Streams and Lazy Evaluation in Lisp and Scheme

Overview

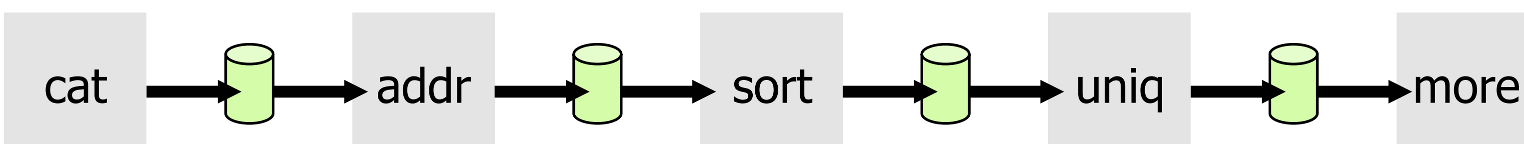
- Different models of expression evaluation
  - Lazy vs. eager evaluation
  - Normal vs. applicative order evaluation
- Computing with streams in Lisp and Scheme

Motivation

- Streams in Unix
- Modeling objects changing with time without assignment.
    - Describe the time-varying behavior of an object as an infinite sequence x1, x2,…
    - Think of the sequence as representing a function x(t).
- Make the use of lists as conventional interface more efficient.

Unix Pipes

- Unix's pipe supports a kind of stream oriented processing
- E.g.: % cat mailbox | addresses | sort | uniq | more
- Output from one process becomes input to another. Data flows one buffer-full at a time
- Benefits:
  - we may not have to wait for one stage to finish before another can start;
  - storage is minimized;
  - works for infinite streams of data



Evaluation Order

- Functional programs are evaluated following a *reduction* (or evaluation or simplification) process
- There are two common ways of reducing expressions
  - Applicative order
    - Eager evaluation
  - Normal order
    - Lazy evaluation

Applicative Order

- In applicative order, expressions at evaluated following the parsing tree (deeper expressions are evaluated first)
- This is the evaluation order used in most programming languages
- It's the default order for Lisp, in particular
- All arguments to a function or operator are evaluated before the function is applied

  e.g.: (square (+ a (* b 2)))

Normal Order

- In normal order, expressions are evaluated only when their value is needed
- Hence: *lazy evaluation*
- This is needed for some special forms

  e.g., (if (< a 0) (print ‘foo) (print ‘bar))
- Some languages use normal order evaluation as their default.
  - Its sometimes more efficient than applicative order since unused computations need not be done
  - It can handle expressions that never converge to normal forms

Motivation

- Goal: sum the primes between two numbers
- Here is a standard, traditional version using Scheme's iteration special form, do

```
(define (sum-primes lo hi)
  ;; sum the primes between LO and HI
  (do [ (sum 0) (n lo (add1 n)) ]
       [(> n hi) sum]
       (if (prime? N)
            (set! sum (+ sum n))
            #t)))
```

Motivation: prime.ss

Here is a straightforward version using the "functional" paradigm:

```
(define (sum-primes lo hi)
  ; sum primes between LO and HI
  (reduce + 0 (filter prime? (interval lo hi))))

(define (interval lo hi)
    ; return list of integers between lo and hi
    (if (> lo hi)
        empty
        (cons lo (interval (add1 lo) hi))))
```

Prime?

```
(define (prime? n)
  ;; returns #t iff n is a prime integer
  (define (evenly-divides? m) (= (remainder n m) 0))
  (not (some evenly-divides? (interval 2 (/ n 2)))))

(define (some F L)
  ;; returns #t iff predicate f is true of some element in list l
  (cond ((null? L) #f)
        ((F (first L)) #t)
        (else (some F (rest L)))))
```

Motivation

- The functional version is interesting and conceptually elegant, but inefficient
  - Constructing, copying and (ultimately) garbage collecting the lists adds a lot of overhead
  - Experienced Lisp programmers know that the best way to optimize is to eliminate unnecessary consing
- Worse yet, suppose we want to know the second prime larger than a million?

  (car (cdr (filter prime? (interval 1000000 1100000))))
- Can we use the idea of a stream to make this approach viable?

A Stream

- A stream will be a collection of values, much like a List
- It will have a first element and a stream of remaining elements
- However, the remaining elements will only be computed (*materialized*) as needed
  - Just in time computing, as it were
- So, we can have a stream of (potential) infinite length and use only a part of it without having to materialize it all

Streams in Lisp and Scheme

- We can push features for streams into a programming language.
  - Makes some approaches to computation simple and elegant
  - The closure mechanism used to implement these features.
- Can formulate programs elegantly as sequence manipulators while attaining the efficiency of incremental computation.

Streams in Lisp/Scheme

- A stream is like a list, so we'll need constructors (~cons), and accessors (~ car, cdr) and a test (~ null?).
- We'll call them:
  - SNIL: represents the empty stream
  - (SCONS X S): create a stream whose first element is X and whose remaining elements are the stream S
  - (SCAR S): returns first element of the stream
  - (SCDR S): returns remaining elements of the stream
  - (SNULL? S): returns true iff S is the empty stream

Streams: key ideas

- Write scons so that the computation needed to produce the stream is delayed until it is needed
  - … and then, only as little of the computation possible will be done
- Only ways to access parts of a stream are *scar* & *scdr*, so they may have to force the computation to be done
- We'll go ahead and always compute the first element of a stream and delay actually computing the rest of a stream until needed by some call to *scdr*
- Two important functions to base this on: *delay* & *force*

Delay and force

- (delay <exp>) ==> a "promise" to evaluate exp
- (force <delayed object>) ==> evaluate the delayed object and return the result

```
> (define p (delay (add1 1)))
> p
#<promise:p>
> (force p)
2
> p
#<promise!2>
> (force p)
2
```

```
> (define p2
    (delay (printf "FOO!\n")))
> p2
#<promise:p2>
> (force p2)
FOO!
> p2
#<promise!#<void>>
> (force p2)
```

Delay and force

- We want (delay S) to return the same function that just evaluating S would have returned

```
> (define x 1)
> (define p (let ((x 10)) (delay (+ x x))))
#<promise:p>
> (force p)
> 20
```

Delay and force

- Delay is built into scheme, but it would have been easy to add
- It’s not built into Lisp, but is easy to add
- In both cases, we need to use macros
- Macros provide a powerful facility to extend the languages

Macros

- In Lisp and Scheme macros let us extend the language
- They are syntactic forms with associated definition that rewrite the original forms into other forms before evaluating
  - E.g., like a compiler
- Much of Scheme and Lisp are implemented as macros

Simple macros in Scheme

- *(define-syntax-rule pattern template)*
- Example:

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

- Whenever the interpreter is about to eval something matching the pattern part of a syntax rule, it expands it first, then evaluates the result

Simple Macros

- (define foo 100)
- (define bar 200)
- (swap foo bar)

  (let ([tmp foo]) (set! foo bar)(set! bar tmp))
- foo
- 200
- bar
- 100

A potential problem

- (let ([tmp 5] [other 6])

  (swap tmp other)

  (list tmp other))
- A naïve expansion would be:
- (let ([tmp 5] [other 6])

```
(let ([tmp tmp])
  (set! tmp other)
  (set! other tmp))
(list tmp other))
```

- Does this return (6 5) or (5 6)?

Scheme is clever here

- (let ([tmp 5] [other 6])

  (swap tmp other)

  (list tmp other))
- (let ([tmp 5] [other 6])

  (let ([tmp_1 tmp])

  (set! tmp_1 other)

  (set! other tmp_1))

  (list tmp other))
- This returns (6 5)

mydelay in Scheme

```
➢(define-syntax-rule (mydelay expr)
        (lambda ( ) expr))
> (define (myforce promise)  (promise))
> (define p (mydelay (+ 1 2)))
> p
#<procedure:p>
> (myforce p)
3
> p
#<procedure:p>
```

mydelay in Lisp

```
(defmacro mydelay (sexp)
    `(function (lambda ( ) ,sexp)))

(defun force (sexp)
   (funcall sexp))
```

Streams using DELAY and FORCE

```
(define sempty empty)

(define (snull? stream) (null? stream))

(define-syntax-rule (scons first rest)
    (cons first (delay rest)))

(define (scar stream) (car stream))

(define (scdr stream) (force (cdr stream)))
```

Consider the interval function

- Recall the interval function:

```
 (define (interval lo hi)
    ; return a list of the integers between lo and hi
    (if (> lo hi) empty (cons lo (interval (add1 lo) hi))))
```

- Now imagine evaluating (interval 1 3):

(interval 1 3)

(cons 1 *(interval 2 3)*)

(cons 1 (cons 2 *(interval 3 3)*))

(cons 1 (cons 2 (cons 3 *(interval 4 3)*)))

(cons 1 (cons 2 (cons 3 ‘())))

➔ (1 2 3)

… and the stream version

- Here's a stream version of the interval function:

```
(define (sinterval lo hi)
  ; return a stream of integers between lo and hi
  (if (> lo hi)
      sempty
      (scons lo (sinterval (add1 lo) hi))))
```

- Now imagine evaluating (sinterval 1 3):

(sinterval 1 3)

(scons 1 . *#<procedure>))*

Stream versions of list functions

```
(define (snth n stream)
  (if (= n 0)
      (scar stream)
      (snth (sub1 n) (scdr stream))))

(define (smap f stream)
  (if (snull? stream)
      sempty
      (scons (f (scar stream))
             (smap f (scdr stream)))))

(define (sfilter f stream)
  (cond ((snull? stream) sempty)
        ((f (scar stream))
         (scons (scar stream) (sfilter f (scdr stream))))
        (else (sfilter f (scdr stream)))))
```

Applicative vs. Normal order evaluation

```
(car (cdr
  (filter prime? (interval 10 1000000))))
```

```
(scar
  (scdr
    (sfilter prime? (interval 10 1000000))))
```

Both return the second prime larger than 10 (which is 13)

- With lists it takes about 1000000 operations
- With streams about three

Infinite streams

```
(define (sadd s1 s2)
  ; returns a stream which is the pair-wise
  ; sum of input streams S1 and S2.
  (cond ((snull? s1) s2)
        ((snull? s2) s1)
        (else (scons (+ (scar s1) (scar s2))
                     (sadd (scdr s1)(scdr s2))))))
```

Infinite streams 2

- This works even with infinite streams
- Using *sadd* we define an infinite stream of ones:

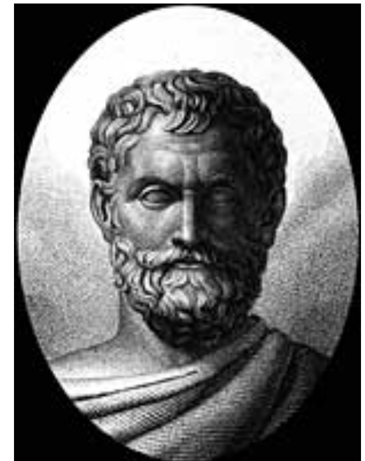
  (define ones (scons 1 ones))
- An infinite stream of the positive integers:

  (define integers (scons 1 (sadd ones integers)))

The streams are computed as needed

(snth 10 integers) => 11

Sieve of Eratosthenes



Eratosthenes (air-uh-TOS-thuh-neez), a Greek mathematician and astronomer, was head librarian of the Library at Alexandria, estimated the Earth's circumference to within 200 miles and derived a clever algorithm for computing the primes less than N

1. Write a consecutive list of integers from 2 to N
2. Find the smallest number not marked as prime and not crossed out. Mark it prime and cross out all of its multiples.
3. Goto 2.

Finding all the primes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	10
									0

Scheme sieve

```
(define (sieve S)
  ; run the sieve of Eratosthenes
  (scons (scar S)
         (sieve
           (sfilter
             (lambda (x) (> (modulo x (scar S)) 0))
             (scdr S)))))

(define primes (sieve (scdr integers)))
```

Remembering values

- We can further improve the efficiency of streams by arranging for automatically convert to a list representation as they are examined.
- Each delayed computation will be done once, no matter how many times the stream is examined.
- To do this, change the definition of SCDR so that
  - If the cdr of the cons cell is a function (presumable a delayed computation) it calls it and destructively replaces the pointer in the cons cell to point to the resulting value.
  - If the cdr of the cons cell is not a function, it just returns it

Summary

- Scheme's functional foundation shows its power here
- Closures and macros let us define delay and force
- Which allows us to handle large, even infinte streams easily
- Other languages, including Python, also let us do this