SICP section 5.4

April 04, 2008 at 08:39 Tags SICP

When I began writing the explicit-control evaluator code for this section, I ran into a minor language incompatibility. While I've used CL to implement the meta-circular evaluator of chapter 4, I've employed Scheme for the implementation of the register machine simulator earlier in this chapter. This means that I can't use the meta-circular evaluator's primitive functions directly in my register-machine code.

So I've downloaded the code from chapter 4 of the book fromhere. The specific file that is included in my code is this one – I've cleaned it up and modified it a little to be compatible with PLT Scheme, and provide only the required functionality.

The explicit-control evaluator itself is available here. I've modified it a little, adding some functionality for non-interactive evaluation. Specifically, this code:

```
non-interactive-eval
  (perform (op initialize-stack))
  (test (op null?) (reg machine-args))
  (branch (label machine-end))
  (assign exp (op car) (reg machine-args))
  (assign machine-args (op cdr) (reg machine-args))
  (assign env (op get-global-environment))
  (assign continue (label non-interactive-eval))
  (goto (label eval-dispatch))
```

Can be jumped to instead of `read-eval-print-loop`. It assumes there is a list of expressions to be evaluated in the `machine-args` register, and goes over them, evaluating them from left to right. This allows for convenient execution of the evaluator as follows:

```
(define code
  '(
    (define (sumsq a b)
      (+ (* a a) (* b b)))
    (define x 5)
    (define y 7)
    (define ans (sumsq x y))
    (print ans)
    ))

(set-register-contents! ec-eval 'machine-args code)
(start ec-eval)
=>
74
```

You'll also see in the beginning of the controller code these two lines:

```
  ; (goto (label read-eval-print-loop))
  (goto (label non-interactive-eval))
```

Currently the jump to `read-eval-print-loop` is commented, but this can be easily changed.

Another change you'll note in the definition of the machine's operations. I got tired of writing each operator name twice (once quoted, and once plain), and wrote the macro[1] `qq` to handle it for me:

```
(define-macro (qq e)
  `(list ',e ,e))
```

For example, `(qq adjoin-arg)` gets translated in "compile-time" into `(list 'adjoin-arg adjoin-arg)` so I have less repetitive typing to do.

Exercise 5.23

Adding support for `cond`:

```
ev-cond
  (assign exp (op cond->if) (reg exp))
  (goto (label ev-if))
```

And the appropriate dispatch is:

```
(test (op cond?) (reg exp))
(branch (label ev-cond))
```

Finally, to make it all work the operations `cond?` and `cond->if` must be added to the machine.

Exercise 5.24

Here's the code:

```
  ;; Implemented explicitly as a basic special form,
  ;; without converting to a nested if
  ;;
ev-cond-basic
  (assign unev (op cond-clauses) (reg exp))
ev-cond-ev-clause
  (assign exp (op first-exp) (reg unev))
  (test (op cond-else-clause?) (reg exp))
  (branch (label ev-cond-action))
  (save exp)
  (save env)
  (save unev)
  (save continue)
  ;; Setup an evaluation of the clause predicate
  (assign exp (op cond-predicate) (reg exp))
  (assign continue (label ev-cond-clause-decide))
  (goto (label eval-dispatch))

ev-cond-clause-decide
  (restore continue)
  (restore unev)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-cond-action))
ev-cond-next-clause
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-cond-ev-clause)) ; loop to next clause

  ;; We get here when the clause condition was found to
  ;; be true (or it was an 'else' clause), and we want
  ;; the actions to be evaluated. The clause is in exp.
  ;; We setup a call to ev-sequence and jump to it.
  ;;
ev-cond-action
  (assign unev (op cond-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

Now `eval-dispatch` can jump to `ev-cond-basic`:

```
  (test (op cond?) (reg exp))
  (branch (label ev-cond-basic))
```

Exercise 5.25

I'll pass.

Exercise 5.26

To make this work, I've re-included the modified `make-new-machine` and `make-stack` from exercise 5.14, and removed the command to reinitialize the stack after each executed expression in `non-interactive-eval`. The code I'm running is:

```
(define code
  '(
    (define (factorial n)
      (define (iter product counter)
        (if (> counter n)
            product
            (iter (* counter product)
                  (+ counter 1))))
      (iter 1 1))

    (newline)
    (print (factorial 3))
    (newline)
  ))

(set-register-contents! ec-eval 'machine-args code)
(start ec-eval)

(newline)
(newline)
```

a. The maximal depth required for this code is 13

b.

n	pushes
2	113
3	148
4	183
5	218

From this, the number of pushes is `35n + 43`.

Exercise 5.27

Type	Maximal depth	Number of pushes
Recursive	5n + 6	32n – 2
Iterative	13	35n + 43

Exercise 5.28

After changing `ev-sequence` to:

```
ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```

The table of exercise 5.27 turns into:

Type	Maximal depth	Number of pushes
Recursive	8n + 6	34n – 2
Iterative	3n + 17	37n + 47

Exercise 5.29

a.

n	depth
2	16
3	21
4	26
5	31

From this, the formula for depth is `5n + 6`.

b.

Let's collect some more data for this:

n	pushes
2	86
3	142
4	254
5	422
6	702
7	1150
8	1878

It can be easily seen that `S(n) = S(n-1) + S(n-2) + 26`.

To find out `a` and `b`, I'll subtract `S(n)` from `S(n+1)` to cancel out `b`:
`S(n+1)-S(n) = a*Fib(n+2) - a*Fib(n+1) = a*(Fib(n+2) - Fib(n+1))`. But due to the property of the Fibonacci series, this means that: `S(n+1)-S(n) = a*Fib(n)`. And since `S(n) = S(n-1) + S(n-2) + 26`, we can rewrite it as `a*Fib(n-1) = S(n-2) + 26`, or `S(n) = a*Fib(n+1) - 26`.

Now we're ready to assign the various results into this to compute `a`. Having the formula and `b` in our hands, it's easy to see that `a = 56`.

So, finally: `S(n) = 56Fib(n+1) - 26`

Exercise 5.30

a. Maybe I'm missing something, but this doesn't seem to be so much work. After all, a variable is only handled in a single place in the `eval` dispatch, so this is the place to make a modification. First of all, I'll change the supporting Scheme code to have special values for unbound variables:

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (cons 'bound (car vals)))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (cons 'unbound '())
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

(define (var-val-bound? varval)
  (and
    (pair? varval)
    (eq? (car varval) 'bound)))

(define (var-val-extract-value varval)
  (cdr varval))
```

Now, changing `ev-variable` in the evaluator code:

```
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (test (op var-val-bound?) (reg val))
  (branch (label ev-variable-var-bound))
  (goto (label unbound-variable))
ev-variable-var-bound
  (assign val (op var-val-extract-value) (reg val))
  (goto (reg continue))
```

And finally, a new error type:

```
unbound-variable
  (assign val (const unbound-variable-error))
  (goto (label signal-error))
```

b. Well, this indeed is a lot of work, and it's very repetitive, so I'm reluctant to do it. The basic model is identical to what I've written in part **a** of this question.

Each primitive that can signal an error must be modified similarly to the way `lookup-variable-value` was modified, to return a pair of `(error? value)`, with the appropriate support procedures. The evaluator code in `primitive-apply` must check for the error, similarly to the modified `ev-variable` in the code above.

---

[1] Personally, I like the `define-macro` utility more than the Scheme's `syntax-rules` and `syntax-case` macro tools. `define-macro` is almost identical to Common Lisp's `defmacro` and is natively supported by the most common Scheme implementations (Bigloo, Chicken, Gambit, Gauche and PLT Scheme). To use it in PLT Scheme you'll have to include the `defmacro.ss` library:

```
(require (lib "defmacro.ss"))
```

For comments, please send me ✉ an email.

↑ Back to top