CS450

Structure of Higher Level Languages

Lecture 11: Infinite streams

Tiago Cogumbreiro

Press arrow keys ← → to change slides.

Infinite streams

Stream

> A stream is an infinite sequence of values.

For example, how would you represent the set of $\mathcal{N}$ natural numbers in a program?

```
1  → 2 → 3 → 4 → 5 → 6 → 7 → …
```

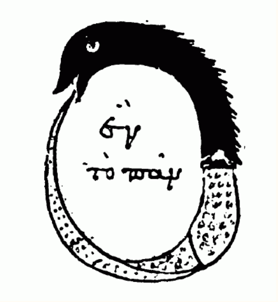
Did you know? The concept of streams is also used in:

- 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 TypedRacket

```
(define-type (stream T)
  ; A thunk that returns a stream-add
  (-> (stream-add T))
)
(struct [T] stream-add (
  ; Holds an element
  [first : T]
  ; And a thunk to another stream of T
  [rest : (stream T)]
  )
)
```

- A stream of type `T` is an infinite "linked-list" whose elements have type `T`
- We encode the notion of infinitude with **delayed-construction** of the elements on the list
- Here: stream is a thunk. that holds a `stream-add`, that holds a stream
- I like to imagine a stream as a data-structure equivalent of Ouroboros



Streams in (Typed)Racket

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

Powers of two

```
(thunk (stream-add 1 (thunk (stream-add 2 (thunk (stream-add 4 (thunk ...)))))))
```

Visually

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

Using streams

```
(match (powers-of-two) ; 1. A stream is a thunk; call to build the first element
  [(stream-add x s)    ; 2. Inside the thunk there is a stream-add
    (displayln x)      ; 3. Output the first element: 1
    (match (s)                ; 4. Construct the second element
      [(stream-add y s)
        (check-equal? y 2)    ; 5. Output the 2nd element: 2
        (match (s)            ; 6. Call and match the 3rd element
          [(stream-add z _)
            (check-equal z 4) ; 7. Output the 3rd element: 4
          ]
        )
      ]
    )
  ]
)
```

Generalize sampling the first 3 elements

```
(: sample3 (All [T] (-> (stream T) (Listof T))))
(define (sample3 s)
  (match (s)
    [(stream-add x1 s)
      (match (s)
        [(stream-add x2 s)
          (match (s)
            [(stream-add x3 _)
              (list x1 x2 x3)
            ]
          )
        ]
      )
    ]
  )
)
(sample3 powers-of-two)
; '(1 2 4)
```

Count elements in stream

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.

```
(: count-until
  (All [T]
    (->
      ; Given a predicate
      (-> T Boolean)
      ; Take a stream
      (stream T)
      ; How many elements
      Number
    )
  )
)
```

Spec

```
(count-until (lambda ([x : Number]) (< (cast x Real) 8)) po
; 3
(count-until (lambda ([x : Number]) (<= (cast x Real) 0)) p
; 0
```

Solution

```
(define (count-until pred s)
  (: count-until-iter (-> Number (stream T) Number))
  (define (count-until-iter count s)
    (match (s)
      ; If the predicate holds on x
      [(stream-add x s)
        (if (pred x)
          ; loop: increment count in next iter
          (count-until-iter (+ count 1) s)
          ; return count
          count
        )
      ]
    )
  )
  (count-until-iter 0 s))
```

- Make sure we write `match (s)` to let the stream build the first element
- Pattern-matching helps us retrieve the element built and the rest of the stream

```
def count_until(pred, s):
  count = 0
  while True:
    (x, s) = s() # similar to match
    if pred(x):
      count += 1
      # loop = rec call
    else:
      return count
```

Implementing powers of two

Example: powers of two

- Implement `(: powers-of-two (stream Number))`

Example: powers of two

- Implement `(: powers-of-two (stream Number))`

Solution

```
(: powers-of-two (stream Number))
(define (powers-of-two)
  (: powers-of-two-iter (-> Number (stream Number)))
  (define (powers-of-two-iter n)
    (thunk (stream-add n (powers-of-two-iter (* 2 n)))))
  ((powers-of-two-iter 1))
)
```

Common mistake: forgot thunk

```
(: powers-of-two (stream Number))
(define (powers-of-two1)
  (: powers-of-two-iter (-> Number (stream Number)))
  (define (powers-of-two-iter n)
     (stream-add n (powers-of-two-iter (* 2 n))))
  ((powers-of-two-iter 1))
)
```

If you were to run this code, it would run into an infinite loop, as **recursion is not guarded**!

Common mistake: forgot thunk

```
(: powers-of-two (stream Number))
(define (powers-of-two1)
  (: powers-of-two-iter (-> Number (stream Number)))
  (define (powers-of-two-iter n)
     (stream-add n (powers-of-two-iter (* 2 n))))
  ((powers-of-two-iter 1))
)
```

If you were to run this code, it would run into an infinite loop, as **recursion is not guarded**!

```
lecture11.rkt:56:4: Type Checker:
  Polymorphic function `stream-add'
  could not be applied to arguments:
Argument 1:
  Expected: T
  Given:    Number
Argument 2:
  Expected: (-> (stream-add T))
  Given:    (-> (stream-add Number))

Result type:     (stream-add T)
Expected result: (-> (stream-add Number))

  in: (stream-add n
         (powers-of-two-iter (* 2 n)))
```

The stream of constants

Example: constant

> Implement a function `const` that given a value it returns a stream that always yields that value: `(: const (-> Number (stream Number)))`

```
(sample3 (const 20))
; '(20 20 20)
```

Solution

```
(: const (-> Number (stream Number)))
(define (const v)
  (thunk (stream-add v (const v))))
```

The stream of natural numbers

Streams in Racket

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

A stream of natural numbers

```
(thunk (stream-add 0 (thunk (stream-add 1 (thunk (stream-add 2 (thunk ...)))))))
```

Visually

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

The type of naturals is a stream of numbers:

```
(: naturals (stream Number))
```

Natural numbers

> Implement the stream of non-negative integers

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

Solution

```
(: naturals (stream Number))
(define (naturals)
  (: naturals-iter (-> Number (stream Number)))
  (define (naturals-iter n)
    (thunk
      (stream-add n
        (naturals-iter (+ n 1)))))
  ((naturals-iter 0)))
```

The map stream

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

```
(: stream-map
  (All [InputElem OutputElem]
    (->
      ; Can convert an InputElem into an OutputElem
      (-> InputElem OutputElem)
      ; Given a stream of InputElem
      (stream InputElem)
      ; Generate a stream of OutputElem
      (stream OutputElem)
    )
  )
)
```

Map for streams

Spec

```
(: stream-map
  (All [InputElem OutputElem]
    (->
      ; Can convert an InputElem into an OutputElem
      (-> InputElem OutputElem)
      ; Given a stream of InputElem
      (stream InputElem)
      ; Generate a stream of OutputElem
      (stream OutputElem)
    )
  )
)
```

Solution

```
(define (stream-map f s)
 (thunk  ; <- very important!
  (match (s) ; <- delayed match
   [(stream-add x s)
    (stream-add
      (f x)   ; replace x by (f x)
      (stream-map f s)
    )
   ]
  )
 )
)
```

Eager-match makes can trigger errors

INCORRECT

```
(define (stream-map f s)
  (match (s) ; <- An INCORRECT eager match
   [(stream-add x s)
    (thunk (stream-add (f x) (stream-map f s))
   ])))
```

CORRECT

```
(define (stream-map f s)
 (thunk  ; <- thunk BEFORE match
  (match (s)
   [(stream-add x s)
    (stream-add (f x) (stream-map f s))])))
```

- An important point of streams is to delay execution as much as possible
- In this **incorrect** example we `match` ***before*** the `thunk`
- A user invoking `(stream-map f s)`, builds the first element of `s`, which is surprising and undesirable
- In the correctness tests of HW4, we will test for eagerness

The stream of even numbers

Even naturals

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

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

Even naturals

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

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

Solution

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

Merge two streams

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

Exercises on streams

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) ...
```

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

```racket
#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

```racket
#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

```racket
(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)
```

Converting filter to stream-filter

```
; List version ---------------------------------------------------------------
(define (filter to-keep? l)
  (cond
   [(empty? l) l]
   [(to-keep? (first l))
    (cons (first l)
          (filter to-keep? (rest l)))]
   [else (filter to-keep? (rest l))]))
; Stream-version -------------------------------------------------------------
(define (stream-filter to-keep? s)
  (cond
    ; <- no base case; streams are infinite
    [(to-keep? (stream-get s)) ; <- first becomes stream-get
     (cons (stream-get s)
           ; Second element is always a thunk
           (thunk (stream-filter to-keep? (stream-next s))))]
    [else (stream-filter to-keep? (stream-next s))])) ; rest becomes stream-next
```

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

```racket
#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

```racket
#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

```racket
(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