SICP sections 4.1.6 - 4.1.7

December 14, 2007 at 11:09 Tags SICP

Exercise 4.16

Changing `lookup-variable-value`:

```
(defun lookup-variable-value (var env)
  (labels (
      (env-loop (env)
        (when (> *evaluator-debug-level* 2)
          (format t "scanning env: ~a~%" env))
        (if (eq env the-empty-environment)
          (error "Unbound variable ~a" var))
          (let* ( (result (find-binding-in-frame (first-frame env) var))
                  (found (car result))
                  (value (cdr result)))
            (if found
              (if (eq value '*unassigned*)
                (error "Using an unassigned var ~a" var)
                value)
              (env-loop (enclosing-environment env))))))
    (env-loop env)))
```

The procedure `scan-out-defines`:

```
(defun scan-out-defines (body)
  "Takes a procedure body and returns an equivalent
  one that has no internal definition, by
  transforming:

  (lambda <vars>
    (define u <e1>)
    (define v <e2>)
    <e3>)

  Into:

  (lambda <vars>
    (let ((u '*unassigned*)
          (v '*unassigned*))
      (set! u <e1>)
      (set! v <e2>)
      <e3>))"

  (let ((defines '())
        (non-defines '()))
    (dolist (exp body)
      (if (definition? exp)
        (push exp defines)
        (push exp non-defines)))
    (if (null defines)
      body
      (progn
        ; The order of non-defines is important, so
        ; we restore the order that was reversed by
        ; using -push-. The order of defines, OTOH,
        ; is not important
        ;
        (nreverse non-defines)
        (list
          (make-let
            (mapcar
              #'(lambda (def)
                  (list
                    (definition-variable def)
                    (make-quoted '*unassigned*)))
                defines)
            (make-begin
              (append
                (mapcar
                  #'(lambda (def)
                      (make-assignment
                        (definition-variable def)
                        (definition-value def)))
                    defines)
                non-defines))))))))
```

And finally, installing `scan-out-defines` inside `make-procedure`:

```
(defun make-procedure (parameters body env)
  (list
    'procedure
    parameters
    (scan-out-defines body)
    env))
```

IMHO `make-procedure` is a better place, because it is done once when the procedure is defined and not in each application of the procedure.

Exercise 4.17

There is an extra frame in the scanned-out version, because we added a `let` which is, to remind you, a disguised `lambda`. Each `lambda` creates a sub-environment of its own to give local values to its arguments. This is the environment in which `e3` is evaluated.

Since the `let` is defined to completely enclose the body of the outer `lambda`, its new environment doesn't really add any necessary scope that can't be defined in the outer lambda.

Alternatively, we could just regroup the statements in the outer `lambda` to move all internal definitions to the top. This would work because when a definition is evaluated, the body of the lambda is *not* evaluated. It is evaluated only when the procedure is actually applied. Therefore, it can contain references to other definitions that are defined after it. What is important is that no executable line in the code is evaluated before all those internal definitions were defined.

Exercise 4.18

This won't work. When `dy` is defined, it expects `y` to be already defined. But `y` was only assigned to `a`, not yet to `u`. The definition of `dy` is transformed to `b` which evaluates to `e2` in the `let` form and expects `u` to be there.

For the same reason this *will* work in the original scanned-out version[1], because there the assignments to `u` and `v` are direct, and not through other proxy values.

Exercise 4.19

If we've decided that all definitions must be simultaneous, then of course Eva is right, because the behavior she describes is the one that fits the requirement of simultaneity.

We must somehow sort the definitions in a way that for each pair of definitions X and Y, if X uses Y, Y's assignment must come before X. Fortunately, this is a common request and the *Topological sort* algorithm was made up just for such uses. So we can sort the internal definitions topologically by usage and end up with the correct order.

There is a problem though – things aren't as trivial as they appear at first. Consider recursive definitions. Obviously, the following is wrong and must result in an error:

```
(define (f x)
  (define b (f a))
  (define a (g b))
  (+ a b))
```

Because of the mutual reference, this can't be resolved. However, what about delayed (lazy) evaluation:

```
(define (f x)
  (define b (f (delay a)))
  (define a (g b))
  (+ a b))
```

In principle, there's no problem with this code, and in fact the `solve` function mentioned in the previous exercise does just this.

So, we must also be able to resolve recursive definitions, and pay attention to delayed evaluation. All in all, it is far from simple to implement.

Exercise 4.20

a. The implementation is somewhat similar to `scan-out-defines`:

```
(defun letrec? (exp) (tagged-list? exp 'letrec))

(defun letrec->let (exp)
  "Transforms into a let, such that all variables
  are created with a let and then assigned their
  values with set!"
  ; Note that since letrec is identical in syntax to
  ; let, we can freely use the let- accessors.
  ;
  (let ((initforms (let-initforms exp))
        (body (let-body exp)))
    (make-let
      (mapcar ; initforms
        #'(lambda (initform)
            (list
              (car initform)
              (make-quoted '*unassigned*)))
        initforms)
      (make-begin ; body
        (append
          (mapcar
            #'(lambda (initform)
                (make-assignment
                  (car initform)
                  (cadr initform)))
              initforms)
            (list body))))))
```

b. To understand better what happens under the hood, let's see how a`let` is transformed:

```
(let->combination
  '(let ((even?
            (lambda (n)
              (if (= n 0)
                  true
                  (odd? (- n 1)))))
           (odd?
            (lambda (n)
              (if (= n 0)
                  false
                  (even? (- n 1))))))
      1))
=>
((LAMBDA (EVEN? ODD?) 1)
    (LAMBDA (N) (IF (= N 0) TRUE (ODD? (- N 1))))
    (LAMBDA (N) (IF (= N 0) FALSE (EVEN? (- N 1)))))
```

The environment of the outer `lambda` which was created from `let` isn't the enclosing environment of the internal `lambda@s. Rather, these are applied as arguments. Therefore, the body of the @lambda` representing `even?` doesn't see the variable `odd?`.

On the other hand, consider the expansion of`letrec`:

```
(let->combination
  (letrec->let
    '(letrec  ((even?
                (lambda (n)
                  (if (= n 0)
                      true
                      (odd? (- n 1)))))
              (odd?
                (lambda (n)
                  (if (= n 0)
                      false
                      (even? (- n 1))))))
        1))))
=>
((LAMBDA (EVEN? ODD?)
  (BEGIN
    (SET! EVEN? (LAMBDA (N) (IF (= N 0) TRUE (ODD? (- N 1)))))
    (SET! ODD? (LAMBDA (N) (IF (= N 0) FALSE (EVEN? (- N 1)))))
    1))
 '*UNASSIGNED* '*UNASSIGNED*)
```

Now both `lambda@s are defined within the same lexical scope under the outer @lambda` and can see each other.

Exercise 4.21

a.

```
(interpret
  '((lambda (n)
     ((lambda (fact)
        (fact fact n))
      (lambda (ft k)
        (if (= k 1)
            1
            (* k (ft ft (- k 1)))))))
   10))
=>
3628800
```

We can understand this in stages. The outermost function (let's call it `lambda-n`) takes one argument and returns a result. That's easy.

Now, `lambda-fact` takes an argument and applies it to itself and `n`. It is in fact applied to the function `lambda-ft` which takes two arguments, and applies the first to itself with the second argument decreased.

So, when `lambda-fact` is called, it is given `lambda-ft` and `n` as an argument. It calls `lambda-ft`, giving it `lambda-ft` and `n` as arguments. `lambda-ft`, in turn, can now call itself recursively, because it has received itself as an argument. This way we allow unnamed functions to call themselves – by passing them as an argument to themselves.

Computing Fibonacci is very similar, based on the same trick of a function that takes an argument and applies it to itself.

```
(interpret
'((lambda (n)
  ((lambda (fib)
      (fib fib n))
    (lambda (ft k)
      (if (< k 2)
        1
        (+  (ft ft (- k 1))
            (ft ft (- k 2)))))))
 6))
=>
13
```

b.

```
(interpret
'(define (f x)
    ((lambda (even? odd?)
        (even? even? odd? x))
      (lambda (ev? od? n)
        (if (= n 0)
          true
          (od? ev? od? (- n 1))))
      (lambda (ev? od? n)
        (if (= n 0)
          false
          (ev? ev? od? (- n 1)))))))

(interpret '(f 6))
=> T

(interpret '(f 13))
=> NIL
```

Section 4.1.7

Here is the code of the optimized evaluator, translated into CL:

```
(defun eval-opt. (exp env)
  (funcall (analyze. exp) env))

(defun 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))
        (t
          (error "Unknown expression in EVAL: " exp))))

(defun analyze-self-evaluating (exp)
  (lambda (env) exp))

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

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

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

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

(defun analyze-if (exp)
  (let ((pproc (analyze. (if-predicate exp)))
        (cproc (analyze. (if-consequent exp)))
        (aproc (analyze. (if-alternative exp))))
    (lambda (env)
      (if (true? (funcall pproc env))
          (funcall cproc env)
          (funcall aproc env)))))

(defun analyze-lambda (exp)
  (let ((vars (lambda-parameteres exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env)
      (make-procedure vars bproc env))))

(defun analyze-sequence (exps)
  (labels (
      (sequentially (proc1 proc2)
        (lambda (env)
          (funcall proc1 env)
          (funcall proc2 env)))
      (sloop (first-proc rest-procs)
        (if (null rest-procs)
          first-proc
          (sloop
            (sequentially first-proc (car rest-procs))
            (cdr rest-procs)))))
    (let ((procs (mapcar #'analyze. exps)))
      (if (null procs)
        (error "Empty sequence in ANALYZE-SEQUENCE"))
      (sloop (car procs) (cdr procs)))))

(defun analyze-application (exp)
  (let ((fproc (analyze. (operator exp)))
        (aprocs (mapcar #'analyze. (operands exp))))
    (lambda (env)
      (execute-application
        (funcall fproc env)
        (mapcar
          #'(lambda (aproc) (funcall aproc env))
          aprocs)))))

(defun execute-application (proc args)
  (cond ((primitive-procedure? proc)
          (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         (funcall
          (procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            args
            (procedure-env proc))))
        (t
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION ~a"
          proc))))

(defun interpret (exp)
  (eval-opt. exp the-global-environment))
```

It uses the primitives defined in the original evaluator, with one small exception. The`scan-out-defines` call in `make-procedure` must be disabled, because the underlying implementation of a procedure was changed.

Exercise 4.22

We just have to add this to `analyze.`:

```
  ((let? exp)
    (analyze. (let->combination exp)))
```

`let->combination` transforms the `let` form into a `lambda` form which `analyze.` knows how to handle.

Exercise 4.23

Alyssa's version defers the call to `execute-sequence` to the runtime, while the version in the text unrolls the calls at analysis time.

Exercise 4.24

I'll perform a benchmark using the factorial function. With the original evaluator:

```
(interpret
  '(define (factorial n)
    (if (= n 1)
      1
      (* (factorial (- n 1)) n))))

(time
  (dotimes (i 1000 t)
    (interpret
    '(factorial 50))))
=>
Real time: 32.875 sec.
Run time: 32.078125 sec.
```

And with the optimized evaluator:

```
(interpret
  '(define (factorial2 n)
    (if (= n 1)
      1
      (* (factorial2 (- n 1)) n))))

(time
  (dotimes (i 1000 t)
    (interpret
      '(factorial2 50))))
=>
Real time: 24.078125 sec.
Run time: 23.8125 sec.
```

This suggests that the original evaluator spends around 30% of its time analyzing the same expressions all over again.

Updated code

The latest version of the evaluator code: evaluator.lisp, evaluator_testing.lisp, evaluator_optimized_analysis.lisp

[1] Given that we've implemented the `delay` function, which is unusual in that it doesn't evaluate its argument.

For comments, please send me an email.

Back to top