CS450

Structure of Higher Level Languages

Lecture 08: Streams

Tiago Cogumbreiro

Today we will...

1. Program using streams
2. Revisit functional patterns applied to streams
3. Compose stream operations using functional patterns

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

```
#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)))
```

Enumerate a stream

Build a stream from a given stream `s` defined as

```
e0  e1  e2  e3  e4  e5 ...
```

the stream `(stream-enum s)` returns

```
(cons 0 e0)  (cons 1 e1)  (cons 2 e2)  (cons 3 e3)  (cons 4 e4)  (cons 5 e5) ...
```

Enumerate a stream

Spec

```
#lang racket
(require rackunit)

(define s0 (stream-enum (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))
```

Enumerate a stream

Spec

```
#lang racket
(require rackunit)

(define s0 (stream-enum (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-enum s)
  (stream-zip (naturals) s))
```

Filter

How would a filter work with streams?

Filter

Spec

```
#lang racket
(define s0
  (stream-filter (curry ≤ 10)
    (naturals)))
(check-equal? (stream-get s0) 10)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 11)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 12)
```

Filter

Spec

```
#lang racket
(define s0
  (stream-filter (curry ≤ 10)
    (naturals)))
(check-equal? (stream-get s0) 10)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 11)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 12)
```

Solution

```
(define (stream-filter pred s)
  (define (stream-filter-iter s)
    (thunk
      ; Get the head of the stream
      (define h (stream-get s))
      ; Filter the rest of the stream
      (define next
        (stream-filter-iter
          (stream-next s)))
      ; Predicate holds, then return h
      (cond [(pred h) (cons h next)]
            ; Otherwise, unfold the stream
            [else (next)])))
  ((stream-filter-iter s)))
```

Drop every other element

> Given a stream defined below, drop every other element from the stream. That is, given a stream `s` defined as...

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

> stream `(stream-drop-1 s)` returns

```
e0 e2 e4 ...
```

Drop every other element...

Spec

```
#lang racket
(require rackunit)

(define s0 (stream-drop-1 (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)
```

Drop every other element...

Spec

```
#lang racket
(require rackunit)

(define s0 (stream-drop-1 (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 (stream-drop-1 s)
  ; for each e yield (i, e)
  (define enum-s (stream-enum s))
  ; given (i, e) only keep (even? i)
  (define even-s
    (stream-filter
      ;(lambda (x) (even? (car x)))
      (compose even? car)
      enum-s))
  ; convert (i, e) back to e
  (stream-map cdr even-s))
```

More exercises

- `(stream-ref s n)` returns the element in the `n`-th position of stream `s`
- `(stream-interleave s1 s2)` interleave each element of stream `s1` with each element of `s2`
- `(stream-merge f s1 s2)` for each `i`-th element of stream `s1` (say `e1`) and `i`-th element of stream `s2` (say `e2`) return `(f e1 e2)`
- `(stream-drop n s)` ignore the first `n` elements from stream `s`
- `(stream-take n s)` returns the first `n` elements of stream `s` in a list in appearance order