CS450

Structure of Higher Level Languages

Lecture 15: Streams

Tiago Cogumbreiro

Today we will learn...

- streams
- functional patterns applied to streams
- compose stream operations

Streams

Stream

> A stream is an infinite sequence of values.

Did you know? The concept of streams is also used in:

- Reactive programming (eg, a stream of GUI events for Android development)
- Stream processing for digital signal processing (eg, image/video codecs with the language StreamIt)
- Unix pipes (eg, a pipeline of Unix process producing and consuming a stream of data)
- See also Microsoft LINQ and Java 8 streams

Streams in Racket

> A stream can be recursively defined as a a pair holds a value and another stream
> `stream = (cons some-value (thunk stream))`

Powers of two

```
(cons 1 (thunk (cons 2 (thunk (cons 4 (thunk ...))))))
```

Visually

```
1   2   4   ...
```

Using streams

```racket
(check-equal? 1 (car (powers-of-two)))                 ; the 1st element of the stream
(check-equal? 2 (car ((cdr (powers-of-two)))))         ; the 2nd element of the stream
(check-equal? 4 (car ((cdr ((cdr (powers-of-two))))))) ; the 3rd element of the stream
```

Revisiting our example with helper functions

```
; Retrieves the current value of the stream
(define (stream-get stream) (car stream))
; Retrieves the thunk and evaluates it, returning a thunk
(define (stream-next stream) ((cdr stream)))

(check-equal? 1 (stream-get (powers-of-two)))
(check-equal? 2 (stream-get (stream-next (powers-of-two))))
(check-equal? 4 (stream-get (stream-next (stream-next (powers-of-two)))))
```

Programming with streams

> Let us write a function that given a stream and a predicate, counts how many times a predicate holds true until it becomes false.

Spec

```
(check-equal? 3 (count-until (powers-of-two) (lambda (x) (< x 8))))
(check-equal? 0 (count-until (powers-of-two) (lambda (x) (≤ x 0))))
(check-equal? 3 (count-until (powers-of-two) (curryr < 8)))  ; Reverse Currying
(check-equal? 0 (count-until (powers-of-two) (curryr ≤ 0))) ; Reverse Currying
```

Programming with streams

> Let us write a function that given a stream and a predicate, counts how many times a predicate holds true until it becomes false.

Spec

```
(check-equal? 3 (count-until (powers-of-two) (lambda (x) (< x 8))))
(check-equal? 0 (count-until (powers-of-two) (lambda (x) (≤ x 0))))
(check-equal? 3 (count-until (powers-of-two) (curryr < 8)))  ; Reverse Currying
(check-equal? 0 (count-until (powers-of-two) (curryr ≤ 0))) ; Reverse Currying
```

Solution

```
(define (count-until stream pred)
  (define (count-until-iter s count)
    (cond [(pred (stream-get s)) (count-until-iter (stream-next s) (+ count 1))]
          [else count]))
  (count-until-iter stream 0))
```

Example: powers of two

- Implement the stream `powers-of-two`

Example: powers of two

> Implement the stream `powers-of-two`

Solution

```
(define (powers-of-two)
  (define (powers-of-two-iter n)
    (thunk
      (cons n (powers-of-two-iter (* 2 n)))))
  ((powers-of-two-iter 1)))
```

Example: constant

> Implement a function `const` that given a value it returns a stream that always yields that value.

```
(check-equal? 20 (stream-get (const 20)))
(check-equal? 20 (stream-get (stream-next (const 20))))
(check-equal? 20 (stream-get (stream-next (stream-next (const 20)))))
```

Example: constant

> Implement a function `const` that given a value it returns a stream that always yields that value.

```
(check-equal? 20 (stream-get (const 20)))
(check-equal? 20 (stream-get (stream-next (const 20))))
(check-equal? 20 (stream-get (stream-next (stream-next (const 20)))))
```

Solution

```
(define (const v)
  (define (const-iter) (cons v const-iter))
  (const-iter))
```

Common mistakes (1)

```
(define (const v)
  (define const-iter (cons v const-iter))
  (const-iter))
```

Common mistakes (1)

```
(define (const v)
  (define const-iter (cons v const-iter))
  (const-iter))
```

> `const-iter` is not a thunk. The error is that `const-iter` is not defined (as the body of the definition is evaluated).

Common mistakes (2)

```
(define (const v)
  (define (const-iter) (cons v (const-iter)))
  (const-iter))
```

Common mistakes (2)

```
(define (const v)
  (define (const-iter) (cons v (const-iter)))
  (const-iter))
```

> in the body of `const-iter` the thunk `const-iter` is evaluated. This function does not terminate.

Streams in Racket

> A stream can be recursively defined as a a pair holds a value and another stream
> `stream = (cons some-value (thunk stream))`

A stream of natural numbers

```
(cons 0 (thunk (cons 1 (thunk (cons 2 (thunk ...))))))
```

Visually

```
0  1   2   3   4   5   6   ...
```

Using streams

```
(check-equal? 0 (stream-get (naturals)))
(check-equal? 1 (stream-get (stream-next (naturals))))
(check-equal? 2 (stream-get (stream-next (stream-next (naturals)))))
```

Natural numbers

▌Implement the stream of non-negative integers

```
0 1 2 3 4 5 6 7 ...
```

Spec

```
#lang racket
(require rackunit)

(define s0 (naturals))
(check-equal? 0 (stream-get s0))

(define s1 (stream-next s0))
(check-equal? 1 (stream-get s1))

(define s2 (stream-next s1))
(check-equal? 2 (stream-get s2))
```

Natural numbers

- Implement the stream of non-negative integers

```
0 1 2 3 4 5 6 7 ...
```

Spec

```
#lang racket
(require rackunit)

(define s0 (naturals))
(check-equal? 0 (stream-get s0))

(define s1 (stream-next s0))
(check-equal? 1 (stream-get s1))

(define s2 (stream-next s1))
(check-equal? 2 (stream-get s2))
```

Solution

```
(define (naturals)
  (define (naturals-iter n)
    (thunk
      (cons n (naturals-iter (+ n 1)))))
  ((naturals-iter 0)))
```

Map for streams

- Given a stream `s` defined as

```
e0 e1 e2 e3 e4 ...
```

- and a function `f` the stream `(stream-map f s)` should yield

```
(f e0) (f e1) (f e2) (f e3) (f e4) ...
```

Map for streams

Spec

```
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals)))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Map for streams

Spec

```
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals)))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```
(define (stream-map f s)
  (define (stream-map-iter s)
    (thunk
      (cons
        (f (stream-get s))
        (stream-map-iter (stream-next s)))))
  ((stream-map-iter s)))
```

Even naturals

> Build a stream of even numbers. Tip: use `stream-map` and `naturals`.

```
0 2 4 6 8 10 12 ...
```

Spec

```
#lang racket
(require rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Even naturals

Build a stream of even numbers. Tip: use `stream-map` and `naturals`.

```
0 2 4 6 8 10 12 ...
```

Spec

```
#lang racket
(require rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```
(define (even-naturals)
  (stream-map
    (curry * 2)
    (naturals)))
```

Zip two streams

- Given a stream `s1` defined as

```
e1 e2 e3 e4 ...
```

- and a stream `s2` defined as

```
f1 f2 f3 f4 ...
```

- the stream `(stream-zip s1 s2)` returns

```
(cons e1 f1) (cons e2 f2) (cons e3 f3) (cons e4 f4) ...
```

Zip for streams

Spec

```racket
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))

(check-equal? (stream-get s0) (cons 0 0))

(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))

(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```

Zip for streams

Spec

```
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))

(check-equal? (stream-get s0) (cons 0 0))

(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))

(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```

Solution

```
(define (stream-zip s1 s2)
  (define (stream-zip-iter s1 s2)
    (thunk
      (cons
        (cons (stream-get s1)
              (stream-get s2))
        (stream-zip-iter
          (stream-next s1)
          (stream-next s2)))))
  ((stream-zip-iter s1 s2)))
```