CSE341 – Section 6
Memoization, Streams, and More

Cody Schroeder

February 13th, 2013

Outline

1. SML/Racket
   - Refresher
   - Lexical Scope
   - Mutation

2. Memoization
   - Fibonacci
   - General Memoization

3. Streams
   - Definition
   - Working with Streams

Refresher

Racket Fibonacci

```racket
(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

SML Fibonacci

```sml
fun fib n = if n <= 1
  then n
  else fib (n-1) + fib (n-2)
```

SML and Racket aren’t so different a lot of the time. A lot of what we learned in SML will transfer over. Functional constructs are still frequently used.

Lexical Scope

Variable lookup rules are nearly identical between SML and Racket. One difference is the top-level `letrec` in a Racket module.

How do these procedures differ?

- **Racket**
  ```racket
  (define minus-fact-of-36
    (let ([v (fib 36)])
      (lambda (x)
        (- x v))))
  ```
  - Computes `(fib 36)` once

- **SML**
  ```sml
  fun minus-fact-of-37 =
    (let ([v (fib 37)])
      (lambda (x)
        (- x v)))
  ```
  - Computes `(fib 37)` every call

Mutation

We care even more about scoping rules in the presence of mutation.

What do these procedures do when called?

```racket
(define increment-and-return1
  (let ((v 0))
    (lambda (x)
      (begin (set! v (+ x v)) v)))
)
```

Incorrect: will always return `x`

```
(define increment-and-return2
  (lambda (x)
    (let ((v 0))
      (begin (set! v (+ x v)) v))))
```

increment-and-return is meant to be a function that keeps a global counter and increments the counter with `x` during each call.

Mutation Functions

- In Racket there are multiple functions that have mutation as a side-effect.
  - `set!` assigns to some variable. It updates its value in the environment.
  - In Java, analogous to `x = 5;` (where 5 is just some value)
  - `set-mcar!` and `set-mcdr!` assigns to the fields of a mpair structure.
  - `car` and `cdr` could be considered fields in a mpair structure
  - In Java, analogous to `x.car = 5;` and `x.cdr = 10;`

Why is this procedure slow?

(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

Our fibonacci function ends up recomputing many values in the long run due to the recursive structure of the solution.
- How can we fix this? *Other than using an iterative solution*...
- How about we store already computed results in some sort of cache?
- The cache could be a mutable structure that will be added to as new results are computed.
- This is the idea of memoization!
- In the previous tree example, the entire right subtree doesn’t have to be recomputed. It’ll be found in the cache.
- Our fibonacci function will become exponentially faster.

A Memoized Fibonacci

(define fib
    (let ((memo '((0 . 0) (1 . 1)))
        (lambda (n)
          (let ((prev-ans (assoc n memo)))
            (if prev-ans
                (cdr prev-ans)
                (let ((ans (+ (fib (- n 1)) (fib (- n 2))))
                    (set! memo (cons (cons n ans) memo))
                    ans))))))

How fast can (fib 70000) be computed now?

There’s a lot of redundant computation in this implementation.

Memoization

Our fibonacci function ends up recomputing many values in the long run due to the recursive structure of the solution.
- How can we fix this? *Other than using an iterative solution*...
- How about we store already computed results in some sort of cache?
- The cache could be a mutable structure that will be added to as new results are computed.
- This is the idea of memoization!
- In the previous tree example, the entire right subtree doesn’t have to be recomputed. It’ll be found in the cache.
- Our fibonacci function will become exponentially faster.
A Stream is...
- A thunk that evaluates to a pair of an element and another stream.
- This is an infinitely recursive definition. There's no end to a stream.

Example
1 (define natural-numbers
2  (letrec ([next-nat (lambda (n)
3        (letrec ([nxt (lambda () (cons n (next-nat (+ 1 n)))))
4         (next-nat 1)])]

See code: streams.rkt.