

Programming Abstractions

Lecture 31: Streams

Stephen Checkoway

Review of delay and force

`(delay exp)` creates a *promise* which when forced evaluates `exp` and returns the value

`(force p)` forces the promise `p` to obtain a value; if the promise's `exp` has not been evaluated yet, it is evaluated and cached; otherwise the cached value is returned

What is printed by this code?

```
(let* ([x 10]  
      [y (delay x)])  
  (set! x 20)  
  (displayln (force y)))
```

- A. 10
- B. 20
- C. It's an error

What is printed by this code?

```
(let* ([x 10]
       [y (delay x)])
  (set! x 20)
  (displayln (force y))
  (set! x 30)
  (displayln (force y)))
```

A. 20
20

B. 20
30

C. 30
30

D. It's an error

Concurrent execution

```
(require racket/promise)

(displayln "Before")

(define p (delay/thread
            (sleep 5)
            (displayln "Done!")
            42))

(displayln "During computation")
(force p)
(displayln "After")
```

What is the most likely output of

```
(define p1 (delay (println "Hello!")))
(define p2 (delay/thread (println "Goodbye!")))
(sleep 1) ; Wait one second
(force p1)
(force p2)
```

A. Hello!
Goodbye!
Hello!
Goodbye!

B. Hello!
Goodbye!

C. Goodbye!
Hello!
Hello!
Goodbye!

D. Goodbye!
Hello!

Promises in other languages

JavaScript has `async` which starts some potentially long-running calculation or (more typically) starts loading a resource from the Internet and returns a promise

This is paired with `await` which waits for the promise to finish computing/resource to download and returns the answer

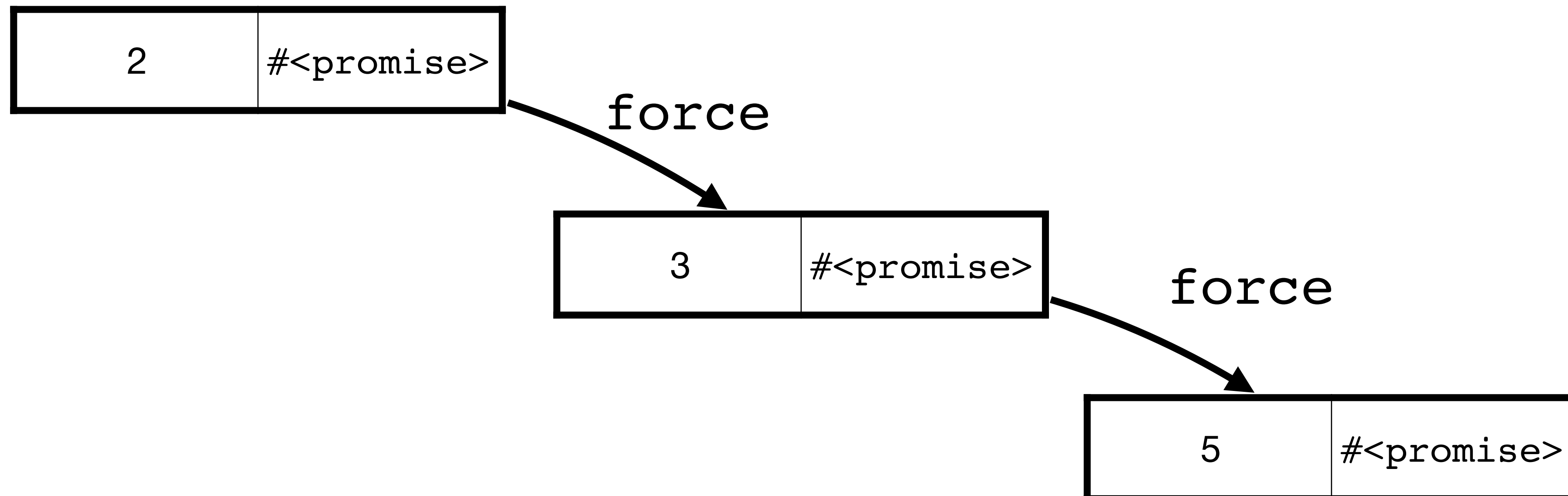
Rust has something similar

Last time: infinite list of primes

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



An infinite list is an instance of a stream

A stream is a (possibly infinite) sequence of elements

A list is a valid, finite stream

▸ `(stream? '(1 2 3)) => #t`

Infinite streams must be built lazily out of promises (using `delay` internally)

Accessing elements of a stream forces their evaluation

Let's build a stream

As with our infinite list of primes we'll use a cons-cell holding a value and a promise

API

- `(stream-cons head tail)`
- `(stream-first s)`
- `(stream-rest s)`
- `(stream-empty? s)`
- `empty-stream`

Constructing a lazy stream

```
(stream-cons head tail)
```

We can't use a procedure because it'll evaluate head and tail

```
(define-syntax stream-cons  
  (syntax-rules ()  
    [(_ head tail) (delay (cons head (delay tail)))]))
```

stream-cons returns a promise which when forced gives a cons cell where the second element is a promise

Accessing the stream

```
(stream-first s)  (stream-rest s)
```

s is either a promise or a cons cell so we need to check which

```
(define (stream-first s)
  (if (promise? s)
      (stream-first (force s))
      (car s)))
```

```
(define (stream-rest s)
  (if (promise? s)
      (stream-rest (force s))
      (cdr s)))
```

We can't use `first` and `rest` because those check if their arguments are lists

Checking if a stream is empty

```
(define empty-stream null)
(define (stream-empty? s)
  (if (promise? s)
      (stream-empty? (force s))
      (null? s)))
```

Constructing an infinite 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))
```

Accessing the elements

We can use `stream-first` and `stream-rest` to iterate through the elements

```
(define (stream-ref s idx)
  (cond [(zero? idx) (stream-first s)]
        [else (stream-ref (stream-rest s) (sub1 idx))]))
```

Primes as a stream

```
(define (prime? n) ...) ; Same as last time
```

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

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


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))
```


Fibonacci numbers as a stream: take 2

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

We can build our Fibonacci sequence directly from that definition (this is silly)

```
(define fibs
  (stream-cons
    0
    (stream-cons
      1
      (stream-add fibs (stream-rest fibs)))))
```

Streams in Racket

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

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)
```

Write the procedure `(stream-length s)` which returns the length of a finite stream

i.e., `(stream-length (stream 1 2 3 4 5))` returns 5

Use `stream-empty?` and `stream-rest`

Write more stream procedures

Write the procedure `(stream->list s)` that takes a finite-length stream and returns the elements as a list

Write the following procedures that act like their list counterparts, but operate lazily on streams; in particular, do not convert them to lists!

- `(stream-take s num)`
Returns a stream containing the first `num` elements of `s`, make sure this is lazy
- `(stream-drop s num)`
Returns a stream containing all of the elements of `s` in order *except* for the first `num`
- `(stream-filter f s)`
Returns a stream containing the elements `x` of `s` for which `(f x)` returns true
- `(stream-map f s)`
Returns a stream by mapping `f` over each element of `s`

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)`

Multi-argument stream-map

```
(stream-map f s ...)
```

Racket has stream-map built-in but unlike its list counterparts, it only takes a single stream

Generalize it to take any number of streams where the length of the returned string is the minimum length of any of the stream arguments (i.e., return empty-stream if any of the streams becomes empty); you'll want to use ormap, map and apply

```
▸ (define (stream-map f . ss) ...)
```