CS 565 September 3, 1997
Jens Palsberg Readings: EOPL, Chapter 4.

Lecture Note 3:

A mostly Object-Oriented Programming Style in Scheme

Contents

1 Sequencing and Side Effects in Scheme **2**

2 An Object-Oriented Programming Style **7**

2.1 Sharing of bindings 8

2.2 Modifying a binding local to a procedure 9

2.3 Classes and objects 10

3 Implementing Recursion by Assignments **11**

4 Lambda-calculus and Beta-reduction **12**

1 Sequencing and Side Effects in Scheme

(`begin` exp_1 ... exp_n)

is handy if one of more of $exp_1 \ldots exp_n$ have side effects. It is a special form where $exp_1 \ldots exp_n$ are evaluated in order, and the value of exp_n is the value of the entire `begin` expression.

```
(define for-each
  (lambda (proc lst)
    (if (null? lst)
        'done
        (begin
          (proc (car lst))
          (for-each proc (cdr lst))))))
```

The bodies of `lambda`, `let`, `letrec`, and the consequents of `cond` are implicitly contained in a `begin` expression.

```
(define displayln
   (lambda lst
     (for-each display lst)
     (newline)))
```

Example: factorial

```
> (define fact
     (lambda (n)
       (displayln "Entering fact with n = " n)
       (let ((ans (if (zero? n)
                      1
                      (* n (fact (- n 1))))))
         (displayln "Returning from fact with " ans)
 ans)))
> (fact 3)
Entering fact with n = 3
Entering fact with n = 2
Entering fact with n = 1
Entering fact with n = 0
Returning from fact with 1
Returning from fact with 1
Returning from fact with 2
Returning from fact with 6
6
```

Data Structure Mutation

The standard procedures `set-car!` and `set-cdr!` change the car and cdr fields of pairs!

```
> (define lst (list 1 2))
> (set-car! lst 3)
> lst
(3 2)
> (set-cdr! lst (list 4 5))
> lst
(3 4 5)
> (set-cdr! lst 6)
> lst
(3 . 6)
```

Data Structure Mutation

Here is a procedure `reverse!` which destroys the original list.

```
> (define reverse!
    (letrec ((loop
              (lambda (last ls)
                (let ((next (cdr ls)))
                  (set-cdr! ls last)
                  (if (null? next)
                      ls
                      (loop ls next))))))
      (lambda (ls)
        (if (null? ls)
            ls
            (loop '() ls)))))
> (define ls (list 1 2 3))
> (reverse! ls)
(3 2 1)
> ls
(1)
```

Variable Assignment

Think of variables as being bound to *locations*.

Then a variable reference refers to the value contained in such a location.

The special form

`(set!` *var exp*`)`

evaluates the expression *exp*, places its value in the location that is the binding of variable *var*, and returns an unspecified value.

```
> (define x 1)
> (set! x 2)
> x
2
> (let ((y 3))
    (set! y 4)
    (+ y 1))
5
```

2 An Object-Oriented Programming Style

The style is based on three ideas:

- Sharing of bindings.
- Procedures stored in variables.
- A let expression which returns a procedure.

2.1 Sharing of bindings

```
> (define empty? '*)
> (define push! '*)
> (define pop! '*)
> (define top '*)
> (let ((stk '()))
    (set! empty?  (lambda ()  (null? stk)))
    (set! push!   (lambda (x) (set! stk (cons x stk))))
    (set! pop!    (lambda ()  (if (empty?)
                                  (error "Stack empty")
                                  (set! stk (cdr stk)))))
    (set! top     (lambda ()  (if (empty?)
                                  (error "Stack empty")
                                  (car stk)))))

> (push! 1)
> (push! 2)
> (top)
2
> (pop!)
> (top)
1
> (pop!)
```

The four procedures `empty?`, `push!`, `pop!`, and `top` share the same `stk` binding. Thus, the stack is a data abstraction.

2.2 Modifying a binding local to a procedure

```
(define stack
  (let ((stk '()))
    (lambda (message)
      (case message
        ((empty?) (lambda ()  (null? stk)))
        ((push!)  (lambda (x) (set! stk (cons x stk))))
        ((pop!)   (lambda ()  (if (null? stk)
                                  (error "Stack empty")
                                  (set! stk (cdr stk)))))
        ((top)    (lambda ()  (if (null? stk)
                                  (error "Stack empty")
                                  (car stk))))
        (else (error
                (format "stack: does not understand : ~s"
                        message)))))))
```

```
> ((stack 'push!) 1)
> ((stack 'push!) 2)
> ((stack 'top))
2
> ((stack 'pop!))
> ((stack 'top))
1
> ((stack 'pop!))
```

Note that the `lambda` is *inside* the `let`.

This `stack` is much like an object in object-oriented programming.

2.3 Classes and objects

The following procedure generate stacks.

```
(define make-stack
  (lambda ()
    (let ((stk '()))
      (lambda (message)
        (case message
          ((empty?) (lambda ()  (null? stk)))
          ((push!)  (lambda (x) (set! stk (cons x stk))))
          ((pop!)   (lambda ()  (if (null? stk)
                                    (error "Stack empty")
                                    (set! stk (cdr stk)))))
          ((top)    (lambda ()  (if (null? stk)
                                    (error "Stack empty")
                                    (car stk))))
          (else (error
                  (format "stack: does not understand : "
                          message)))))))))
```

```
> (define s1 (make-stack))
> (define s2 (make-stack))
> ((s1 'push!) 1)
> ((s2 'push!) 2)
> ((s1 'top))
1
> ((s2 'top))
2
```

Each time `make-stack` is invoked, a new stack procedure, with its own `stk` binding, is returned. This gives a simple form of classes, as known from object-oriented programming.

3 Implementing Recursion by Assignments

Here is how we can transform programs with `letrec` into program without `letrec`, but with `set!`.

(`letrec` ((*var*$_1$ *exp*$_1$) ...(*var*$_n$ *exp*$_n$))
  body)

⇒

(`let` ((*var*$_1$ `'*`) ...(*var*$_n$ `'*`))
  (`set!` *var*$_1$ *exp*$_1$)
  ...
  (`set!` *var*$_n$ *exp*$_n$)
  body)

The let expression first introduces variable bindings that initially contain some irrelevant value (here: `'*`). The expressions *exp*$_1$ ... *exp*$_n$ are then evaluated *in the scope* of these bindings, and their values are assigned to the bindings. Finally, *body* is evaluated in the scope of the bindings.

Note that this implementation fixes the order in which *exp*$_1$ ... *exp*$_n$ are evaluated.

4 Lambda-calculus and Beta-reduction

How do we determine the result of a procedure call?

```
> (define foo
    (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
22
```

Let's calculate:

```
     (foo
       (+ 4 1) 7)

=>   ((lambda (x y) (+ (* x 3) y))
       (+ 4 1) 7)

=>   (+ (* (+ 4 1) 3) 7)

=>   22
```

We could also calculate like this

```
     (foo
       (+ 4 1) 7)

=>   (foo
       5 7)

=>   ((lambda (x y) (+ (* x 3) y))
       5 7)

=>   (+ (* 5 3) 7)

=>   22
```

A subset of Scheme

Let us look at a tiny subset of Scheme.

Syntax:

⟨exp⟩ ::= ⟨varref⟩
| (lambda (⟨var⟩) ⟨exp⟩)
| (⟨exp⟩ ⟨exp⟩)

The only values that can be computed are *procedures*. No numbers, no booleans, only procedures!

The only computation that can take place is *procedure application*. No plus, no if, only procedure application!

This language is called The Lambda Calculus. The traditional syntax for procedures in the lambda calculus uses the Greek letter λ (lambda), replacing the second alternative in the above grammar with

λ ⟨var⟩ . ⟨exp⟩

Free and bound variable references

A variable x *occurs free* in an expression E if and only if

- E is a variable reference and E is the same as x; or
- E is of the form `(`$E_1 E_2$`)` and x occurs free in E_1 or E_2; or
- E is of the form `(lambda (`y`)` E'`)`, where y is different from x and x occurs free in E'.

A variable x *occurs bound* in an expression E if and only if

- E is of the form `(`$E_1 E_2$`)` and x occurs bound in E_1 or E_2; or
- E is of the form `(lambda (`y`)` E'`)`, where x and y are different variables and x occurs bound in E', *or* x and y are the same variable and y occurs free in E'.

Example: the variable references `x` and `y` occur bound in the expression

```
(lambda (y) ((lambda (x) x) y))
```

Another example: the variable reference `x` occurs bound, and the variable reference `y` occurs free in the expression

```
((lambda (x) x) y)
```

Procedure application

Now consider an application:

`(` *rator rand* `)`

To actually perform a procedure application, we of course have to evaluate the operator (*rator*) to a procedure. This gives the following situation:

`(` `(lambda (`*var*`)` *body* `)` *rand* `)`

The basic Scheme rule for this situation is:

The operand (*rand*) is evaluated to a value *before* the procedure call takes place.

Example:

```
    ((lambda (x) x)
     ((lambda (y) (+ y 9)) 5))

=>  ((lambda (x) x) (+ 5 9))

=>  ((lambda (x) x) 14)

=>  14
```

This is called *applicative-order* reduction.

The slogan is: "always evaluate the arguments first".

This is the evaluation principle for Scheme and ML.

An Alternative Rule

Consider again the situation:

`( (lambda (`*var*`) `*body*` ) `*rand*` )`

Let us now change the Semantics completely!

Here is a new evaluation principle:

The procedure call takes place without evaluating the operand *rand* **at all.**

Example:

```
    ((lambda (x) x)
     ((lambda (y) (+ y 9) 5))

=>  ((lambda (y) (+ y 9)) 5)

=>  (+ 5 9)

=>  14
```

This is called *leftmost* reduction.

The slogan is: "don't work if you can avoid it; be lazy!".

This is the evaluation principle for Miranda and Haskell.

Does it make a Difference?

Now we have one language with two different evaluation principles. That is: two semantics!

What if we take a program and first run it with applicative-order reduction (just run it in Scheme) and then run it again but this time with leftmost reduction (have to build a new system for this case ...).

Is there at least one program where the two runs will give different results?

Answer: the following is true of all programs:

- If the run with applicative-order reduction terminates, then the run with leftmost reduction terminates. (But the converse is in general false).
- If both runs terminate, then they give the same result.

A case where applicative-order reduction is too eager

Sometimes applicative-order reduction fails to terminate, even though leftmost reduction terminates.

Notice that:

```
  (define delta (lambda (x) (x x)))

    (delta delta)

=>  (delta delta)

=>  (delta delta)

=>  ...
```

Consider the program:

```
  ((lambda (y) 3) (delta delta))
```

Applicative-order reduction fails to terminate because it continues forever with trying to evaluate the operand.

Leftmost reduction terminates, simply because it throws away the argument:

```
    ((lambda (y) 3) (delta delta))

=>  3
```

Applicative-order reduction is often the most efficient

Consider this leftmost reduction:

```
    ((lambda (x) (x (x y)))
     ((lambda (w) w) z))

=>  (((lambda (w) w) z) (((lambda (w) w) z) y)))

=>  (z (((lambda (w) w) z) y)))

=>  (z (z y))
```

Let us then run the program with applicative order reduction:

```
    ((lambda (x) (x (x y)))
     ((lambda (w) w) z))

=>  ((lambda (x) (x (x y)))
     z)

=>  (z (z y))
```

The second computation is one step shorter!

Why? Because in the first computation, there were taken two copies of the argument before the procedure application `((lambda (w) w) z)` took place. That gave an extra step.

Beta-Redex

Notice that both applicative-order and leftmost reduction are *top-down*.

Both look for procedure applications that are *outside* `lambda`, that is, at the top-most level of the program.

They choose different applications, of course.

What is it they look for?

They look for patterns of the form:

`(` `(lambda (`*var*`)` *body* `)` *rand* `)`

Such a pattern is called a *redex*.

We can generalize applicative-order reduction and leftmost reduction by allowing the next redex chosen to be *any* redex, also one in the body of a `lambda`.

The process of doing a "procedure call" for some redex is called *beta-reduction*.

The idea is:

`(` `(lambda (`*var*`)` *body* `)` *rand* `)`

$\Rightarrow$ *body*`[`*rand*`/`*var*`]`

Here, *body*`[`*rand*`/`*var*`]` intuitively means "the result is like *body* except that all occurrences of *var* is replaced by *rand*".

When we allow arbitrary beta-version to take place, then the tiny subset of Scheme is called the *lambda calculus*.

Name clashes

Care must be taken to avoid name clashes.

Example:

```
((lambda (x)
   (lambda (y) (x y)))
 (y w))
```

should *not* be transformed into

```
(lambda (y) ((y w) y))
```

The reference to `y` in `(y w)` should remain free!

The solution is to change the name of the inner variable name `y` to some name, say `z`, that does not occur free in the argument `y w`.

```
((lambda (x)
   (lambda (z) (x z)))
 (y w))
```

Now we get:

```
(lambda (z) ((y w) z))
```

Beta-reduction

We need to define what $E[M/\texttt{x}]$ means. We do this inductively on the form of E.

Variables

There are two subcases:

$\texttt{x}[M/\texttt{x}] = M$ Yes, substitute!

$\texttt{y}[M/\texttt{x}] = \texttt{y}$ where $\texttt{x} \neq \texttt{y}$ No substitution here!

Applications

Do the substitution on both operator and operand:

$(F\ G)[M/\texttt{x}] = (F[M/\texttt{x}]\ G[M/\texttt{x}])$

Just propagate the work!

Procedures

There are two subcases:

`(lambda (x)` E`)`$[M/\texttt{x}]$ = `(lambda (x)` E`)`

No further work needed!

`(lambda (y)` E`)`$[M/\texttt{x}]$ = `(lambda (z)` $(E[\texttt{z}/\texttt{y}])[M/\texttt{x}]$ `)`

where $\texttt{x} \neq \texttt{y}$ and z does not occur free in E or M.

The second subcase creates a lot of new variables. We can save work by observing that if x does not occur free in E, then there is nothing to do. Moreover, if y does not occur free in M, then we need not change from y to z.

Alpha-Conversion

We can systematically change the names of bound variables:

`(lambda (`*var*`)` *exp*`) = (lambda (`*newvar*`)` *exp*`[`*newvar*`/`*var*`])`

Such renaming is called alpha-conversion.

Example of beta-reduction

```
(define i
 ((lambda (k1)
    ((lambda (k4) (k4 (lambda (x) (lambda (k5) (k5 x)))))
     (lambda (v1)
       ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
        (lambda (v2) ((v1 v2) k1)))
     )))
  (lambda (a) a)))
```

Let us calculate `(i 2)`.

```
(i 2)
=>
(((lambda (k4) (k4 (lambda (x) (lambda (k5) (k5 x)))))
   (lambda (v1)
     ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((v1 v2) (lambda (a) a))))
   ))
  2)
=>
(((lambda (v1)
     ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((v1 v2) (lambda (a) a))))
   )
     (lambda (x) (lambda (k5) (k5 x))))
  2)
```

```
=>
(((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((
                (lambda (x) (lambda (k5) (k5 x)))
                        v2) (lambda (a) a))))
  2)
=>
(((lambda (v2) ((
                (lambda (x) (lambda (k5) (k5 x)))
                        v2) (lambda (a) a)))
                    (lambda (y) (lambda (k3) (k3 y))))
  2)
=>
((((lambda (x) (lambda (k5) (k5 x)))
                        (lambda (y) (lambda (k3) (k3 y)))
                          ) (lambda (a) a))
  2)
=>
(((lambda (k5) (k5 (lambda (y) (lambda (k3) (k3 y)))))
                             (lambda (a) a))
  2)
=>
(((lambda (a) a) (lambda (y) (lambda (k3) (k3 y))))
  2)
=>
((lambda (y) (lambda (k3) (k3 y)))
  2)
=>
(lambda (k3) (k3 2))
```