CSCI 275: Programming Abstractions

Lecture 22: Streams
Fall 2024

Stephen Checkoway
Slides from Cynthia Taylor

A Step Back from MiniScheme

Homeworks 5, 6 and 8 are MiniScheme

Homework 5: Environments, A, B

Homework 6: C, D, E

Through let, which we covered Monday

Homework 8: F, G, H

Interlude Today & Friday: Streams

```
(define (foo x)
  (display x)
  (display "\n")
  (cons x '(10)))
```

Note: helpful for MiniScheme debugging, display different values in `parse` or `eval-exp`

What value of `x` gets displayed?

```
(foo (list (+ 1 2) (+ 4 5)))
```

A. `((+ 1 2) (+ 4 5))`

B. `(list (+ 1 2) (+ 4 5))`

C. `(3 9)`

D. Something else

Racket has *eager* evaluation

Remember how function calls are evaluated

```
(my-func (list x y (+ x y 32))
         (if (> c 0) x y))
```

`my-func` is evaluated to a procedure

Then, the arguments are evaluated to values

Finally, the procedure's body is evaluated with the parameters bound to argument values

Creating an infinite list

Consider

```
(define (make-list start)
   (cons (start (make-list (add1 start)))))
```

The intention is `(make-list 0)` makes the infinite list `'(0 1 2 3 …)`

Why doesn't this work?

Lazy evaluation

What we want is *lazy* evaluation where expressions aren’t evaluated until they’re needed

Haskell has this behavior by default (Haskell is so cool)

In Racket, we need a new approach

Control Evaluation: Promises

Some new Scheme special forms!

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

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

One Set of Implementations

```
(define (delay exp)
  (lambda ()
    exp))
```

“Thunk”ing
is delaying the evaluation until later, here we wrap it in a no-argument lambda

THIS DOESN'T QUITE WORK! WHY?

```
(define (force promise)
  (promise))
```

How to call a no-argument lambda

Promises in Racket

We're going to use Racket's promises rather than our own

`(require racket/promise)` — Loads the library

`(delay body ...+)` — Returns a promise that when forced for the first time evaluates the body expressions

When subsequently forced, it returns the original value forced

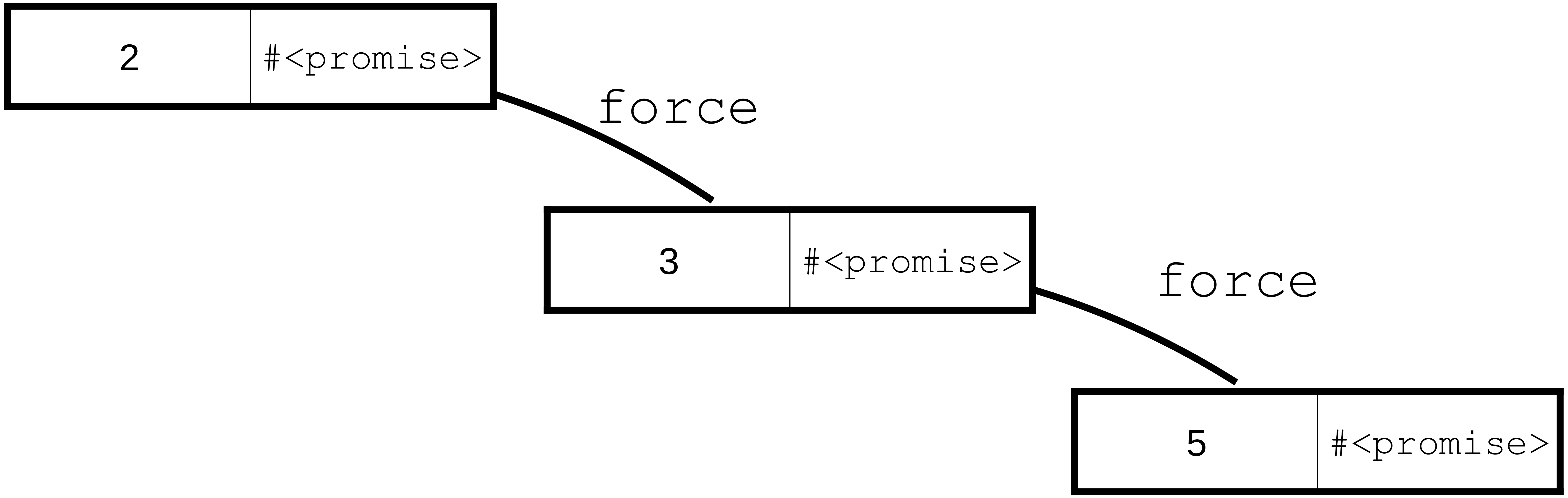
`(force promise)` — Force the promise

Let's build an infinite list of prime numbers

First, we need to think about how we want to represent this

Let's use a `cons` cell where

- the `car` is a prime; and
- the `cdr` is a promise which will return the next `cons` cell



Given prime?， Let's make a prime generator

`next-prime` checks if *n* is prime and if so, returns a `cons` cell containing n and a promise to construct the next one; otherwise it recurses on *n+2*

```
(define (next-prime n)
  (cond [(prime? n) (cons n
                          (delay (next-prime (+ n 2))))]
        [else (next-prime (+ n 2))]))
```

`primes` returns a cons cell containing 2 and a promise to construct the next one

```
(define primes
  (cons 2
        (delay (next-prime 3))))
```

```
(define primes
  (cons 2
        (delay (next-prime 3))))
```

and let `(define prime-lst (primes)).`

What is `(force (cdr prime-lst))?`

A. `'(3 #<promise>)`

B. `'(3 . #<promise>)`

C. `'(3 5 7 11 13 #<promise>)`

D. Something else

Infinite list in action!

We need cdr here, not rest, as a promise of a list is not a list itself

```
> (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>)>)>)
```

Introducing streams

A stream is a (potentially infinite) data structure

It contains a promise to return the first element in the stream and a promise to get the rest of the stream

We could build this out of Racket’s delay/force or…

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
```

Why does

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

work when the list-version does not?

A. Streams are magic

B. Streams are lazy so the stream-cons doesn't run until all-zeros is accessed for the first time

C. Streams are lazy so although the stream is constructed by `stream-cons`, its "first" and "rest" part aren't evaluated until forced by `stream-first` and `stream-rest`

D. Racket treats streams specially so it knows this construction is okay

Fibonacci numbers as a stream

Recall the Fibonacci numbers are defined by

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

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

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

Let's write some Racket!

Open up a new file in DrRacket

Make sure the top of the file contains

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

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)
```

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)))]))
```

Write some infinite-length streams

```
(require racket/stream)
```

```
(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 stream 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)
```

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