CS 3110

Streams and Laziness

Nate Foster
Spring 2020

Today's scene: Gates Hall

Review

Previously:

- Promises
- Monads

Today:

- Streams
- Laziness

“Infinite” lists

How can an infinite length list fit in a finite computer memory?

aka infinite lists, sequences, delayed lists, lazy lists

STREAMS

List representation

```
(** An ['a mylist] is a finite
    list of values of type
    ['a]. *)
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
```

Stream representation?

```
(** An ['a stream] is an infinite
    list of values of type
    ['a]. *)
type 'a stream =
  | Nil
  | Cons of 'a * 'a stream
```

Stream representation?

```
(** An ['a stream] is an infinite
    list of values of type
    ['a]. *)
type 'a stream =
  ~~| Nil~~
  | Cons of 'a * 'a stream
```

Stream representation?

```
type 'a stream =
   | Cons of 'a * 'a stream
```

Let’s try coding these:

- the stream of 1's
- the stream of natural numbers

Key idea of this entire lecture:

Be lazy:

delay evaluation

Dem
o

thunk

```
fun () -> (* a delayed computation *)
```

Stream representation

```
(** An ['a stream] is an infinite list
      of values of type ['a].
    AF:  [Cons (x, f)] is the stream
      whose head is [x] and tail is
      [f()].
    RI:  none *)
type 'a stream =
  Cons of 'a * (unit -> 'a stream)
```

Dem
o

Notation

Write

`<a; b; c; …>`

to mean stream whose first elements are `a`, `b`, `c`.

Stream sum

```
(** [sum <a1; a2; ...> <b1; b2; ...>]
    is [<a1 + b1; a2 + b2; ...>] *)
let rec sum
  (Cons (h_a, tf_a))
  (Cons (h_b, tf_b))
=
  ?
```

Demo

LAZINESS

Dem
o

Lazy

- Syntax: **`lazy`** `e`

- Static semantics:
  if `e : t` then **`lazy`** `e : t lazy_t`

- Dynamic semantics:
  - **`lazy`** `e` evaluates to a *delayed value*
  - does not evaluate e to a value yet
  - when forced for the first time, evaluates `e` to a value `v`
  - if forced again, return `v` without evaluating `e` again

Lazy

Standard library module for

- delaying evaluation
- remembering results once computed

```
module Lazy :
  sig
    type 'a t = 'a lazy_t
    val force : 'a t -> 'a
  end
```

Type constructor [lazy_t] is built-in to language

Implementing Lazy

- **`force`**: can implement yourself with references
- **`lazy`**: can't implement yourself

Dem
o

Stream and laziness

```
type 'a stream =
  Cons of 'a * 'a stream Lazy.t
```

vs

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

Upcoming events

- [Monday] R8 Due
- [Friday] A5 Due

This is happily lazy.

THIS IS 3110