Mutation: Succumbing to the Dark Side?

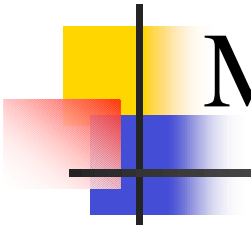
Corky Cartwright
Department of Computer Science
Rice University

Motivation

- Four common problems:
  - Assume that we are repeatedly evaluating a method/function `m` often evaluating `m` on the same list of arguments. How can we avoid performing the same computation more than once?
  - Assume we want to compute the number of a nodes in a tree data structure where nodes can be shared (the standard situation in functional programming or OO programming with immutable data). How can we efficiently perform this computation.
  - Perhaps simplest data structure from the perspective of machine implementation is the array: a fixed-size list of elements T that is allocated in contiguous machine memory where each element T is represented by a fixed size chunk of memory. The array was the *only* data structure in the original Fortran language. How can we create such structures using simple machine operations? How can we efficiently compute new ones?
  - How can I represent cyclic linked structures (general graphs rather trees)?
- The best (and hence light rather dark) solutions to these four problems all rely on *mutation*

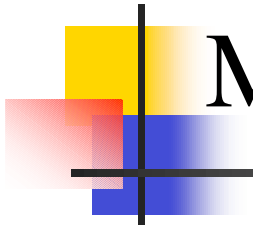
Mutation: Definition

- *Mutation is rebinding a variable to a new value*. What is a variable? A cell in computer memory containing a value such as an `int` or a reference (address of) to an `Object`. Rebinding that variable requires destroying the former binding, replacing the contents of the memory cell (for the variable) with new value.
- Mutation is nearly non-existent in mathematics. We don't change numbers or functions; we simply construct new one. Why? From the perspective of human thought, creating new values is much simpler. We don't have to remember what changes have been made to any existing value and there is no extra cost incurred in creating new mathmatical objects as opposed to changing existing ones.
- In computation, the tradeoffs are different. Mutation has a large conceptual overhead--we have remember exactly what has changed at any point in a computation--but it also has huge efficiency and modeling advantages. The efficiency advantage is that the cost of creating a new data structure (assuming we can dispense with an existing one) is simply the cost of the changes (differences) between the new structure and the old one.



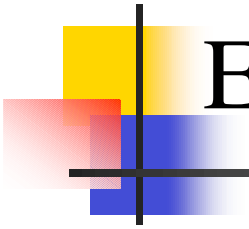
Modeling Involves Mutation

- In many computational models, objects in the model evolve over time. Examples:
  - Bank accounts
  - Stock prices
  - Enrollment in a college class
  - Temperature in your dorm room
- Physical systems change over time, but the identities of the objects in the system changes much less often that the properties of those objects. Example: humanity. Every few seconds, significant properties of almost every human being change (location, heart rate, posture, etc.) but new human beings are born infrequently (relative to changes in the status of the existing population).



Mutation Manifesto

- *Execution recapitulates system evolution*
- Given a physical system, it evolves in time. In most computations, the natural way to model this evolution is to simply update a data structure representing the state of the system.
- What is the functional (immutable alternative)? Modeling physical systems as functions mapping time to states. But this is expensive (and in many cases conceptually exhausting) because all history is explicitly retained in the computational model.



Example 1: Memo Functions

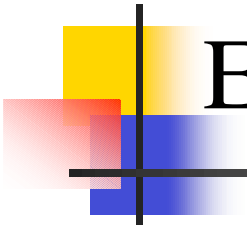
- Consider a naive program to compute the Fibonacci function. How can we speed it up without any mathematical reworking of the problem. Brute force speed-up:

```
class MyMath {
  static long fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
  }
}
```

Memo Functions cont.

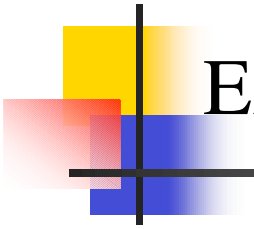
- We can avoid recomputing `fib(n)` for a given value of `n` by maintaining a table that records all previously computed values. We will use a HashMap for this purpose although we could easily use an expandable array to represent the table with less (constant factor) executiion overhead but more programming overhead

```
import java.util.HashMap;
class BetterMath {
  static HashMap<Integer, Long> Fib = new HashMap<Integer, Long>();
  static long fib(int n) {
  if (n <= 1) return 1;
  else {
    Long cachedAnswer = Fib.get(n);
    if (cachedAnswer != null) return cachedAnswer;
    else {
      long newAnswer = fib(n-1) + fib(n-2);
      Fib.put(n, newAnswer);
      return newAnswer;
    }
  }
}
```



Example 2: Counting Tree Nodes

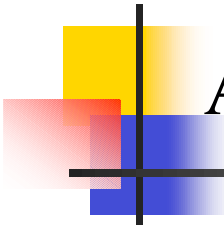
- Idea: we avoid counting a node more than once. How can we do this? When we start to visit a node, abort the visitation if node has "already been visited". How do we determine if a node has "already been visited"?
  - Add a boolean "flag" field to our node representation initialized to false and mutate it to true when a node is visited.
    - Requires changing the node representation.
    - Boolean flags be cleared (requiring a tree traversal) before reuse.
  - Add a `static HashSet<Node>` field to node class (or other convenient class) that holds the set of nodes that have been visited.
    - Less intrusive; node representation is unchanged.
    - Slightly more overhead. How is `HashSet` implemented?



Example 3: Initializing and Manipulating Arrays

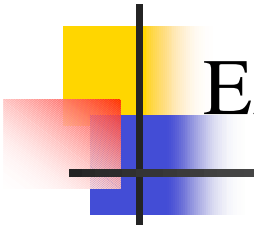
- Can arrays be incorporated in a functional language? Yes but they can only be used to hold immutable tables mapping $0 < i < n$ to some type T. How can we create them? We need a primitive operation that takes two arguments n and f:nat → T that specifies the value $f(i)$ of the ith array element.
- How can I initialize arrays without functions as data (alternatively using simple machine operations)? By allocating a block of memory (of proper size) and mutating the elements in that block. Use a loop (a special form of tail recursion):

```
for (int i = 0; 0 < n; i++) a[i] = <some expression in i>;
```



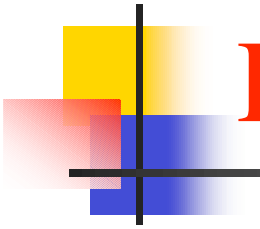
Arrays cont.

- Good online language feature/syntax (but not style or design!) reference: http://leepoint.net/notes-java/
- Arrays are important if you need to squeeze the last possible bit of overhead out of a computation. For example, quicksort and insertion sort can be done more efficiently (particularly quicksort) if lists are represented as arrays and those arrays are mutated.



Example 4: Cyclic Linked Structures

- Josephus Problem: naive simulation of cannibals killing Christians arranged in a circle.
- Regular infinite trees or lists (analogous to repeating decimals) are easily represented by lists extended to allow back pointers. Example:
  zeroes = cons(0, zeroes) (where cons is lazy [call-by-name])
- How can we implement zeroes? In R5RS Scheme:
  (define zeroes (cons 0 empty))
  (set-cdr! zeroes zeroes)



For Next Class

- Homework due on Wednesday. Should have working interpreter code by Saturday night.
- Big tests (like bigData) will be installed this afternoon. The bigger ones will require enlarging the stack of the DrJava interactions JVM. Insert the argument string
  `-Xss64M`
  in the dialog box labeled JVM args for Interactions JVM in the Miscellaneous panel of DrJava Preferences.