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 $x_1, x_2, \ldots$
  • 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 are 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 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.
  – 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
• Here is a standard, traditional version using iteration

(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

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)))))
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 w/o 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 will be 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 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 iff S is the empty stream
Streams: key ideas

• 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 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 and 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
mydelay in Scheme

```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) (equal stream sempty))

(define-syntax-rule (scons first rest)  
  (cons first (mydelay 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):
  (interval 1 3)
  (scons 1 . #<procedure>))
We’ll need stream versions of other familiar list manipulation 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 f) (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:

(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)
        (#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:
  (define ones (scons 1 ones))
• And an infinite stream of the positive integers as:
  (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-nee), 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.
Finding all the primes

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>14</td>
<td>xx</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>xx</td>
<td>xx</td>
<td>23</td>
<td>24</td>
<td>xx</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>xx</td>
</tr>
<tr>
<td>31</td>
<td>xx</td>
<td>33</td>
<td>34</td>
<td>xx</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>xx</td>
</tr>
<tr>
<td>41</td>
<td>43</td>
<td>44</td>
<td>xx</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>53</td>
<td>54</td>
<td>xx</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>63</td>
<td>64</td>
<td>xx</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>73</td>
<td>74</td>
<td>xx</td>
<td>76</td>
<td>xx</td>
<td>78</td>
<td>79</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>83</td>
<td>84</td>
<td>xx</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>93</td>
<td>94</td>
<td>xx</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Scheme sieve

(define (sieve S)
  ; run the sieve of Eratosthenes
  (scons (scar S)
    (sieve
      (sfilter
        (lambda (x) (> (mod 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