Streams and Laziness

Nate Foster
Spring 2019

Today’s music: "Lazy Days" by Shwayze
What is the type of \( f \)?

\[
\text{let rec } f \ x = f \ x
\]

A. Doesn't compile
B. 'a -> 'a
C. 'a -> 'b
D. unit -> unit
Review

Previously in 3110:
• Functional programming
• Modular programming

Third unit of course: Data structures

Today:
• Streams
• Laziness
INFINITE LISTS
Discussion

How can an infinite length list fit in a finite computer memory?
"Infinite" data structures

• Sequences of numbers: the naturals, primes, Fibonacci, ...

• Data processed by a program: from a file, from the user, from the network

• Game tree (for some games):
  – nodes = game positions
  – edges = legal moves
Question

What does nats evaluate to?

(* [from n] is the infinite list [[n; n+1; ...]] *)

```plaintext
let rec from n = n :: from (n+1)

let nats = from 0
```

A. [0; 1; 2; ...]
B. Never terminates (infinite loop)
C. Exception
D. Stack overflow
aka infinite lists, sequences, delayed lists, lazy lists
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

Try coding these if possible:
• the stream of 1's
• the stream of natural numbers
Key idea of this entire lecture:

Delay evaluation
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)
Notation

Write

\langle a; b; c; \ldots \rangle

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

(** [sum <a1; a2; ...> <b1; b2; ...>] is [<a1 + b1; a2 + b2; ...>] *)

let rec sum
  (Cons (h_a, tf_a))
  (Cons (h_b, tf_b)) = ?
Discussion

(** [map f <a; b; c; ...>] is
  [<f a; f b; f c; ...>] *)

let rec map f (Cons (h, tf)) =
A CUTE FIBONACCI TRICK
Fibonacci

<table>
<thead>
<tr>
<th>fibs</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
</table>

# Fibonacci

<table>
<thead>
<tr>
<th>fibs</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibs</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>...</td>
</tr>
</tbody>
</table>
## Fibonacci

<table>
<thead>
<tr>
<th>fibs</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl fibs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>...</td>
</tr>
</tbody>
</table>
Fibonacci

<table>
<thead>
<tr>
<th></th>
<th>fibs</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tl fibs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

(fibs + tl fibs)

fibs is
1 then
1 then
Fibonacci

```ocaml
let rec fibs = 
  Cons(1, fun () -> 
    Cons(1, fun () -> 
      sum fibs (tl fibs)))
```

But try: take 100 fibs

**Exponential amount of recomputation:** regenerate entire prefix of fibs, twice, for each element produced

Solution: the Lazy module, covered in textbook
Upcoming events

• [tomorrow] A3 due
• [next Tuesday] prelim exam
  • 90 minutes
  • Early and late seating
  • Must bring ID Card
  • 5-8 problems
  • 1-page handwritten cheat-sheet allowed
• Practice problems posted (see Discourse)
• Review session on Sunday (see Discourse)

This is judiciously lazy.

THIS IS 3110