




Mutation and Bi-Directional Linked Lists

Corky Cartwright

Vivek Sarkar

Department of Computer Science

Rice University



Mutation: Succumbing to the Dark Side? (Lecture 26)

- Four common problems in computing:
 - 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 OO programming with immutable data). How can we efficiently perform this computation?
 - Perhaps the simplest data structure from the perspective of machine implementation is the array: a fixed-size list of elements that is allocated in contiguous machine memory where each element **e** 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 solutions to these four problems all rely on data **mutation**.



Mutation: Definition

- *Mutation is **re**binding 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 to (address of) an **Object**. Rebinding that variable destroys the former binding, replacing the contents of the memory cell (for the variable) with a 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 mathematical objects as opposed to changing existing ones.
- In computation, the trade-offs are different. Mutation has a large conceptual overhead--we have to 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 (and roster) of a college class
 - Temperature in your dorm room
- Physical systems change over time, but the identities of the objects in the system change much less often than 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). Consider the human beings in this room.



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);  
    }  
}
```

- Aside: what is **static**?



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) execution 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 array construction operation that takes two arguments n and a function f mapping int to T that specifies the value $f(i)$ of the i th array element.
- How can we initialize arrays without using functions as data (alternatively, *only* 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) like the following:

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



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)`



Background on Lists

- Scheme lists and composite pattern lists in Java are internally represented using a linked list of **Cons** nodes. Each **Cons** node **N** is a chunk of memory containing a field **first** and a field **rest**. In each node **N**, these fields are the addresses of:
 - the object **o** that is the first element in the list rooted at **N** and
 - the **Cons** node **N'** representing the rest of the list rooted at **N'**.
- In functional programming (Java programming with immutable objects), these fields are never modified after they are initialized. In imperative (mutable data) programming, they can be modified.
- Mutation can be performed with discipline and taste. We will focus initially on the mutable generalization of composite lists.



Mutable Generalization of Functional Lists

In Comp 212, we would introduce the notion of QuasiLists (LRS structures in the terminology of Nguyen and Wong) which make the **first** and **rest** fields of a **Cons** node mutable. See the class notes on OO design.

- But QuasiLists provide no asymptotic speed-up over functional lists. Inserting or removing elements from the end of a list takes $O(n)$ time.
- Traditional linked lists can provide asymptotic speed-ups.
- OO style dictates the disciplined use of mutation
 - Never modify fields directly.
 - Support high level mutation via mutating methods.



Example: BiLists

- In Comp 212, we would introduce mutable "singly linked lists" first. Functional lists are singly linked and mutable singly linked lists are lighter weight (simpler and, in many cases, faster) See the OO design notes. Allowing mutation on singly linked lists asymptotically speeds-up some operations on lists, but others (such as deleting the last element of a list) take $O(n)$ time in the absence of double-linking.
- Furthermore, representing nodes as objects adds weight (a two machine word header in each node) to a linked-list implementation so double-linking adds only modest extra space (one word) and time cost (allocating and reclaiming more space takes more time).
- A doubly-linked representation adds a predecessor address field to each **Cons** node.



Comments on BiList code

- Supports the *iterator* design pattern, which is applicable to any data structure that holds a collection of items.
- Key operations:
 - *Factory method* (design pattern) for constructing an iterator
 - Method for advancing the iterator cursor
 - Method for getting the current item
 - Method for testing whether cursor is at the end enumerating the collection.



Code: BiIterator from BiList.java (HW 10)

```
private class BiIterator implements BiIteratorI<T> {
    Node<T> current;
    BiIterator() {
        current = BiList.this.head.succ; // current is first item (if one exists)
    }
    public void first() {
        current = BiList.this.head.succ; // current is first item (if one exists)
    }
    public void last() {
        current = BiList.this.head.pred; // current is last item (if one exists)
    }

    public void next() {
        current = current.succ;           // wraps around end
    }

    public void prev() {
        current = current.pred;          // wraps around end
    }
    public T currentItem() {
        if (current == BiList.this.head) {
            throw new IteratorException("No current element in " + BiList.this);
        }
        return current.item;
    }
}
```



Biterator class (cont.)

```
public boolean atEnd() {  
    return current == BiList.this.head;  
}  
} // BiIterator
```

This is advanced Java code. Why? `BiIterator` is a private, inner class.