CSE341: Programming Languages

Lecture 15
Mutation, Pairs, Thunks, Laziness, Streams, Memoization

Dan Grossman

Fall 2011

Today

Primary focus: Powerful programming idioms related to:

- Delaying evaluation (using functions)
- Remembering previous results (using mutation)

Lazy evaluation, Streams, Memoization

But first need to discuss:

- Mutation in Racket
- The truth about cons cells (they're just pairs)
- mcons cells (mutable pairs)

Set!

- Unlike ML, Racket really has assignment statements
  - But used *only-when-really-appropriate!*

```
(set! x e)
```

- For the **`x`** in the current environment, subsequent lookups of **`x`** get the result of evaluating expression **`e`**
  - Any code using this **`x`** will be affected
  - Like Java's **`x = e`**

- Once you have side-effects, sequences are useful:

```
(begin e1 e2 … en)
```

Example

Example uses **`set!`** at top-level; mutating local variables is similar

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4)) ; 7
(set! b 5)
(define z (f 4))   ; 9
(define w c)       ; 7
```

Not much new here:

- Environment for closure determined when function is defined, but body is evaluated when function is called
- Once an expression produces a value, it is irrelevant how the value was produced

Top-level

- Mutating top-level definitions is particularly problematic
  - What if any code could do **`set!`** on anything?
  - How could we defend against this?

- A general principle: If something you need not to change might change, make a local copy of it. Example:

```
(define b 3)
(define f
  (let ([b b])
    (lambda (x) (* 1 (+ x b)))))
```

Could use a different name for local copy but do not need to

But wait…

- Simple elegant language design:
  - Primitives like **+** and ***** are just predefined variables bound to functions
  - But maybe that means they are mutable
  - Example continued:

```
(define f
  (let ([b b]
        [+ +]
        [* +])
    (lambda (x) (* 1 (+ x b)))))
```

  - Even that won't work if **f** uses other functions that use things that might get mutated – all functions would need to copy everything mutable they used

No such madness

In Racket, *you do not have to program like this*

- Each file is a module
- *If* a module does not use **`set!`** on a top-level variable, then Racket makes it constant and forbids **`set!`** outside the module
- Primitives like **`+`**, **`*`**, and **`cons`** are in a module that does not mutate them

In Scheme, you really could do **`(set! + cons)`**

- Naturally, nobody defended against this in practice so it would just break the program

Showed you this for the *concept* of copying to defend against mutation

The truth about **cons**

`cons` just makes a pair

- By convention and standard library, lists are nested pairs that eventually end with **`null`**

```
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define hi (cdr (cdr pr)))
(define no (list? pr))
(define yes (pair? pr))
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi2 (car (cdr (cdr pr))))
```

Passing an improper list to functions like **`length`** is a run-time error

So why allow improper lists?

- Pairs are useful
- Without static types, why distinguish **`(e1,e2)`** and **`e1::e2`**

cons cells are immutable

What if you wanted to mutate the *contents* of a cons cell?

- In Racket you can't (major change from Scheme)
- This is good
  - List-aliasing irrelevant
  - Implementation can make a fast **`list?`** since listness is determined when cons cell is created

This does *not* mutate the contents:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

- Like Java's **`x = new Cons(42,null)`**, not **`x.car = 42`**

mcons cells are mutable

Since mutable pairs are sometimes useful (will use them later in lecture), Racket provides them too:

- **`mcons`**
- **`mcar`**
- **`mcdr`**
- **`mpair?`**
- **`set-mcar!`**
- **`set-mcdr!`**

Run-time error to use **`mcar`** on a cons cell or **`car`** on a mcons cell

Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are *eager* (call-by-value)
- Conditional branches are not

It matters: calling **`fact-wrong`** never terminates:

```
(define (my-if-bad x y z)
  (if x y z))

(define (fact-wrong n)
    (my-if-bad (= n 0)
               1
               (* n (fact-wrong (- n 1)))))
```

Thunks delay

We know how to delay evaluation: put expression in a function!

- Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a *thunk*

- As a verb: *thunk the expression*

This works (though silly to wrap **`if`** like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
    (my-if (= n 0)
           (lambda() 1)
           (lambda() (* n (fact (- n 1))))))
```

Avoiding expensive computations

Thunks let you skip expensive computations if they aren't needed

Great if take the true-branch:

```
(define (f th)
  (if (…) 0 (…   (th) …)))
```

But a net-loss if you end up using the thunk more than once:

```
(define (f th)
  (… (if (…) 0 (… (th) …))
     (if (…) 0 (… (th) …))
     …
     (if (…) 0 (… (th) …))))
```

In general, might now how many (more) times result is needed

Best of both worlds

Assuming our expensive computation has no side effects, ideally we would:

- Not compute it until needed
- Remember the answer so future uses complete immediately

Called *lazy evaluation*

Languages where most constructs, including function calls, work this way are *lazy languages*

- Haskell

Racket predefines support for *promises*, but we can make our own

- Thunks and mutable pairs are enough

Delay and force

```
(define (my-delay th)
  (mcons #f th))

(define (my-force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

An ADT represented by a mutable pair

- **`#f`** in car means cdr is unevaluated thunk
- Ideally hide representation in a module

Using promises

```
(define (f p)
  (… (if (…) 0 (… (my-force p) …))
     (if (…) 0 (… (my-force p) …))
     …
     (if (…) 0 (… (my-force p) …))))
```

```
(f (my-delay (lambda () e)))
```

Streams

- A stream is an *infinite sequence* of values
  - So can't make a stream by making all the values
  - Key idea: Use a thunk to delay creating most of the sequence
  - Just a programming idiom

A powerful concept for division of labor:

- Stream producer knows how create any number of values
- Stream consumer decides how many values to ask for

Some examples of streams you might (not) be familiar with:

- User actions (mouse clicks, etc.)
- UNIX pipes: **`cmd1 | cmd2`** has **`cmd2`** "pull" data from **`cmd1`**
- Output values from a sequential feedback circuit

Using streams

Coding up a stream in your program is easy

- We will do functional streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

```
'(next-answer . next-thunk)
```

So given a stream **`st`**, the client can get any number of elements

- First: **`(car (s))`**
- Second: **`(car ((cdr (s))))`**
- Third: **`(car ((cdr ((cdr (s))))))`**

(Usually bind **`(cdr (st))`** to a variable or pass to a recursive function)

Example using streams

This function returns how many stream elements it takes to find one for which tester does not return **`#f`**

- Happens to be written with a tail-recursive helper function

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                   (if (tester (car pr))
                       ans
                       (f (cdr pr) (+ ans 1)))))])
     (f stream 1)))
```

- **`(stream)`** generates the pair
- So recursively pass **`(cdr pr)`**, the thunk for the rest of the infinite sequence

Making streams

- How can one thunk create the right next thunk? Recursion!
  - Make a thunk that produces a pair where cdr is next thunk

```
(define ones (lambda () (cons 1 ones)))

(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1)))))])
     (lambda () (f 1))))

(define powers-of-two
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (* x 2)))))])
     (lambda () (f 2))))
```

- Why is this wrong?

```
(define ones-bad (lambda () (cons 1 (ones-bad))))
```

Memoization

- If a function has no side effects and doesn't read mutable memory, no point in computing it twice for the same arguments
  - Can keep a *cache* of previous results
  - Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

- Similar to how we implemented promises, but the function takes arguments so there are multiple "previous results"

- For recursive functions, this *memoization* can lead to *exponentially* faster programs
  - Related to algorithmic technique of dynamic programming

How to do memoization: see example

- Need to create a (mutable) cache that all calls using the cache shared
  - That is, must be defined outside the function(s) using it

- See **`lec15.rkt`** for an example with fibonacci numbers

  - Good demonstration of the idea because it is short, but, as shown in the code, there are also easier less-general ways to make **`fibonacci`** efficient

  - (An association list (list of pairs) is a simple but sub-optimal data structure for a cache; okay for our example)