SICP section 5.2

February 23, 2008 at 18:04 Tags SICP

The full implementation of the register-machine simulator is here.

Exercise 5.7

Done

Exercise 5.8

When control reaches `there`, the contents of `a` are 3. This happens because:

- When the machine code is assembled, the function `extract-labels` builds a list of all the labels in the code. When two labels have the same name, this label name will appear twice in the list.
- When the `goto` instruction is executed, it finds the label to go to by calling `lookup-label`, which uses `assoc` on the list of labels. `assoc` returns the first "hit", so the fist label of the duplicates is jumped to.

We'll modify `extract-labels` to test whether a label already exists before adding it to the list:

```
(define (extract-labels text)
  (if (null? text)
    (cons '() '())
    (let ((result (extract-labels (cdr text))))
      (let ((instructions (car result))
            (labels (cdr result)))
        (let ((next-instruction (car text)))
          (if (symbol? next-instruction) ; a label ?
            (if (label-exists labels next-instruction)
              (error "Label name is duplicated: " next-instruction)
              (cons instructions
                    (cons (make-label-entry next-instruction instructions) labels)))
            (cons (cons (make-instruction next-instruction) instructions)
                  labels)))))))
```

This auxiliary function is used to preserve the abstraction of `labels`:

```
(define (label-exists labels label-name)
  (assoc label-name labels))
```

Exercise 5.9

Adding a simple label test to `make-operation-exp` does the trick:

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
        (aprocs
         (map (lambda (e)
                (if (label-exp? e)
                  (error "Using operation on label: " e)
                  (make-primitive-exp e machine labels)))
              (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

Exercise 5.10

At this point in the book, I don't find this exercise interesting enough to pursue.

Exercise 5.11

a.

The change will be done after the label `afterfib-n-2`. These lines:

```
(assign n (reg val))
(restore val)
```

Place `Fib(n-2)` in `n` and `Fib(n-1)` in `val`. They can be replaced by the single line:

```
(restore n)
```

Which places `Fib(n-1)` in `n`, because `Fib(n-2)` is already in `val`, and we only use the values in an addition which is commutative, so it doesn't care about the order of its addends:

```
(assign val (op +) (reg val) (reg n))
```

b.

I'll store a `(reg-name reg)` pair on the stack instead of just the `reg`. These are the new `save` and `restore`:

```
(define (make-save inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (push stack (cons reg-name (get-contents reg)))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (let* ((stack-top (pop stack))
             (saved-reg-name (car stack-top))
             (saved-reg (cdr stack-top)))
        (if (equal? reg-name saved-reg-name)
          (begin
            (set-contents! reg saved-reg)
            (advance-pc pc))
          (error (format "Restoring saved reg ~a into ~a~%"
                         saved-reg-name reg-name)))))))
```

c.

First of all, I'll modify the stack data structure to make it manage several stacks, each with its own name. Now the state variable `s` holds an association list of `(stack-name stack)` which is accessed with `assoc`. `push` and `pop` will receive the stack name:

```
(define (make-stack)
  (let ((s '()))
    (define (push reg-name x)
      (let ((reg-stack (assoc reg-name s)))
        (if reg-stack
          (set-cdr! reg-stack (cons x (cdr reg-stack)))
          (error "PUSH: No stack for register " reg-name))))
    (define (pop reg-name)
      (let ((reg-stack (assoc reg-name s)))
        (if reg-stack
          (if (null? (cdr reg-stack))
            (error "POP: Empty stack for register " reg-name)
            (let ((top (cadr reg-stack)))
              (set-cdr! reg-stack (cddr reg-stack))
              top))
          (error "POP: No stack for register " reg-name))))
    (define (add-reg-stack reg-name)
      (if (assoc reg-name s)
        (error "Stack already exists for " reg-name)
        (set! s (cons (cons reg-name '()) s))))
    (define (initialize)
      (for-each
        (lambda (stack)
          (set-cdr! stack '()))
        s)
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) pop)
            ((eq? message 'add-reg-stack) add-reg-stack)
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request -- STACK" message))))
    dispatch))

(define (pop stack reg-name)
  ((stack 'pop) reg-name))

(define (push stack reg-name value)
  ((stack 'push) reg-name value))
```

Now, the `allocate-register` internal function in `make-new-machine` must be rewritten. Each time the machine is asked to allocate a new register, it adds a stack for this register to the stack management object:

```
(define (allocate-register name)
  (if (assoc name register-table)
    (error "Multiply defined register: " name)
    (begin
      (set! register-table
            (cons (list name (make-register name))
                  register-table))
      ((stack 'add-reg-stack) name)
      'register-allocated)))
```

And finally, these are the new `make-save` and `make-restore`:

```
(define (make-save inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
          (reg (get-register machine reg-name)))
    (lambda ()
      (push stack reg-name (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
          (reg (get-register machine reg-name)))
    (lambda ()
      (set-contents! reg (pop stack reg-name))
      (advance-pc pc))))
```

Here's a demonstration of this feature:

```
(define fib
  (make-machine
    '(n val n1)
    `((= ,=))
    '(
      (save n)
      (assign n (const 40))
      (save n)

      (save val)
      (assign val (const 10))
      (save val)

      (restore n)
      (assign n1 (reg n))
      (restore n)

      (assign val (const 1))
      (restore val)

      )))

(set-register-contents! fib 'n 8 )
(set-register-contents! fib 'val 3)
(start fib)
(printf ":~a~%" (get-register-contents fib 'n))
(printf ":~a~%" (get-register-contents fib 'n1))
(printf ":~a~%" (get-register-contents fib 'val))
=>
:8
:40
:10
```

Note how each register has its own stack, and a `save` or `restore` to another register don't affect it.

Exercises 5.12 – 5.13

I'll pass. I understand the simulator well enough now and don't feel these exercises will add to my comprehension.

Exercise 5.14

The factorial machine does all its `save`-s first and only then its `restore`-s. Hence, the maximal depth of the stack and the amount of `push`-es are equal. Let's try to match a linear equation for `P` – the amount of pushes:

n	P
1	0
2	2
3	4
4	6

5	8
6	10

From this it's quite obvious that for an input `n` the amount of pushes is `2n-2`.

Exercise 5.15

This is the modified `make-new-machine`. Changed and added lines are marked with a comment.

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (instruction-count 0)           ;; **
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
          (error "Multiply defined register: " name)
          (set! register-table
                (cons (list name (make-register name))
                      register-table)))
        'register-allocated)
      (define (lookup-register name)
        (let ((val (assoc name register-table)))
          (if val
              (cadr val)
              (error "Unknown register: " name))))
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
            'done
            (begin
              (set! instruction-count (+ 1 instruction-count)) ;; **
              ((instruction-execution-proc (car insts)))
              (execute)))))
      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc the-instruction-sequence)
               (execute))
              ((eq? message 'install-instruction-sequence)
               (lambda (seq)
                (set! the-instruction-sequence seq)))
              ((eq? message 'allocate-register) allocate-register)
              ((eq? message 'get-register) lookup-register)
              ((eq? message 'install-operations)
               (lambda (ops) (set! the-ops (append the-ops ops))))
              ((eq? message 'stack) stack)
              ((eq? message 'operations) the-ops)
              ((eq? message 'get-instruction-count)  ;; **
                (let ((count instruction-count))
                  (set! instruction-count 0)
                  count))
              (else (error "Unknown request -- MACHINE" message))))
      dispatch)))
```

The instruction count is incremented in the `execute` procedure, prior to executing a new instruction.

Exercise 5.16

I'll add a state variable into the large `let` at the top of `make-new-machine`:

```
(instruction-trace-on #f)
```

The new messages the machine accepts are:

```
((eq? message 'trace-on)
  (set! instruction-trace-on #t))
((eq? message 'trace-off)
  (set! instruction-trace-on #f))
```

And this is the new `execute` procedure (it augments both instruction counting and tracing):

```
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
      'done
      (begin
        (set! instruction-count (+ 1 instruction-count))
        (if instruction-trace-on
          (printf "trace: ~a~%" (instruction-text (car insts))))
        ((instruction-execution-proc (car insts)))
        (execute)))))
```

Exercise 5.17

To make this work I'll change the way instructions are represented. Thankfully, the "instruction" abstraction is hidden behind a constructor and a set of accessors, so not too much code has to be modified:

```
(define (make-instruction text)
  (list text '() '()))
(define (make-instruction-with-label text label)
  (list text label '()))
(define (instruction-text instruction)
  (car instruction))
(define (instruction-label instruction)
  (cadr instruction))
(define (instruction-execution-proc instruction)
  (caddr instruction))
(define (set-instruction-label! instruction label)
  "Sets the label that is tied to this instruction"
  (set-cdr! instruction proc))
(define (set-instruction-execution-proc! instruction proc)
  (set-car! (cddr instruction) proc))
```

As you can see, an instruction is now implemented by a triplet: the instruction text, the label tied to it and its execution procedure.

Now, all that's left is to modify `extract-labels` to attach labels to relevant instructions:

```
(define (extract-labels text)
  (if (null? text)
    (cons '() '())
    (let ((result (extract-labels (cdr text))))
      (let ((instructions (car result))
            (labels (cdr result)))
        (let ((next-instruction (car text)))
          (if (symbol? next-instruction) ; a label ?
            (if (label-exists labels next-instruction)
              (error "Label name is duplicated: " next-instruction)
              (cons
                (if (null? instructions)
                  '()
                  (cons
                    (make-instruction-with-label
                      (instruction-text (car instructions))
                      next-instruction)
                    (cdr instructions)))
                (cons (make-label-entry next-instruction instructions) labels)))
            (cons (cons (make-instruction next-instruction) instructions)
                  labels)))))))
```

Here's a sample run:

```
(define fib
  (make-machine
    '(n val n1)
    `((= ,=))
    '(
      (save n)
      (assign n (const 40))
      (save n)
    george
      (save val)
      (assign val (const 10))
    just-a-label
      (save val)
      )))

(set-register-contents! fib 'n 8)
(fib 'trace-on)
(start fib)

=>

trace: (save n)
trace: (assign n (const 40))
trace: (save n)
at label: george
trace: (save val)
trace: (assign val (const 10))
at label: just-a-label
trace: (save val)
```

Exercise 5.18

Here is the modified `make-register`. I've added the required messages, and changed the `cond` to a friendlier[1] `case`:

```
(define (make-register name)
  (let ((contents '*unassigned*)
        (trace-on #f))
    (define (dispatch message)
      (case message
        ((get) contents)
        ((set)
          (lambda (value)
            (when trace-on
              (printf "reg trace: ~a <- ~a (was ~a)~%"
                name value contents))
            (set! contents value)))
        ((trace-on) (set! trace-on #t))
        ((trace-off) (set! trace-off #f))
        (else ((error "Unknown request -- REGISTER" message)))))
    dispatch))
```

The following procedure and messages were added to `make-new-machine`:

```
  ...
  (define (set-register-trace! name trace-msg)
    (let ((reg (assoc name register-table)))
      (if reg
        ((cadr reg) trace-msg)
        (error "Unknown register: " name))))
  ...
  ((eq? message 'reg-trace-on)
    (lambda (reg-name)
      (set-register-trace! reg-name 'trace-on)))
  ((eq? message 'reg-trace-off)
    (lambda (reg-name)
      (set-register-trace! reg-name 'trace-off)))
```

Here's a sample, with both instruction and register tracing on:

```
(define fib
  (make-machine
    '(n val n1)
    `((= ,=))
    '(
      (assign n (const 40))
      (save n)
    george
      (save val)
      (assign val (const 10))
      (assign n (const 125))
      )))

=>

trace: (assign n (const 40))
reg trace: n <- 40 (was 8)
trace: (save n)
at label: george
trace: (save val)
trace: (assign val (const 10))
trace: (assign n (const 125))
reg trace: n <- 125 (was 40)
```

Exercise 5.19

I'll pass.

[1] I deem `case` to be friendlier here because all the `cond` clauses are comparing the same variable `message`) to

different values. Using `case` here saves quite a few keystrokes.

For comments, please send me an email.

Back to top