Stream Programming

FOCS, Fall 2020

Dataflow networks

Last time dataflow networks

- A way to program over streaming data

It's a graphical model — we'll see how to program graphical models next homework

Another way to program over streaming data is to consider a stream to be an infinite list and write list processing functions as usual

Infinite lists

A list in OCaml or most programming languages is a finite structure

You cannot construct an infinite list in Ocaml

- You'd spend all your time building it

Yet, some languages lets you define infinite lists, the equivalent of writing:

```
let rec from k = k :: (from (k + 1))
```

where `from 10` would be the list [10, 11, 12, 13, 14, 15, 16, …]

Lazy evaluation

One way to do it is to use lazy evaluation

- only evaluate an expression if you need its value

Consider the function

```
let test a b = a
```

It does not use the second argument at all

In a language with lazy evaluation, `test 10 (g 20)` returns `10` *even if* `(g 20)` *is an infinite loop*

Haskell

A functional programming language like OCaml but which uses lazy evaluation

- An expression is evaluated only if its value is needed

```
square x = x * x

length [] = 0
length (h : t) = 1 + length t

squares lst = map (\a -> a * a) lst
```

Lists in Haskell

Like Ocaml, if L is a list, `a : L` is a list with first element a and rest of the list L

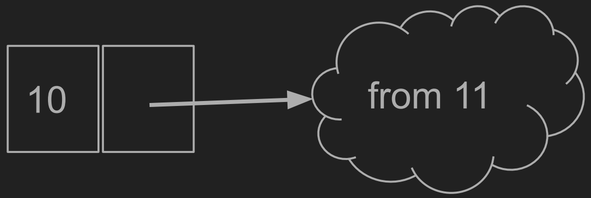
```
10 : [1, 2, 3]   →  [10, 1, 2, 3]
```

But `a : L` is lazy in its second argument

- it does not evaluate L until L is needed
  (e.g., somebody trying to access the elements of L)

With `from k = k : (from (k + 1))`

`from 10` is perfectly well defined



Lists in Haskell

Like Ocaml, if L is a list, `a : L` is a list with first element a and rest of the list L

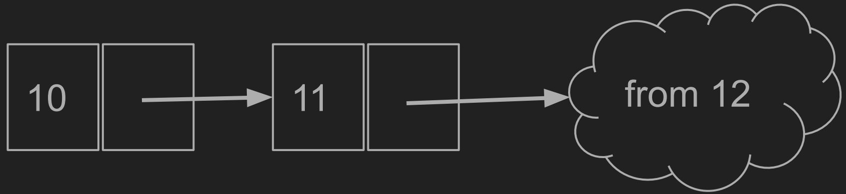
```
10 : [1, 2, 3]   →  [10, 1, 2, 3]
```

But `a : L` is lazy in its second argument

- it does not evaluate L until L is needed
  (e.g., somebody trying to access the elements of L)

With `from k = k : (from (k + 1))`

`from 10` is perfectly well defined



Primitive components

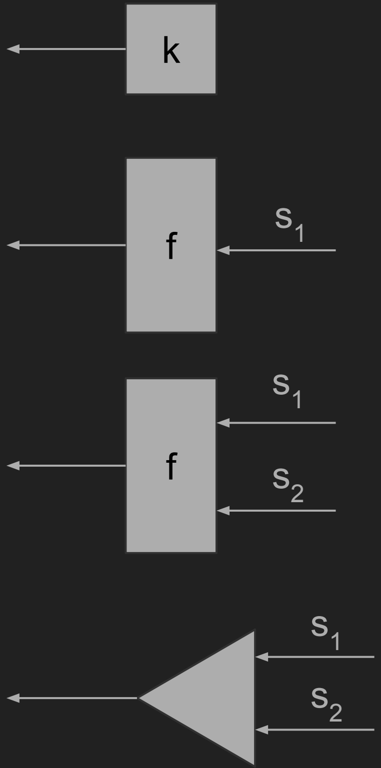
$\text{cst a} = \text{a} : (\text{cst a})$

$\text{map f} (h : t) = (f\ h) : (\text{map f t})$

$\text{map2 f} (h_1 : t_1)\ (h_2 : t_2) = (f\ h_1\ h_2) : (\text{map2 f}\ t_1\ t_2)$

$\text{fby} (h : t)\ s = h : s$

$\text{tail} (h : t) = t$



cst k

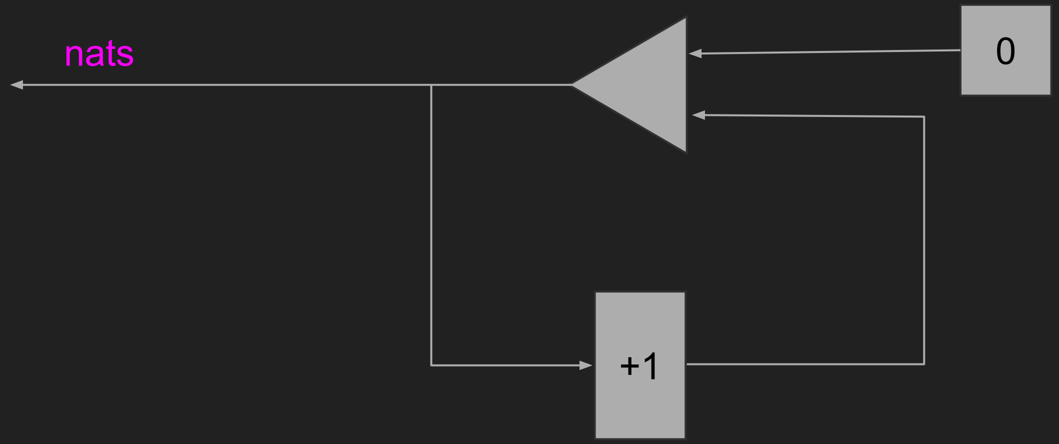
map f s_1

map2 f s_1 s_2

fby s_1 s_2

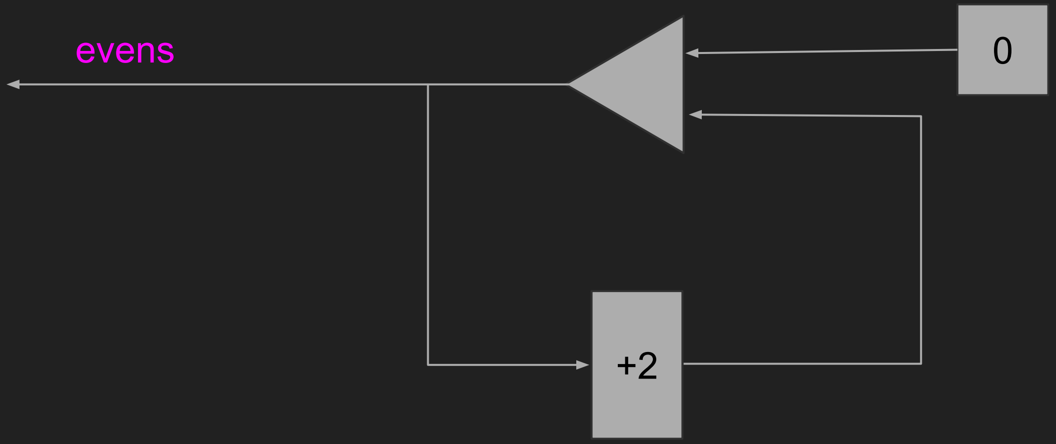
tail s_1

Example: nats



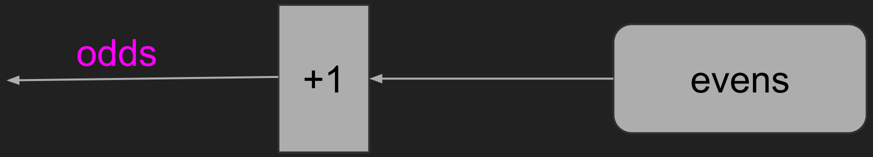

```
plus1 s = map (\a -> a + 1) s

nats   = fby (cst 0) (plus1 nats)
```

Example: evens



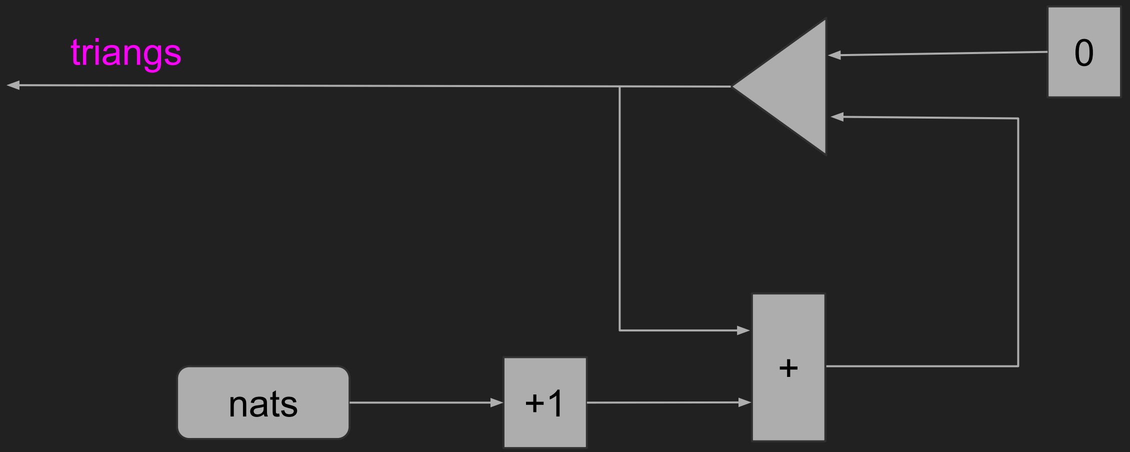
evens = fby (cst 0) (plus1 (plus 1 evens))

Example: odds



odds = plus1 evens

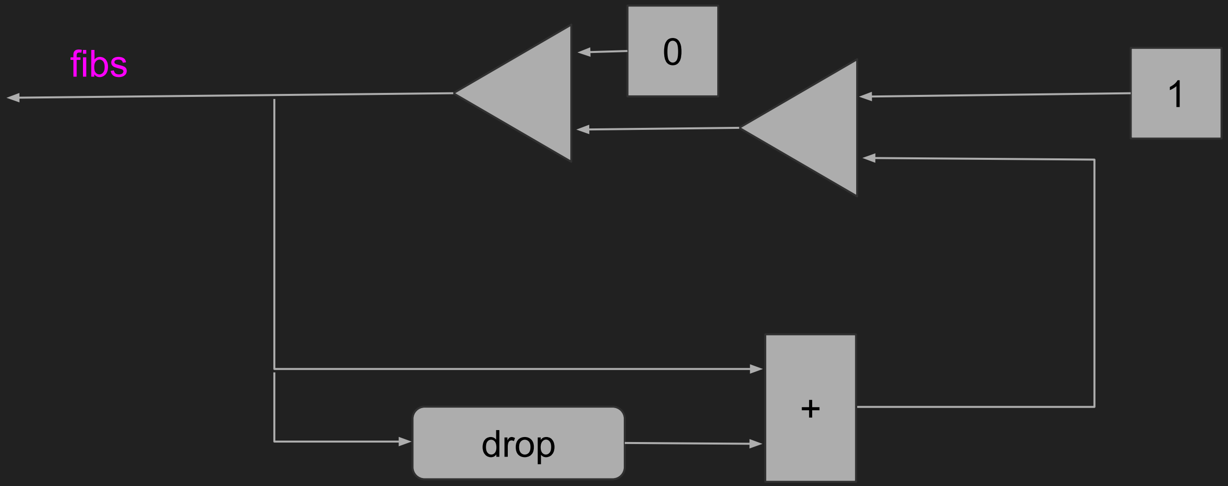
Example: triangular numbers



plus s_1 s_2 = map2 (\a -> \b -> a + b) s_1 s_2

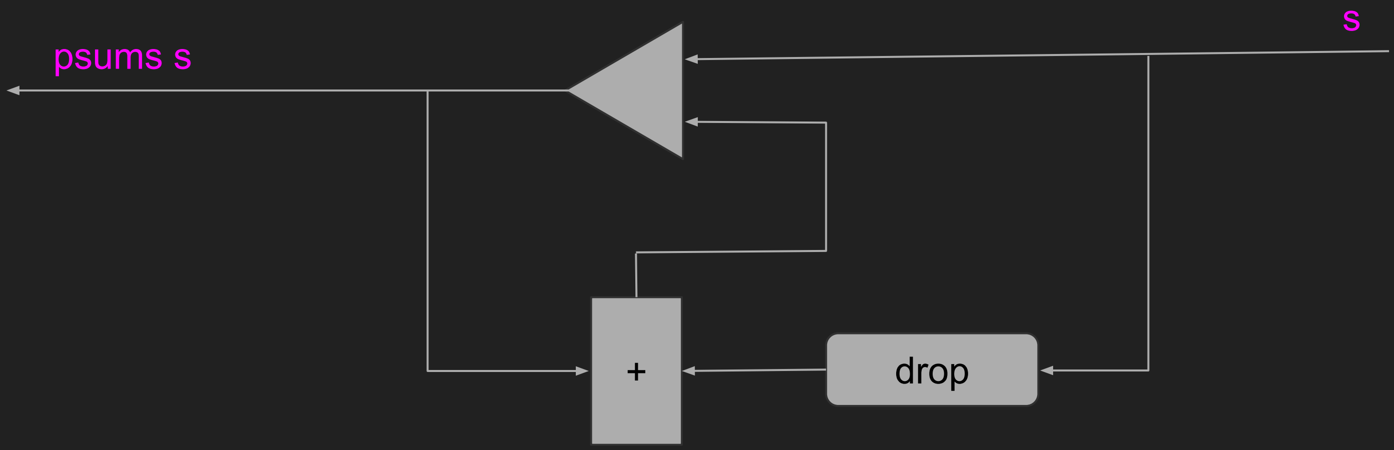
triangs = fby (cst 0) (plus triangs (plus1 nats))

Example: Fibonacci numbers



fibs = fby (cst 0) (fby (cst 1) (plus fibs (tail fibs)))

Example: partial sums



psums s = fby s (plus (psums s) (tail s))

Recursive dataflow networks

A dataflow network that uses a copy of itself as a component

Sieve of Eratosthenes - how to compute the stream of prime numbers:

- sieving a stream: take a stream of values, keep the first value, and sieve the rest of the stream *after* removing all multiples of the first value
- sieving the naturals numbers starting from 2 yields the prime numbers

```
divides c x = (mod x c == 0)
sieve s = fby s (sieve (filter (\a -> not (divides (head s) a)) (tail s)))

primes = sieve (plus1 (plus1 nats))
```

Infinite lists without lazy evaluation

Even if you do not have lazy evaluation, you can still program with infinite streams

- OCaml: a stream is a pair of an element and a thunk
  - a thunk is a function of no arguments that returns a stream
  - until the thunk is called, it's like an "unevaluated stream"

```
type 'a stream = St of 'a * (unit -> 'a stream)

let map (f: 'a -> 'b) (s: 'a stream): 'b stream =
   match s with (h, th) -> St (f h, fun () -> map f (th ()))

let nats (): int stream =
   St (0, fun () -> map (fun x -> x + 1) (nats ()))
```