CSCI 275: Programming Abstractions

Lecture 23: Streams (cont.)

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Reminder: Streams

Data structure that is going to allow us to *write* sequential code, but have the structure evaluated *incrementally*

What do we need to do this?

1. A new data structure
2. An ability to control when terms are evaluated

Reminder: Better Evaluation in Built-in Racket

`(delay exp)` returns an object called a *promise*, without evaluating `exp`

`(force promise)` evaluates the promised expression and returns its value

- If the promise's `exp` has not been evaluated yet, it is evaluated and cached; otherwise, the cached value is returned
- A promised expression is evaluated only once, no matter how many times it is forced!

Promises in action!

This worked, but it was a bit annoying if we wanted to process the whole list!

```
> (define prime-lst (primes))
> prime-lst
'(2 . #<promise>)
> (force (cdr prime-lst))
'(3 . #<promise>)
> (force (cdr (force (cdr prime-lst))))
'(5 . #<promise>)
> prime-lst
'(2 . #<promise!(3 . #<promise!(5 . #<promise>)>)>)
```

Available Stream Procedures

These are already built-in, so we don't need to write them!

```
(require racket/stream)
(stream exp ...) ; Works like (list exp ...)
(stream? v)
(stream-cons head tail)
(stream-first s)
(stream-rest s)
(stream-empty? s)
empty-stream
(stream-ref s idx)
```

And several others

Constructing an Infinite Length Stream

Write a procedure which

- returns a stream constructed via `stream-cons`
- where the tail of the stream is a recursive call to the procedure

Call the procedure with the initial argument

```
(define (integers-from n)
  (stream-cons n (integers-from (add1 n))))

(define positive-integers (integers-from 0))
```

Constructing an infinite-length stream

Simplest infinite-length stream: A stream of all zeros

```
(define all-zeros
  (stream-cons 0 all-zeros))
```

Note: we cannot do this with a list!

```
(define all-zeros-lst
  (cons 0 all-zeros-lst))
```

```
Error: all-zeros-lst: undefined;
       cannot reference an identifier before its definition
```

Fibonacci numbers as a stream

Recall the Fibonacci numbers are defined by

$f_0 = 0$, $f_1 = 1$ and $f_n = f_{n-1} + f_{n-2}$

```
(define (next-fib m n)
  (stream-cons m (next-fib n (+ m n))))

(define fibs (next-fib 0 1))
```

Building streams from streams

How to write a procedure that adds two streams together

- Use `stream-cons` to construct the new stream
- Use `stream-first` on each stream to get the heads
- Recurse on the tails via `stream-rest`

```
(define (stream-add s t)
  (cond [(stream-empty? s) empty-stream]
        [(stream-empty? t) empty-stream]
        [else
         (stream-cons (+ (stream-first s)
                         (stream-first t))
                      (stream-add (stream-rest s)
                                  (stream-rest t)))]))
```

A helpful procedure for testing

We want to be able to look at the first *n* elements of a stream to be able to test whether it worked or not.
We don’t want to have to write `(stream-rest (stream-rest … )))`
`stream-take` lets us see the first *n* elements of a stream

```
(stream->list (stream-take fibs 10))
```

gives

```
‘(0 1 1 2 3 5 8 13 21 34)
```

Let's (all) write some Racket!

Open up a new file in DrRacket

Make sure the top of the file contains

```
#lang racket
(require racket/stream)
```

Available Stream Procedures

These are already built-in, so we don't need to write them!

```
 (require racket/stream)
 (stream exp ...) ; Works like (list exp ...)
 (stream? v)
 (stream-cons head tail)
 (stream-first s)
 (stream-rest s)
 (stream-empty? s)
empty-stream
 (stream-ref s idx)
```

And several others

Write some infinite-length streams

```
(constant-stream x)
```

Returns a stream containing an infinite number of x

```
(stream->list (stream-take (constant-stream 'ha) 10))
=> '(ha ha ha ha ha ha ha ha ha ha)
```

```
(stream-cycle s)
```

Returns an infinite-length stream consisting of the elements of s repeating in order.

```
(stream->list (stream-take
         (stream-cycle (stream ‘A ‘B ‘C)) 10))
=> '(A B C A B C A B C A)
```

Stream Procedures

Implement `(stream-filter f s)` which returns a stream containing the elements of `s` (in order) such that applying `f` to the element returns anything other than `#f`

Hint: Think about how you'd implement the filter function for lists using basic recursion with empty?, empty, cons, first, and rest

Bonus: You can prevent your implementation from evaluating f on elements of the stream **at the time you call stream-filter** by wrapping your implementation in a call to stream-lazy

Write some more stream procedures

```
(stream-double s)
```

Returns a stream containing each element of s twice

```
(stream-double (stream 1 2 3)) =>
 (stream 1 1 2 2 3 3)
```

```
(stream-interleave s t)
```

Returns a stream that interleaves elements of s and t

```
(stream-interleave (stream 1 2 3) '(a b c d))
=> (stream 1 'a 2 'b 3 'c 'd)
```