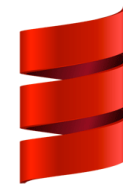


CS 360

Programming Languages

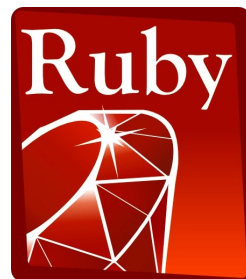
Streams Wrapup



Scala



Swift



Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 1: Use a function that takes a(n) argument(s) from which the next element of the stream can be constructed.

```
(define (integers-from n)
  (stream-cons n (integers-from (+ n 1))))
(define ints-from-2 (integers-from 2))
```

- When you use this technique, your code usually looks a lot like you have infinite recursion.
- Often the code is very clear (easy to see how it works).

Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 2: Construct the stream directly by defining it in terms of a modified version of another stream or itself.

```
(define ints-from-2-alt  
  (stream-cons 2  
    (stream-map (lambda (x) (+ x 1))  
                ints-from-2-alt)))
```

- This technique is fine, but can be harder to figure out how it works.

Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 2: Construct the stream directly by defining it in terms of a modified version of another stream or itself.

```
(define ints-from-2-alt-alt
  (stream-cons 2
    (stream-map2 +
      infinite-ones
      ints-from-2-alt-alt)))
```

Fibonacci

- Method 1:

```
(define (make-fib-stream a b)
  (stream-cons a (make-fib-stream b (+ a b))))
```

```
(define fibs1 (make-fib-stream 0 1))
```

Fibonacci

- Method 2:

```
(define fibs
  (stream-cons 0
    (stream-cons 1
      (stream-map2 + (stream-cdr fibs) fibs))))
```

Sieve of Eratosthenes

- Start with an infinite stream of integers, starting from 2.
- Remove all the integers divisible by 2.
- Remove all the integers divisible by 3.
- Remove all the integers divisible by 5...etc

Sieve of Eratosthenes

```
(define (not-divisible-by s div)
  (stream-filter
    (lambda (x) (> (remainder x div) 0)) s))

(define (sieve s)
  (stream-cons
    (stream-car s)
    (sieve (not-divisible-by s (stream-car s)))))

(define primes (sieve ints-from-2))
```


Fibonacci

```
(define (make-fib-stream a b)
  (stream-cons a (make-fib-stream b (+ a b))))
(define fibs1 (make-fib-stream 0 1))
```

- More efficient (but less clear?) than

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#t (+ (fib (- n 1)) (fib (- n 2))))))
```

- How to get the best of both worlds?

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

```

(define fast-fib
  (let ((cache ' ()))
    (define (lookup-in-cache cache n)
      (cond ((null? cache) #f)
            ((= (caar cache) n) (cadar cache))
            (#t (lookup-in-cache (cdr cache) n))))

    (lambda (n)
      (if (or (= n 0) (= n 1)) n
          (let ((check-cache (lookup-in-cache cache n)))
            (cond ((not check-cache)
                   (let ((answer (+ (fast-fib (- n 1))
                                     (fast-fib (- n 2))))
                       (set! cache (cons (list n answer) cache))
                       answer))
                  (#t check-cache)))))))

```

Memoization in other languages

- Code for memoization is often easier with an explicit hashtable data structure:

```
int fib(int n) {
    static map<int, int> cache;
    if (n < 2) return n;
    if (cache.count(n) == 0) {
        int ans = fib(n-1) + fib(n-2);
        cache[n] = ans;
        return ans;
    } else return cache[n];
}
```

Memoization wrapup

- Memoization is related to streams in that streams also remember their previously-computed values.
 - Remember how promises save their results and return them instead of re-computing?
- But memoization is more flexible because it works with any function.
- Memoization is a classic example of the time-space trade-off in CS:
 - With memoization, we use more space, but use less time.