COMP 210, Spring 2001
Lecture 22: Coping with mutation operators

Reminders

Review

We introduced the other mutation operators of Scheme: **`set!`** and the various **`set-struct!`** operators and discussed how mutation operators break our existing evaluation rules. Today, we will fix those rules.

We also introduced the **`begin`** construct which evaluates a sequence of expressions and returns the last one.

Digression

We can use **`begin`** insert "print'' statements in programs to help us debug them. We can observe the arguments passed to a function (or the value of any other expression) in this way. The name of the print operation in Scheme is **`printf`**. It has the following contract, header, and purpose:

```
; printf: string any … any -> void

; (printf format-string v1 … vn) prints formatted output to the
; screen where format-string is a string that is printed;
; format-string can contain special formatting tags:
; ~n or ~% prints a newline
; ~v or ~V prints the next argument among the vs
; ~~ prints a tilde (~)
; The number of special formatting tags should equal the number n
; of values v1 … vn to be printed
```

More on `begin` and `set!`

The **`begin`** construct is only useful if we have expressions with effects other than simply returning a value. In fact, we reserve the term *effect* to describe a mutation to the program state. Thus, begin is useful with **`set!`** and other mutation operators precisely because **`set!`** changes the value of some variable and other mutators modify some data structure).

Consider the following trivial example:

```
(define n 5)
(begin
  (set! n (add1 n))
     n)
```

If we evaluate this, the define expression creates an object named **`n`** and gives it the value five. Evaluating the begin expression first evaluates the **`set!`**, which sets **`n`**'s value to **`6`**, and then evaluates **`n`**, which returns **`6`**.

What happens with this piece of code:

```
(define x 3)
(define y 4)
(begin
  (set! x y)
  (set! y x))
```

The **`define`**s create objects **`x`** and **`y`** that have the values **`3`** and **`4`**, respectively. The **`begin`** expression looks like it should swap those values. However, that's not what happens.

<hand evaluate the expression>

Contrary to the intuitions that we've developed over the past ten weeks, this does not interchange the values of **`x`** and **`y`**. The first **`set!`** changes **`x`**'s value by replacing it with **`y`**'s value. The second **`set!`** takes the value of **`x`** (which is now identical to the value of **`y`**) and uses it to replace the value of **`y`**. Thus, after executing the **`begin`**, **`x`** has the value of 4, as does (surprise!) **`y`**. The net effect is the same as if we had never executed the second **`set!`**.

Can we write a program **`swap: number number → void`** that takes two numbers and swaps their values? This should take less than two minutes.

```
; swap: number number → void
;   interchanges the values of the arguments
(define (swap x y)
  (local [(define temp x)]
    (begin
      (set! x y)
      (set! y temp))))
```

This program uses a variable temp to preserve the value of **`x`** while it overwrites **`x`** with **`y`**'s value. Then, it takes the preserved value (**`x`**'s old value) and assigns it to **`y`**. Because the expressions inside the begin execute in sequential order rather than concurrently (at the same time, or in parallel), we need an extra place to hide one of the values.

But does this work? No, it does not work. If we execute it using DrScheme, we get the following behavior:

```
(define a 5)
(define b 6)
(swap a b)
a
> 5
```

```
        b
        > 6
```

Why? Remember our rewriting rules. When we rewrite **`(swap a b)`**, what happens? The rewriting engine replaces any occurrences of **`a`** and **`b`** in the body of **`swap`** with their respective values. Does this mean that the **`set!`** expressions inside the body actually change the values of the constants? Of course not. What variables does **`swap`** modify? **`x`** and **`y.`**

Are these variables synonyms for **`a`** and **`b`**? No. They are local to the body of **`swap.`**

Evaluating set!

We can easily extend our evaluation rules to handle the **`set!`** construct. Let ***x*** be a variable name and ***v*** and ***newv*** be values. If the form

(set! *x newv*)

is the leftmost reducible form then

…

(define *x v*)

…

(set! *x newv*)

…

=>

…

(define *x newv*)

…

(void)

…

Note: **`(void)`** is a value that the DrScheme interactions window does not print when it appears at top-level (not embedded inside another value).

Devising evaluation rules to handle data mutation operations like **`vector-set!`** is more difficult. To gain insight, let's see if we can simulate mutable structures in the Scheme dialect for which we have evaluation rules.

Understanding set-structure!

Assume that Scheme has no structures. Can we implement them using the other constructs of Scheme? Consider the following implementation of pairs.

```
; (define-struct Pair (left right))
(define (make-Pair x y)
  (local [(define left x)
          (define right y)]
    (lambda (msg)
      (cond [(symbol=? msg 'left) left]
            [(symbol=? msg 'right) right]
            [(symbol=? msg 'set-left)
             (lambda (val) (set! left val))]
            [(symbol=? msg 'set-right)
             (lambda (val) (set! right val))]
            [else (error 'Pair "illegal operation")]))))
(define (Pair-left p) (p 'left))
(define (Pair-right p) (p 'right))
(define (set-Pair-left! p v) ((p 'set-left) v))
(define (set-Pair-right! p v) ((p 'set-right) v))
```

We can represent structures as procedures that take symbols (messages) as arguments specifying what operation to perform. Such procedural representations are often called "object-oriented" representations because they are similar to the "objects" of object-oriented programming. Note that this representation correctly handles the sharing relationships between variables bound to structures.

Question: what happens when a variable **q** is defined as the value of a variable **p** bound to a simulated structure.

Evaluation Rules for Scheme with Data Mutation

To represent mutable structures, we need values that handle sharing like **lambda** does. Copying a **lambda** form does not change the variables that it mentions; hence all copies are indistinguishable. In contrast, copying a **(make-struct …)** expression creates a distinct structure. Hence we cannot copy **(make-struct …)** expressions. We can avoid such copying by introducing a new form of value called a *location.*

In our revised evaluation system, **(make-struct …)** expressions evaluate to *locations* which are special variables that are values. Each location variable is bound to the corresponding **(make-struct …)** expression in a **(define …)** form. To distinguish location variables from other variables, we will include the character ”$” in the name of each location variable and exclude this character from ordinary variable names. A **`set-structure!`** operation takes a location as it first argument and modifies the specified field of the structure corresponding to that location.

Let's work an example.

```
   (define-struct Pair (left right))
   (define p (make-Pair 3 4))
   (define q p)
   (set-Pair-left! p 5)
   (Pair-left q)
=  (define-struct Pair …)
   (define Pair$1 (make-Pair 3 4))
   (define p Pair$1)
   (define q p)
   (set-Pair-left! p 5)
   (Pair-left q)
=  (define-struct Pair …)
   (define Pair$1 (make-Pair 3 4))
   (define p Pair$1)
   (define q Pair$1)
   (set-Pair-left! p 5)
   (Pair-left q)
   (define-struct Pair …)
   (define Pair$1 (make-Pair 3 4))
   (define p Pair$1)
   (define q Pair$1)
   (set-Pair-left! Pair$1 5)
   (Pair-left q)
=  (define-struct Pair …)
   (define Pair$1 (make-Pair 5 4))
   (define p Pair$1)
   (define q Pair$1)
   (void)
   (Pair-left q)
=  (define-struct Pair …)
   (define Pair$1 (make-Pair 5 4))
   (define p Pair$1)
```

```
    (define q Pair$1)
    (void)
    (Pair-left Pair$1)
=   (define-struct Pair …)
    (define Pair$1 (make-Pair 5 4))
    (define p Pair$1)
    (define q Pair$1)
    (void)
    5
```