Streams and Lazy Evaluation in Lisp

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

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
- EG: \((\text{square} (+ a (* b 2)))\)

Normal Order

- In normal order, expressions are evaluated only their value is needed
- Hence: lazy evaluation
- This is needed for some special forms
  - e.g., \((\text{if} (< a 0) \text{print 'foo} \text{print 'bar}))\)
- Some languages use normal order evaluation as their default.
  - Normal is sometimes more efficient than applicative order since unused computations need not be done
  - Normal order can handle expressions that never converge to normal forms

Motivation

- Suppose we want to sum the primes between two numbers
- Traditional iterative version

```lisp
(defun sum-primes (lo hi)
 ;; sum the primes between LO and HI
 (do ((sum 0))
    ((> lo hi) sum)
    (if (prime lo)
        (setf sum (+ sum lo)))
    (setf lo (1+ lo))))
```

- And the “functional” version:

```lisp
(defun sum-primes (lo hi)
 ;; sum the primes between LO and HI
 (reduce #'+ 0 (filter prime? (interval lo hi))))

(defun interval (lo hi)
 ;; return a list of the integers between lo and hi
 (if (> lo hi) nil (cons lo (interval (1+ lo) hi))))
```
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 a program 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?

Streams in Lisp

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

Streams: key ideas

- A stream will be like a list, so we’ll need constructors (~ nil, cons), and accessors (~ car, cdr) and a test (~ null).
- We’ll call them:
  - SNIL: represents the empty stream
  - (SCONS X Y): create a stream whose first element is X and whose remaining elements are the stream S
  - (SCAR S): returns the first element of the stream
  - (SCDR S): returns the remaining elements of the stream
  - (SNULL S): returns true (T) iff S is the empty stream

- We’ll write SCONS so that the computation needed to actually produce the stream is delayed until it’s needed.
- ... and then, only as little of the computation possible will be done.
- The only way to access parts of a stream are SCAR and CDR, 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 CDR.
- Two important functions to base this on: DELAY and FORCE.
Delay and force

- \((\text{delay} <\text{exp}>) \implies \text{a "promise" to evaluate exp}\)
- \((\text{force} <\text{delayed object}>) \implies \text{evaluate the delayed object and return the result}\)

```lisp
> (setf foo (delay (+ 1 1)))
#<CLOSURE :LAMBDA NIL (+ 1 1)>
> foo
#<CLOSURE :LAMBDA NIL (+ 1 1)>
> (force foo)
2
```

Delay and force

- We want \((\text{DELAY} S)\) to return the same function that just evaluating \(S\) would have returned

```lisp
> (setf x 1)
1
> (setf foo (let ((x 100)) (delay (+ x x))))
#<CLOSURE :LAMBDA NIL (+ X X)>
> (force foo)
200
```

Delay and force

- Since \(\text{DELAY}\) doesn't evaluate its argument, it will have to be a macro.
- Since we want to ultimately evaluate the expression in the environment in \(\text{force}\) when \(\text{delay}\) was called, we'll use a closure
- So, \(\text{DELAY}\) and \(\text{FORCE}\) can be:
  ```lisp
  (defmacro delay (sexp)
   `(function (lambda () ,sexp))
  )
  (defun force (sexp)
   (funcall sexp)
  )
  ```

Streams using \(\text{DELAY}\) and \(\text{FORCE}\)

```lisp
(setf snil NIL)
(defun snull (stream) (equal stream snil))
(defmacro scons (first rest)  (cons ,first (delay ,rest)))
(defun scar (stream) (car stream))
(defun scdr (stream) (force (cdr stream)))
```
Consider the interval function

(defun interval (lo hi)
  ;; return a list of the integers between lo and hi
  (if (> lo hi) nil (cons lo (interval (1+ lo) hi))))

(defvar int123 (interval 1 3))

We’ll need stream versions of other familiar list manipulation functions

(defun snth (n stream)
  (if (= n 0)
    (scar stream)
    (snth (1- n) (scdr stream))))

(defun smapcar (f stream)
  (if (snull stream)
    snil
    (scons (funcall f (scar stream))
           (smapcar f (scdr stream)))))

(defun sfilter (f stream)
  (cond ((snull stream) snil)
        ((funcall f (scar stream))
         (scons (scar stream)
                 (sfilter f (scdr stream)))))
        (t (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

Consider:
```
(defun sadd (s1 s2)
  ;; returns a stream which is the pairwise
  ;; sum of input streams S1 and S2.
  (cond ((snull s1) s2)
        ((snull s2) s1)
        (t (scons (+ (scar s1) (scar s2))
            (sadd (scdr s1) (scdr s2))))))
```

Infinite streams 2

- This works even with infinite streams
- Using Sadd we can define an infinite stream of ones as:
  ```
  (setf ones (scons 1 ones))
  ```
- And an infinite stream of the positive integers as:
  ```
  (setf integers (scons 1 (sadd ones integers)))
  ```

The streams are computed as needed
```
(snth 10 integers) => 11
```

Sieve of Eratosthenes

- **Eratosthenes** was a Greek mathematician and astronomer who 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.

  - Basic idea for finding primes < 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 it’s multiples.
Finding all the primes

| 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 | 41 | 43 | 47 | 53 | 59 | 61 | 67 | 71 | 73 | 79 | 83 | 89 | 97 |

Lisp sieve

Here's a Lisp version using streams

```lisp
(defun sieve (S)
  ; ; run the sieve of Eratosthenes
  (cons (scar S)
    (sieve (sfilter (lambda (x)(> (mod x (scar S)) 0))(scdr S))))
)
```

Watch out for side effects, tho

- The value of a delayed operation depends on the actual time it is called which can be confusing.

```lisp
> (setf a 4)
4
> (setf b (delay (1+ a)))
#<CLOSURE ...>
> (force b)
5
> (setf a 0)
0
> (force b)
1
```

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 (presumably 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.
The new SCDR

- Here's the new SCDR function:
  (defun scdr (stream)
    (let ((theCdr (cdr stream)))
      (if (typep theCdr 'function)
          (setf (cdr stream) (force theCdr))
          theCdr))
  )

- In action:
  > integers
  (1 . #<CLOSURE :LAMBDA NIL (SADD ONES INTEGERS)>)
  > (scdr integers)
  (2 . #<CLOSURE :LAMBDA NIL (SADD (SCDR S1) (SCDR S2))>)
  > integers
  (1 2 . #<CLOSURE :LAMBDA NIL (SADD (SCDR S1) (SCDR S2))>)
  > (enth 8 integers)
  9
  > integers
  (1 2 3 4 5 6 7 8 9 . #<CLOSURE : ... >)