Programming Abstractions

Lecture 31: Streams 1

Stephen Checkoway

Announcements

Last homework is due on **Wednesday**, May 25 at 23:59

Final exam is **optional**

- You can take the final exam which will be similar to the midterms but without extra credit; or
- You can take the average (arithmetic mean) score of exams 1 and 2 with a maximum of 100%
- Either way, the final cannot push you over 100% in the course
- All exams contribute the same amount to your final grade

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

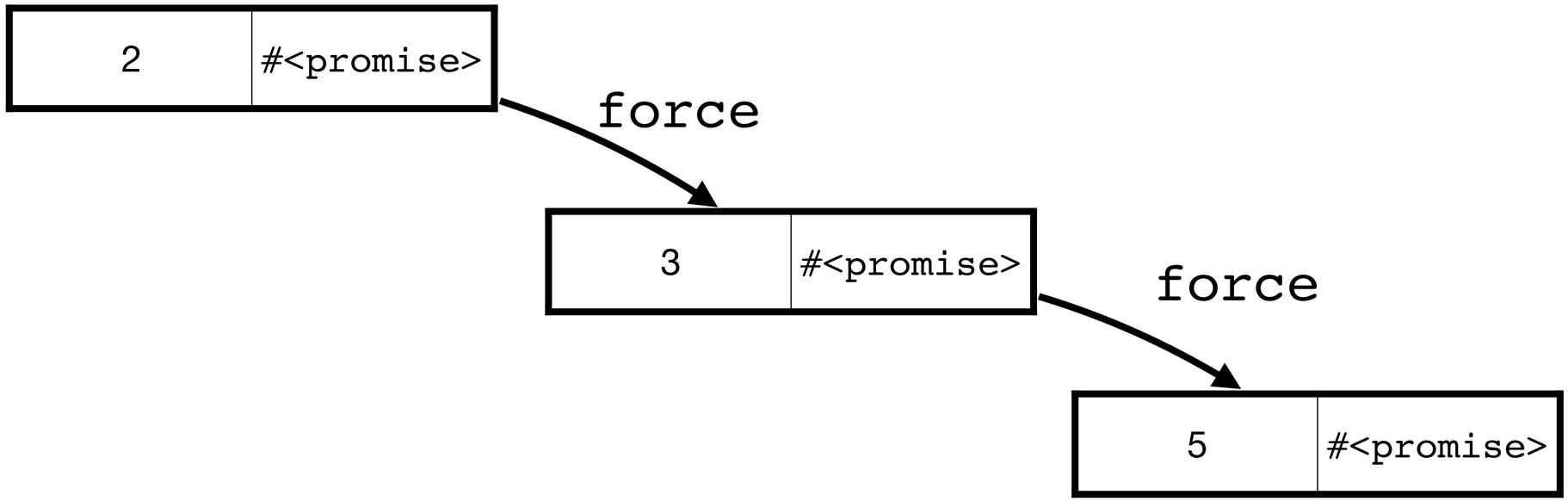
D. It's an error

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)))
```

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))]))
```

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!

Racket standard library function stream->list converts a finite-length!! stream to a list

- `(stream->list (stream 1 5 3 2 8)) => '(1 5 3 2 8)`

Implement this function in DrRacket using `stream-empty?`, `stream-first`, and `stream-rest`

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

(define (stream->list s)
  ...)
```

From lists to streams

Going from lists to streams is easy: Racket considers a list to be a stream

```
> (stream? '(1 2 3))
#t
```

Mapping over and filtering streams

Implement the function `(stream-map f s)` which takes a function `f` and a stream `s` and returns a new stream where f has been applied to each element of `s` in order

- This must be lazy (so no converting to a list and then using `map`)
- Think about how you would implement `(map f lst)` and follow the same approach but use `stream-cons`, `stream-first`, `stream-rest`, and `stream-empty?` rather than `cons`, `first`, `rest`, and `empty?`

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`

Next time

Infinite-length streams!