Lazy Programming

15-150 M21

Lecture 0730
28 July 2021

0 Mutual Recursion

0730.0 (mutual.sml)

```
fun even 0 = true
  | even n = odd(n-1)
and odd 0 = false
  | odd n = even(n-1)
```

Claim `even(n)` and `odd(n)` are valuable for all $n \geq 0$.

Proof. By induction on `n`.

BC `n=0`

$$\texttt{even 0} \Longrightarrow \texttt{true} \qquad \text{(Defn. even)}$$
$$\texttt{odd 0} \Longrightarrow \texttt{false} \qquad \text{(Defn. odd)}$$

IH `even(n)` and `odd(n)` are valuable for some `n`.

$$\texttt{even(n+1)} \Longrightarrow \texttt{odd n} \qquad \text{(Defn. even)}$$
$$\hookrightarrow \texttt{v} \qquad \text{(for some value v, by IH)}$$
$$\texttt{odd(n+1)} \Longrightarrow \texttt{even n} \qquad \text{(Defn. even)}$$
$$\hookrightarrow \texttt{v'} \qquad \text{(for some value v', by IH)}$$

```
mapAlt : ('a -> 'b) -> ('a -> 'b) -> 'a list -> 'b
  list
```

REQUIRES: `f` and `g` are total

ENSURES: `mapAlt f g [`x_0`, `x_1`, `x_2`,...]` $\Longrightarrow$ `[f(`x_0`), g(`x_1`), f(`x_2`),...]`

0730.1 (mutual.sml)

```
fun mapAlt f g [] = []
  | mapAlt f g (x::xs) = f(x) :: mapAlt' f g xs
and mapAlt' f g [] = []
  | mapAlt' f g (x::xs) = g(x) :: mapAlt f g xs
```

0730.2 (mutual.sml)

```
datatype 'a rosetree = Node of 'a rose list
     and 'a rose = Rose of 'a * 'a rosetree

fun size (Node L) = foldr op+ 0 (map size' L)
and size' (Rose(_,T)) = 1 + size T

fun depth (Node L) =
  foldr Int.max 0 (map depth' L)
and depth' (Rose(_,T)) = 1 + depth T
```

1 Streams

Key Distinction

In an *eager* language, expressions are always evaluated to values before being bound to identifiers (e.g. when being passed into a function)

In a *lazy* language, un-evaluated expressions can be bound to identifiers, and evaluated whenever needed

Tradeoffs

There are numerous tradeoffs between eager and lazy languages, such as:

- Lazy evaluation often ends up saving work, but the complexity of lazy code can be harder to reason about
- In a lazy language, you might be passing around an expression which loops forever if evaluated. But this can be useful:

```
x::xs
```

Module: Streams

0730.3 (STREAM.sig)

```
signature STREAM =
sig
  type 'a stream
  datatype 'a front =
    Empty | Cons of 'a * 'a stream
```

0730.4 (Stream.sml)

```
structure Stream : STREAM =
struct
  datatype 'a stream =
    Stream of unit -> 'a front
  and 'a front =
    Empty | Cons of 'a * 'a stream
```

0730.5 (STREAM.sig)

```
val delay : (unit -> 'a front) -> 'a stream
val expose : 'a stream -> 'a front
```

0730.6 (Stream.sml)

```
val delay = Stream
fun expose (Stream d) = d ()
```

Live Coding:
Stream of Natural Numbers

0730.7 (streamFns.sml)

```
fun natsFrom n () =
  Stream.Cons(n,Stream.delay(natsFrom (n+1)))

val nats = Stream.delay(natsFrom 0)
```

0730.8 (STREAM.sig)

```
  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val fromList : 'a list -> 'a stream
  val tabulate : (int -> 'a) -> 'a stream
```

0730.9 (Stream.sml)

```
val empty = Stream (fn () => Empty)
fun cons (x,s) = Stream (fn () => Cons (x,s))
```

Live Coding:

`Stream.tabulate`

0730.10 (Stream.sml)

```
fun tabulate f =
  delay (fn () => tabulate' f)
and tabulate' f =
  Cons (f 0, tabulate (fn i => f (i+1)))
```

0730.11 (STREAM.sig)

```
val null : 'a stream -> bool
val hd : 'a stream -> 'a
val tl : 'a stream -> 'a stream

val take : 'a stream * int -> 'a list
val drop : 'a stream * int -> 'a stream
val toList : 'a stream -> 'a list
```

Append

0730.12 (STREAM.sig)

```
val append : 'a stream * 'a stream -> 'a stream
```

0730.13 (Stream.sml)

```
fun append (s1,s2) =
    delay (fn () => append' (expose s1, s2))
and append' (Empty, s2) = expose s2
  | append' (Cons (x,s1), s2) =
      Cons (x, append (s1, s2))
```

Check Your Understanding

Implement `Stream.fromList` using `append`. For an extra challenge, do it in point-free form (i.e. `val fromList = ...`).

0730.14 (STREAM.sig)

```
  val map :
    ('a -> 'b) -> 'a stream -> 'b stream
  val filter :
    ('a -> bool) -> 'a stream -> 'a stream
  val zip :
    'a stream * 'b stream -> ('a * 'b) stream
```

Bad Map

0730.15 (Stream.sml)

```
fun map f s = map' f (expose s)
and map' f Empty = empty
  | map' f (Cons (x,s)) = cons (f x, map f s)
```

0730.16 (Stream.sml)

```
fun map f s =
  delay (fn () => map' f (expose s))
and map' f Empty = Empty
  | map' f (Cons (x,s)) = Cons (f x, map f s)
```

```
fun filter p s =
  delay (fn () => filter' p (expose s))
and filter' p Empty = Empty
  | filter' p (Cons (x,s)) =
      if p x then Cons (x, filter p s)
      else filter' p (expose s)
fun zip (s1, s2) =
  delay (fn () => zip'(expose s1, expose s2))
and zip' (_, Empty) = Empty
  | zip' (Empty, _) = Empty
  | zip' (Cons (x,s1), Cons (y,s2)) =
       Cons ((x,y), zip (s1,s2))
```

Summary

- Can write mutually-recursive data structures, code, and proofs
- Can insert a delay into the cons operation, allowing us to encode (potentially-infinite) streams
- Have to be careful not to expose elements of a stream until necessary

Next Time

- Mutability and effects in SML
- Imperative programming

Thank you!