Intro to Laziness

15-150 M21

Lecture 0728-1
28 July 2021

0 Lazy Combinator Tree Search

0728-1.0 (lazysearch.sml)

```
(* INVARIANT: For all values p : t pred, p is
  total *)
type 'a pred = 'a -> bool
(*  isEven : int pred *)
fun isEven x = x mod 2 = 0
```

Recall: work-optimized search

0728-1.1 (lazysearch.sml)

```
fun search p Empty = NONE
  | search p (Node(L,x,R)) =
      if p(x) then SOME x
      else
        (case search p L of
            (SOME z) => SOME z
           | _ => search p R)
```

Recall: span-optimized search

0728-1.2 (lazysearch.sml)

```
fun search' p Empty = NONE
  | search' p (Node(L,x,R)) =
      if p(x) then SOME x
      else
        (case (search' p L, search' p R) of
             (SOME z,_) => SOME z
            |(_,SOME z) => SOME z
            | _ => NONE)
```

Can do with nice combinator

0728-1.3 (lazysearch.sml)

```
fun optOrelse (SOME x,_) = SOME x
  | optOrelse (NONE,Y) = Y
infixr optOrelse

fun search' (p:'a pred) Empty = NONE
  | search' p (Node(L,x,R) : 'a tree) =
      if p(x) then SOME x else
      (search' p L) optOrelse (search' p R)
```

This is the span-optimized version because both arguments to `optOrelse` will get evaluated, in parallel (assuming adequate processors).

What about the work-optimized version?

Recall SML is a **eager** language, and so will fully evaluate the arguments to a function before stepping into the function body.

So we can't define a "short-circuiting" `optOrelse` which only evaluates its second arg when its first argument is `NONE`.

SML is eager, unless. . .

we tell it not to be!

A value of type `unit -> t` is of the form

```
fn () => e
```

which we think of "`e`, suspended", that is, `e` but tagged to not evaluate yet.

0728-1.4 (lazysearch.sml)

```
type 'a lazy = unit -> 'a
fun Eval (f:'a lazy):'a = f()
fun Suspend (x:'a):'a lazy = fn () => x
```

Claim `Suspend` is total

Claim If `e:t` is valuable, `Eval(fn () => e)` is valuable. In particular, for all values `v:t`, `Eval(Suspend v)` is valuable.

```
val rec loop : string lazy =
  fn () => loop ()
```

elseTry

```
elseTry : 'a option lazy * 'a option lazy
          -> 'a option lazy
```

REQUIRES: true
ENSURES:

$$\texttt{Eval(elseTry(f,g))} \cong \begin{cases} \texttt{Eval f} & \text{if } \texttt{Eval f} \text{ is not } \texttt{NONE} \\ \texttt{Eval g} & \text{if } \texttt{Eval(f)} \Longrightarrow \texttt{NONE} \end{cases}$$

0728-1.5 (lazysearch.sml)

```
fun elseTry (f : 'a option lazy,
             g : 'a option lazy)
        :'a option lazy =
   fn () =>
   case Eval f of
     NONE => Eval g
   | X => X
infixr elseTry
```

```
Search : 'a pred -> 'a tree -> 'a option lazy
```
REQUIRES: true
ENSURES:

$$\texttt{Eval(Search p T)} \cong \begin{cases} \texttt{SOME(z)} & \text{where } \texttt{z} \text{ is the top-leftmost element} \\ & \text{such that } \texttt{p(z)} \cong \texttt{true} \\ \texttt{NONE} & \text{if there is no such } \texttt{z} \end{cases}$$

0728-1.6 (lazysearch.sml)

```
fun Return (x:'a):'a option lazy =
  Suspend(SOME x)
```

0728-1.7 (lazysearch.sml)

```
fun Search p Empty = Suspend NONE
  | Search p (Node(L,x,R)) =
      if p(x) then Return x else
      Search p L elseTry Search p R
```

Next Time

- More elaborate laziness
- Infinite data structures

Thank you!