

Programming Abstractions

Lecture 30: Promises

Stephen Checkoway

Announcements

Homework 7 due on Friday

Office hours Tuesday 13:30–14:30

Homework 8 due on the last day of class, May 25

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

- A promised expression is evaluated only once, no matter how many times it is forced!

What does this code print?

```
(let* ([x 10]
      [f (λ () (add1 (* 3 x)))])
  [p (delay (add1 (* 3 x)))]
  (printf "(force p)=~s (f)=~s\n" (force p) (f)))
(set! x 5)
(printf "(force p)=~s (f)=~s\n" (force p) (f)))
```

A. (force p)=31 (f)=31
(force p)=31 (f)=16

B. (force p)=31 (f)=31
(force p)=16 (f)=16

C. (force p)=31 (f)=31
(force p)=16 (f)=31

D. (force p)=31 (f)=31
(force p)=31 (f)=31

What happens if we comment out the first printf?

```
(let* ([x 10]
      [f (λ () (add1 (* 3 x)))])
  [p (delay (add1 (* 3 x)))])
; (printf "(force p)=~s (f)=~s\n" (force p) (f))
(set! x 5)
(printf "(force p)=~s (f)=~s\n" (force p) (f)))
```

- A. (force p)=31 (f)=16
- B. (force p)=16 (f)=16
- C. (force p)=16 (f)=31
- D. (force p)=31 (f)=31

Example

```
(define foo  
  (delay  
    (begin  
      (displayln "Promise is evaluated")  
      2) ) )
```

```
(force foo) ; prints "Promise is evaluated"; returns 2  
(force foo) ; returns 2  
(force foo) ; returns 2
```

Example

```
(define foo
  (delay
    (begin
      (displayln "Promise is evaluated")
      2)))
```

begin not needed in Racket
delay allows arbitrary number
of expressions

```
(force foo) ; prints "Promise is evaluated"; returns 2
(force foo) ; returns 2
(force foo) ; returns 2
```

Implementing delay and force

Before we talk about *why* we might want this, let's talk about how we can implement it

First attempt: define delay as a procedure that returns a procedure

```
(define (delay exp)
  (λ ()
    exp))
```

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


What goes wrong with this definition?

```
(define (delay exp)
  (λ ()
    exp))
```

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

- A. exp is evaluated each time it is forced
- B. exp is evaluated before it is forced
- C. It **is not** possible to change the value in the promise after it has been forced
- D. It **is** possible to change the value in the promise after it has been forced

Evaluation isn't delayed

```
(delay  
  (displayln "Lazy evaluation would be nice"))
```

Since `delay` was implemented as a procedure, its argument is evaluated when `delay` is called

`force` will correctly return the value, but it was already computed; we need to delay the computation until `force` is called

We need a macro!

Let's think about what we want

We want

```
(delay exp)  
to become something like  
( $\lambda$  () exp)
```

Second attempt: define delay as a macro which produces a λ

```
(define-syntax delay  
  (syntax-rules ()  
    [(_ exp) ( $\lambda$  () exp)]))
```

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

Example

```
(define foo
  (delay
    (begin
      (displayln "This time, it's lazy!")
      10)))
```

This successfully defines foo as

```
(λ ()
  (begin
    (displayln "This time, it's lazy!")
    10))
```

and it doesn't evaluate until (force foo)

What goes wrong with this definition?

```
(define-syntax delay
  (syntax-rules ()
    [(_ exp) (λ () exp)]))
```

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

- A. `exp` is evaluated each time it is forced
- B. `exp` is evaluated before it is forced
- C. The result of `(delay exp)` is a macro which cannot be returned from—
or passed to—a procedure
- D. `force` needs to be a macro, not a procedure

Each time we force the promise, it's evaluated

```
(force foo) ; prints "This time it's lazy"; returns 10  
(force foo) ; prints "This time it's lazy"; returns 10  
(force foo) ; prints "This time it's lazy"; returns 10
```

We're going to need some mutation

We need to remember two things

- Has the promise been forced yet?
- If so, what was the value?

Mutation is difficult to reason about so it's best to keep the mutation as localized as possible

- We want to avoid global, mutable variables
- We want to avoid any other code even having access to the changing variable

What we really want

We want

`(delay exp)`

to become something like

```
(let ([evaluated #f]
      [value 0])
  (λ ()
    (if evaluated
        value
        (begin
          (set! value exp)
          (set! evaluated #t)
          value))))
```

When the result is forced (i.e., called) the first time

- `exp` will be evaluated
- `value` will be set to the result
- `evaluated` will be set to `#t`
- `value` is returned

On subsequent calls

- `value` is returned

When would we use promises?

We can build an infinite data structure like an infinite list, tree, or graph

- An infinite list of primes
- The Fibonacci sequence

Concurrent execution

- Creating the promise starts a *thread* that performs the computation
- Forcing the promise causes the current thread to wait until the computing thread has finished before returning the answer

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 evaluates the body expressions

`(delay/thread body ...+)` — Starts evaluating the body expressions on another thread and returns a promise that when forced waits for the execution to complete and returns the value

`(force promise)` — Force the promise

Let's build an 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

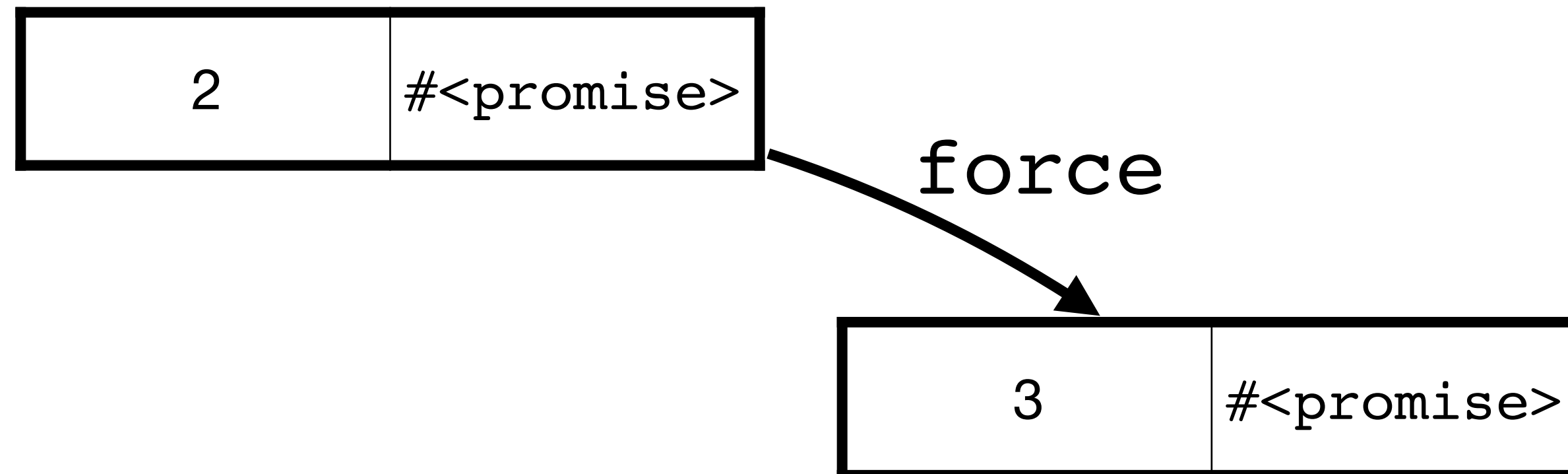
2	#<promise>
---	------------

Let's build an 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

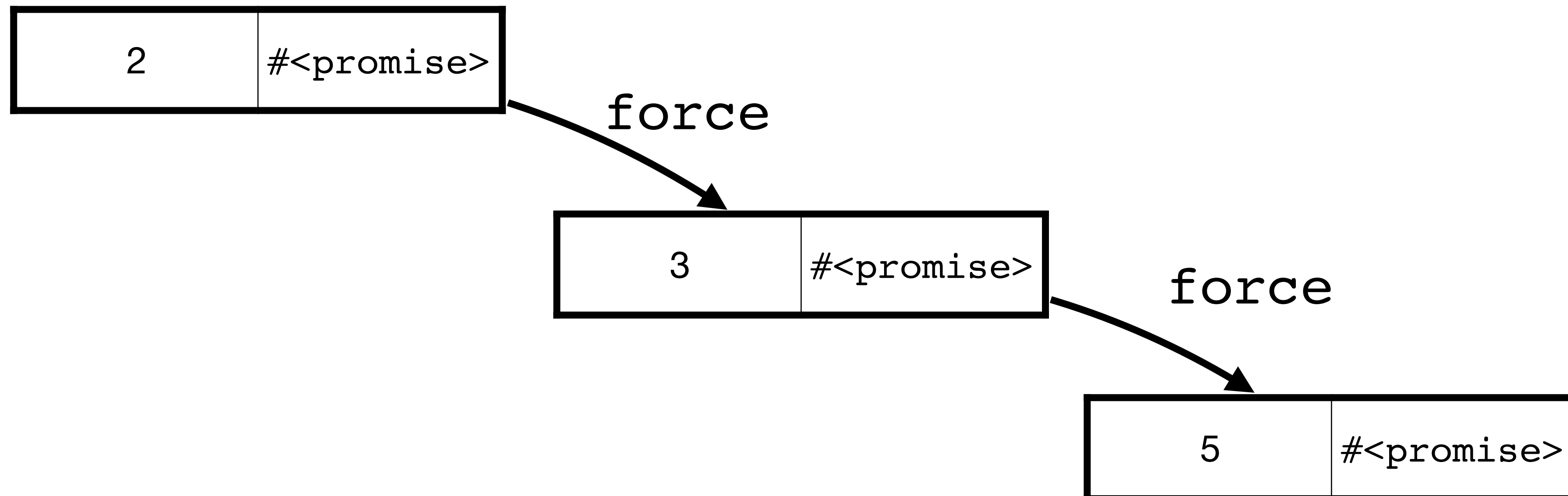


Let's build an infinite list of primes

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



The uninteresting piece: checking primality

```
(define (prime? n)
  (cond [(= n 2) #t]
        [(even? n) #f]
        [else (not
                 (ormap
                  (λ (m) (zero? (remainder n m)))
                  (range 3
                        (add1 (exact-floor (sqrt n)))
                        2)))]))
```

Does the simple thing and checks if dividing n by any odd m up to \sqrt{n} gives remainder 0. (Run time is exponential in the number of bits; we can do better, but it's complicated.)

The interesting piece: building the list

`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))))
```

Infinite list in action!

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


Using our list

```
(define (print-until n prime-lst)
  (let ([prime (car prime-lst)])
    (if (<= prime n)
        (begin
          (displayln prime)
          (print-until n (force (cdr prime-lst))))
        prime-lst))) ; Return the remainder of the list
```

Using our list

```
> (print-until 15 prime-lst)
```

2

3

5

7

11

13

```
'(17 . #<promise>)
```

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