MUTABLE DATA 15

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 19, 2010

1 Review

The readers have asked me to talk about code like this:

```
(if (predicate?) #t #f)
```

What's wrong with this?

2 Introduction to Mutable Data

So last week, we talked about the ability of objects to have local state. Up until now, all our data structures have been functional. In other words, we never changed a data structure, we've always made new ones. Another word for this type of data structure is IMMUTABLE. In other words, it's something that doesn't change. Today and tomorrow, we'll analyze mutable data structures, and look at some advantages and disadvantages that things like this can have.

2.1 Building a Pair in OOP

So let's think about a pair, as defined in OOP notation:

```
(define-class (pair car cdr))
```

Well...that was anticlimactic. This class can do everything we know we can already do with a pair. But let's try to expand this a bit:

```
(define-class (pair car cdr)
  (method (set-car! new-car) (set! car new-car))
  (method (set-car! new-cdr) (set! cdr new-cdr))
```

Now we have a pair that can change its values!

One thing that I want you guys be very careful about is that note that set-car! and set-cdr! are messages that we send to pairs, not values! In other words, a pair is able to change its own values, but you have to ask a pair to change its own values.

2.2 Regular Scheme

Well, we already have talked about the Constructors and Selectors for pairs:

```
(cons a d)
(car pair)
(cdr pair)
```

So now let's introduce 2 MUTATORS:

```
(set-car! PAIR NEW-VALUE)
(set-cdr! PAIR NEW-VALUE)
```

Note that the first argument to the mutators is not a value, it is a pair! In other words, do not say `(set-car! (car pair) 3)` on `(define pair (cons 5 6)`.

3 Equality and Identity

Once we have mutation we need a subtler view of the idea of equality. Up to now, we could just say that two things are equal if they look the same. Now we need two kinds of equality, that old kind plus a new one: Two things are identical if they are the very same thing, so that mutating one also changes the other.

Example:

```
> (define a (list 'x 'y 'z))
> (define b (list 'x 'y 'z))
> (define c a)
> (equal? b a)
#T
> (eq? b a)
#F
> (equal? c a)
#T
> (eq? c a)
#T
```

The two lists `a` and `b` are equal, because they print the same, but they're not identical. The lists `a` and `c` are identical; mutating one will change the other:

```
> (set-car! (cdr a) 'foo)
> a
(X FOO Z)
> b
(X Y Z)
> c
(X FOO Z)
```

If we use mutation we have to know what shares storage with what. For example, `(cdr a)` shares storage with `a`. `(append a b)` shares storage with `b` but not with `a`. (Why not? Read the `append` procedure.)

The Scheme standard says you're not allowed to mutate quoted constants. That's why I said (list 'x 'y 'z) above and not '(x y z). The text sometimes cheats about this. The reason is that Scheme implementations are allowed to share storage when the same quoted constant is used twice in your program.

4 Memoization

Exercise 3.27 from SICP is a pain in the neck because it asks for a very complicated environment diagram, but it presents an extremely important idea. If we take the simple Fibonacci number program:

```
;;;;; In file cs61a/lectures/3.3/fib.scm
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

we recall that it takes $\Theta(2^n)$ time because it ends up doing a lot of subproblems redundantly. For example, if we ask for `(fib 5)` we end up computing `(fib 3)` twice. We can fix this by remembering the values that we've already computed. The book's version does it by entering those values into a local table. It may be simpler to understand this version, using the global `get/put`:

```
;;;;; In file cs61a/lectures/3.3/fib.scm
(define (fast-fib n)
  (if (< n 2)
      n ; base case unchanged
      (let ((old (get 'fib n)))
        (if (number? old) ; do we already know the answer?
            old
            (begin ; if not, compute and learn it
              (put 'fib n (+ (fast-fib (- n 1))
                             (fast-fib (- n 2))))
              (get 'fib n))))))
```

Is this functional programming? That's a more subtle question than it seems. Calling `memo-fib` makes a permanent change in the environment, so that a second call to `memo-fib` with the same argument will carry out a very different (and much faster) process. But the new process will get the *same answer*! If we look inside the box, `memo-fib` works non-functionally. But if we look only at its input-output behavior, `memo-fib` is a function because it always gives the same answer when called with the same argument. What if we tried to memoize `random`? It would be a disaster; instead of getting a random number each time, we'd get the same number repeatedly! Memoization only makes sense if the underlying function really is functional. This idea of using a non-functional implementation for something that has functional behavior will be very useful later when we look at streams.

4.1 Dynamic Programming

This idea of solving partial problems turns out to be very powerful. Don't worry about knowing this section for exams, but I think this is an interesting enough idea to talk about.

Dynamic Programming is the idea that a big problem is composed of smaller problems that can be solved individually. A fibonacci number is a perfect example of this. Instead of computing the numbers from top down, it's often easier to compute them bottom up.

Another example of this is edit distance. This is a metric of how far away 2 words are, if you are allowed only to insert, delete, or replace letters at a time. Such a distance can be called "Levenstein Edit Distance" and is useful in a number of applications from spelling correction to Genomics.

5 Applications

5.1 Guessing Game

5.2 Tables

Tables. We're now ready to understand how to implement the put and get procedures that A&S used at the end of chapter 2. A table is a list of key-value pairs, with an extra element at the front just so that adding the first entry to the table will be no diffferent from adding later entries. (That is, even in an "empty" table we have a pair to set-cdr!)

```
;;;;; In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))
(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table (cons (cons key value)
                                  (cdr the-table)))
        (set-cdr! record value)))
    'ok)
(define the-table (list '*table*))
```

Assoc is in the book:

```
(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records))
         (car records))
        (else (assoc key (cdr records))) ))
```

In chapter 2, A&S provided a single, global table, but we can generalize this in the usual way by taking an extra argument for which table to use. That's how `lookup` and `insert!` work. One little detail that always confuses people is why, in creating two-dimensional tables, we don't need a `*table*` header on each of the subtables. The point is that `lookup` and `insert!` don't pay any attention to the `car` of that header pair; all they need is to represent a table by some pair whose `cdr` points to the actual list of key-value pairs. In a subtable, the key-value pair from the top-level table plays that role. That is, the entire subtable is a value of some key-value pair in the main table. What it means to be "the value of a key-value pair" is to be the `cdr` of that pair. So we can think of that pair as the header pair for the subtable.