SICP sections 4.1.3 - 4.1.5

December 08, 2007 at 10:16 Tags SICP

Exercise 4.11

Since we have the `frame` abstraction, it suffices to modify only it:

```
(defun make-frame (vars vals)
  (if (/= (length vars) (length vals))
    (error "MAKE-FRAME length mismatch")
    (labels (
        (build-pairs (vars vals)
          (if (null vars)
            '()
            (cons
              (cons (car vars) (car vals))
              (build-pairs (cdr vars) (cdr vals))))))
      (cons 'frame (build-pairs vars vals)))))

(defun add-binding-to-frame! (var val frame)
  (push (cons var val) (cdr frame)))

(defun frame-variables (frame)
  (mapcar #'car (cdr frame)))

(defun frame-values (frame)
  (mapcar #'cdr (cdr frame)))
```

We can now use the test suite (see previous post) to quickly verify that with this change the evaluator keeps functioning well. Bah… it doesn't!

After some investigation, it appears that the problem occurs when modifying variables in the environment, an a quick investigation of the code shows why. The function `set-variable-value!` breaks the `frame` abstraction, by assuming its structure and modifying the frame accordingly. Fixing this abstraction leak will be left as an exercise to diligent readers.

Exercise 4.12

For this exercise we go back to the original implementation of frames, as a list of variables / list of values pair. To generalize the traversal of environments, I will first empower the frame abstraction somewhat. This will also fix the abstraction leak we discovered in the previous exercise.

There's no reason why an environment should be responsible for looking up values in frames. Frames can do that. So we'll add these functions to the frame abstraction:

```
(defun find-binding-in-frame (frame var)
  "Looks up the variable in the frame.
  Returns a pair: if the -car- of the pair is t,
  then the variable was found and it's in the -cdr-
  of the pair. If the -car- of the pair is nil,
  then the variable was not found"
  (labels (
      (scan (vars vals)
        (cond ((null vars)
                (cons nil nil))
              ((eq var (car vars))
                (cons t (car vals)))
              (t
                (scan (cdr vars) (cdr vals))))))
    (scan (frame-variables frame)
          (frame-values frame))))

(defun set-binding-in-frame! (frame var val)
  "Sets the variable to the value in the frame.
  Returns t if the variable was found and modified,
  nil otherwise."
  (labels (
      (scan (vars vals)
        (cond ((null vars)
                nil)
              ((eq var (car vars))
                (setf (car vals) val)
                t)
              (t
                (scan (cdr vars) (cdr vals))))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

And now, the environment-level functions are much shorter:

```
(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)))
            (if (car result)
              (cdr result)
              (env-loop (enclosing-environment env))))))
    (env-loop env)))

(defun set-variable-value! (var val env)
  (labels (
      (env-loop (env)
        (if (eq env the-empty-environment)
          (error "Unbound variable ~a" var)
          (if (set-binding-in-frame! (first-frame env) var val)
            t
            (env-loop (enclosing-environment env))))))
    (env-loop env)))

(defun define-variable! (var val env)
  (let ((frame (first-frame env)))
    (if (set-binding-in-frame! frame var val)
      t
      (add-binding-to-frame! frame var val))))
```

I have a nagging feeling that even more of the functionality can be abstracted out, especially if we put macros to use. However, this looks good enough so I'll just move on.

Exercise 4.13

I think it makes most sense to unbind the variable only in its current frame, i.e. in its immediate lexical environment. Unbinding variables in enclosing environments doesn't sound like good encapsulation to me.

First, the function that unbinds a variable at the frame level:

```
(defun unbind-var-in-frame! (frame var)
  "Unbinds a variable in the frame."
  (let ((vars (frame-variables frame))
        (vals (frame-values frame))
        (new-vars '())
        (new-vals '()))
    (loop
      for a-var in vars
      for a-val in vals
      do
      (unless (eq a-var var)
        (push a-var new-vars)
        (push a-val new-vals)))
    (setf (car frame) new-vars)
    (setf (cdr frame) new-vals)))
```

I've decided to employ the almighty `loop` macro here. The functions turns the order of variables in the frame around, but this has no bad effect on its functionality. Now, to add the function at the environment level, plus a recognizer for the new form:

```
(defun make-unbound? (exp)
  (tagged-list? exp 'make-unbound!))

(defun unbind-variable! (exp env)
  (unbind-var-in-frame! (first-frame env) (cadr exp)))
```

Finally, an addition to `eval.`:

```
  ((make-unbound? exp)
    (unbind-variable! exp env))
```

Exercise 4.14

It is clear why Eva's `map` works. Eva has written it in the interpreted Scheme, so there is no reason for it not to work, since the interpreter is quite complete and can run such functions.

On the other hand, Louis committed a mistake by mixing up the interpreted language with the driving language in which the interpreter is written. Installing `map` as a primitive can't work, because `map` takes a function as an argument and applies it. The function it expects must be in the *driving language*, and it is applied with the native `apply`, of Common Lisp in our case. On the other hand, in Louis's interpreted Scheme code, he hands in Scheme functions to `map`. These functions must be applied with `apply.`, and not with the native `apply` of Common Lisp.

Exercise 4.15

This is the *Halting theorem*. The authors provide enough clues to prove it very easily. Wher`(try try)` is executed, `(halts? p p)` is called. `try` is substituted for `p` so it's actually `(halts? try try)`. Now, this can either return true or false.

Suppose it returns true. Then, `try` enters an endless loop – so it obviously doesn't halt, while`halts?` returned true. This is a contradiction.

Suppose now it returns false. Then `try` halts and returns `halted`, which is again a contradiction.

Therefore, there can be no such function as `halts?` which adheres to its spec.

The updated evaluator files: evaluator.lisp, evaluator_testing.lisp

For comments, please send me an email.

Back to top