Discussion 11

Streams and Laziness

Kenneth Fang (kwf37), Newton Ni (cn279)

March 6, 2019

Key Ideas

- Functions can be used to **delay** computation
- Streams represent **infinite** iterators

Noisy Expressions: Case Study

```
let noisy_zero : int =
  print_endline "HELLO, THIS IS 0";
  0

let noisy (number: int) : int =
  Printf.printf "HELLO, THIS IS %i\n" number;
  number
```

Eager Evaluation

```
let noisy (number: int) : int =
  Printf.printf "HELLO, THIS IS %i\n" number;
  number

let print_eager (number: int) : unit =
  Printf.printf "PRINTING...\n";
  Printf.printf "%i\n" number

print_eager (noisy 5)
```

Lazy Evaluation

```
let noisy (number: int) : int =
  Printf.printf "HELLO, THIS IS %i\n" number;
  number

let print_lazy (number: unit -> int) : unit =
  Printf.printf "PRINTING...\n";
  Printf.printf "%i\n" (number ())

print_lazy (fun () -> noisy 5)
```

Laziness: Real Examples

- Side effects

Laziness: Real Examples

- Side effects
- `let input = read_line ()`

Laziness: Real Examples

- Side effects
- `let input = read_line ()`
- Unused code paths

Laziness: Real Examples

- Side effects
- `let input = read_line ()`
- Unused code paths
- `let lookup key otherwise = (* ... *)`

Laziness: Real Examples

- Side effects
- `let input = read_line ()`
- Unused code paths
- `let lookup key otherwise = (* ... *)`
- Asynchronous callbacks

Laziness: Real Examples

- Side effects
- `let input = read_line ()`
- Unused code paths
- `let lookup key otherwise = (* ... *)`
- Asynchronous callbacks
- `load_image "cat.png" (fun image -> (* ... *))`

Streams

```
module type Stream = sig
  (* Abstract type *)
  type t

  (* Create a stream that goes from start to infinity *)
  val make: int -> t

  (* Get current value *)
  val head: t -> int

  (* Go to next iteration *)
  val tail: t -> t
end
```

Streams: Take One

```
type t = int * t
```

Streams: Take Two

```
type t = {
  head: int;
  tail: t;
}

let head s = s.head

let tail s = s.tail

let rec make (n: int) : t =
  failwith "Unimplemented"
```

Streams: Take Two

```
let rec make (n: int) : t =
  { head = n;
    tail = make (n + 1) }
```

Streams: Take Three

```
type t = {
  head: int;
  tail: unit -> t;
}

let head s = s.head

let tail s =
  failwith "Unimplemented"

let rec make (n: int) : t =
  failwith "Unimplemented"
```

Streams: Take Four

```
type t = Stream of int * (unit -> t)
```

Streams: Take Four

```
let head s = match s with
| Stream (h, _) -> h

let head = function
| Stream (h, _) -> h

let head (Stream (h, _)) = h
```

Streams: Exercises

```
(** [map s f] is the stream returning [f x] for
  * each [x] in stream [s] *)
let map s f = failwith "Unimplemented"

(** [take s n] is the list of the first [n]
  * elements in stream [s]. *)
let take s n = failwith "Unimplemented"
```

Streams: Examples

- Basic game interactive game tree in `game.ml`

Streams: Examples

- Basic game interactive game tree in `game.ml`
  - Runnable with provided Makefile

Streams: Examples

- Basic game interactive game tree in `game.ml`
  - Runnable with provided Makefile
- Infinite binary tree exercises in `tree.ml`