CS450 - Structure of Higher Level Languages

Infinite Streams

October 26, 2020

Infinitely Long Streams

We can use streams to represent infinitely long sequences:

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

(define integers (integers-starting-from 1))
```

Then we can filter these infinite sequences, just as before:

```
(define (divisible? x y)
  (= (remainder x y) 0) )

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

Infinitely Long Streams

If we define

```
(define (stream-ref stream n)
  (if (= n 0)
      (stream-car stream)
      (stream-ref (stream-cdr stream) (- n 1)) ))
```

then `(stream-ref no-sevens 100)` will evaluate to 117.
Here is a neat way to produce the Fibonacci sequence:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

Getting Prime Numbers

- The sieve of Eratosthenes is an efficient way to produce prime numbers.
- Given a prime number, filter out all of its multiples.
- Of the remaining numbers, the next one is the next prime.
- Repeat...
- Start from 2.

Getting Prime Numbers

Here is how we get the primes, using a form of the sieve of Eratosthenes:

```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

You can now try evaluating `(stream-ref primes 50)`.

Defining Streams Implicitly

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
  (stream-map + s1 s2))

(define integers (cons-stream 1
     (add-streams ones integers)))

(define fibs
  (cons-stream 0
        (cons-stream 1
              (add-streams (stream-cdr fibs)
                           fibs))))
```

Defining Streams Implicitly

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor)) stream))

(define double (cons-stream 1 (scale-stream double 2)))
```

Defining Streams Implicitly

```
;;; prime numbers -- really clever,
;;; and efficient because it only
;;; checks divisibility by numbers less than sqrt(n).

(define primes
  (cons-stream
   2
   (stream-filter prime? (integers-starting-from 3))))

(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) #t)
          ((divisible? n (stream-car ps)) #f)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

Streams of Pairs

- If we want to produce infinite streams of pairs we'll run into a problem because the "looping" must range over an infinite set.
- For example, suppose we want to produce the stream of pairs of all integers (i, j) with $i < j$ such that $i + j$ is prime.
- If `int-pairs` is the sequence of all pairs of integers (i, j) with $i < j$, we can do the following:

```
(stream-filter (lambda (pair)
                 (prime? (+ (car pair) (cadr pair))))
               int-pairs)
```

(we assume every pair is a list)

How Do We Produce int-pairs?

- More generally, suppose we have two streams $S = (S_i)$ and $T = (T_j)$, and imagine the infinite rectangular array

$$\begin{array}{lll} (S_0, T_0) & (S_0, T_1) & (S_0, T_2)... \\ (S_1, T_0) & (S_1, T_1) & (S_1, T_2)... \\ (S_2, T_0) & (S_2, T_1) & (S_2, T_2)... \end{array}$$

- We wish to generate a stream that contains all the pairs in the array that lie on or above the diagonal, i.e., the pairs

$$\begin{array}{lll} (S_0, T_0) & (S_0, T_1) & (S_0, T_2)... \\ & (S_1, T_1) & (S_1, T_2)... \\ & & (S_2, T_2)... \end{array}$$

- If both S and T are the streams of integers, this will be our desired `int-pairs`.

Streams of Pairs

- In general, the stream of pairs `(pairs S T)` is composed of three parts:
- the pair (S_0, T_0), the rest of the pairs in the first row, and the remaining pairs:

(S_0, T_0)	(S_0, T_1)	$(S_0, T_2)\ldots$
	(S_1, T_1)	$(S_1, T_2)\ldots$
		$(S_2, T_2)\ldots$

- The third piece in this decomposition (pairs that are not in the first row) is (recursively) the pairs formed from `(stream-cdr S)` and `(stream-cdr T)`.
- Also note that the second piece (the rest of the first row) is

```
(stream-map (lambda (x) (list (stream-car s) x))
            (stream-cdr t))
```

Streams of Pairs

- Thus we can form our stream of pairs as follows:

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (<combine-in-some-way>
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
       (pairs (stream-cdr s) (stream-cdr t)))))
```

- We must find a way to combine the two inner streams.
- One idea is to use the stream analog of the list append:

```
(define (stream-append s1 s2)
  (if (stream-null? s1) s2
    (cons-stream (stream-car s1)
                 (stream-append (stream-cdr s1) s2))))
```

Streams of Infinite Pairs

- This is unsuitable for infinite streams because it takes all the elements from the first stream before incorporating the second stream.
- In particular, if we try to generate all pairs of positive integers using

  ```
  (pairs integers integers)
  ```

  our stream will first run through all pairs with the first integer equal to 1, and will never produce pairs with any other value of the first integer.
- We need to devise an order of combination that ensures that every element will eventually be reached if we let our program run long enough.

Interleave

```
(define (interleave s1 s2)
  (if (stream-null? s1) s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

- Interleave takes elements alternately from the two streams.
- Every element of the second stream will eventually find its way into the interleaved stream, even if the first stream is infinite.
- We can thus generate the required stream of pairs as

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```