)` E`) env)` to E:

```
(define (clean code)
  (if (or (symbol? code) (number? code) (null? code))
      code
      (let ((code (map clean code)))
        (if (and (lambda? (car code))
                 (equal? '(env) (cadar code))
                 (equal? '(env) (cdr code)))
            (if (null? (cdddar code))
                (caddar code)
                (cddar code))
            code))))

(pp (clean (analyze '(if p 0 1))))
(lambda (env)
 (if (true? (lookup-variable-value 'p env)) 0 1))
;No value
```

```
(pp (clean (analyze
  '((lambda (x y) (* x y)) 10 20))))
(lambda (env)
 (execute-application
  (make-procedure '(x y)
   (lambda (env)
    (execute-application
     (lookup-variable-value '* env)
     (list (lookup-variable-value 'x env)
           (lookup-variable-value 'y env))))
   env)
  (list 10 20)))
;No value
```

```
(pp (clean (analyze
  '(lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))))
(lambda (env)
 (make-procedure '(n)
  (lambda (env)
   (if
    (true?
     (execute-application
      (lookup-variable-value '= env)
      (list (lookup-variable-value 'n env) 0)))
    1
    (execute-application
     (lookup-variable-value '* env)
     (list
      (lookup-variable-value 'n env)
      (execute-application
       (lookup-variable-value 'fact env)
       (list
        (execute-application
         (lookup-variable-value '- env)
         (list (lookup-variable-value 'n env) 1))))))))
  env))
;No value
```

```
(pp (clean (analyze
'(begin (define make-counter (lambda (n)
          ((lambda (count)
             (lambda () (set! count (1+ count)) count))
           n)))
        (define c (make-counter 1))
        (c)
        (c)))))
(lambda (env)
 (begin (define-variable! 'make-counter
          (make-procedure '(n)
           (lambda (env)
            (execute-application
             (make-procedure '(count)
              (lambda (env)
               (make-procedure '()
                (lambda (env)
                 ((begin (set-variable-value! 'count
                                (execute-application
                                 (lookup-variable-value '1+ env)
                                  (list (lookup-variable-value
                                         'count env)))
                                env)
                               'ok)
                  (lookup-variable-value 'count env)))
                 env))
              env)
             (list (lookup-variable-value 'n env))))
           env)
          env)
```

```
'ok)
```

```
  (begin (define-variable! 'c
           (execute-application
             (lookup-variable-value 'make-counter env)
             (list 1))
           env)
           'ok)
  (execute-application
   (lookup-variable-value 'c env) (list))
  (execute-application
   (lookup-variable-value 'c env) (list))))
;No value
```

What is revealed by the *compiled (target) code* that is not revealed by the *source code*? Mostly, information about the *environment*.

A variable reference `x` is replaced by `(lookup-variable 'x env)`; the structure of the target code tells us *which* environment is used for the lookup.

Little if any information is revealed about the *control* aspect of the target code—control is *naively inherited* from the underlying Scheme system.

For something better, we have to wait for an *explicit-control evaluator*, a coming attraction...