lazy programming

• Standard ML uses *call-by-value* evaluation
  • functions *always* evaluate their arguments
• Some languages use *lazy* evaluation
  • functions may *defer* argument evaluation until its value is *needed*

You can still be lazy in a call-by-value language, but it takes effort and careful planning!
being lazy

*Defer* computation until its result is *needed*

- a function value
  
  \[ h : \text{unit} \rightarrow t \]  
  
  *(a "thunk")*

  represents a *delayed computation* for a value of type \( t \)

- to compute this value, call \( h() \) to *force* the thunk

```
```

```
Never put off until tomorrow what you can do the day after tomorrow.
```

Mark Twain
call-by-value

```
repeat : 'a list -> 'a list
fun repeat L = L @ repeat L
```

`repeat` evaluates, to a function value

```
repeat =>* (fn L => L @ repeat L)
```

`repeat [0]` doesn’t terminate

```
repeat [0] =>* (fn L => L @ repeat L) [0]
=> [0] @ repeat [0]
=>* [0] @ ([0] @ repeat [0])
=>* . . . forever
```

lists are finite

no way to build infinite lists
lazy lists

data type 'a lazylist =
Cons of 'a * (unit -> 'a lazylist)

val Cons = fn - : 'a * (unit -> 'a lazylist) -> 'a lazylist

• Values of type t lazylist represent infinite lazy lists

• A value of type t lazylist has form Cons(v, h)
where v is a value of type t and h is a thunk

  • the head element v is immediately available

  • force h if you need later elements
all zeros

zeros : unit -> int lazylist

fun zeros ( ) = Cons(0, zeros)

Zeros : int lazylist
val Zeros = zeros( )
show

show : int -> 'a lazylist -> 'a list
fun show 0 _ = [ ]
| show n (Cons(x, h)) = x :: show (n-1) (h( ))

show 5 Zeros
evaluates to
[0,0,0,0,0]
natural numbers

nats : int -> int lazylist
fun nats n = Cons(n, fn ( ) => nats (n+1))

Nats : int lazylist
val Nats = nats 0

show 1 Nats evaluates to [0]
show 2 Nats evaluates to [0, 1]
show 3 Nats evaluates to [0, 1, 2]
mklazy : int list \rightarrow int lazylist

fun mklazy [ ] = Zeros
| mklazy (x::L) = Cons(x, fn ( ) => mklazy L)

Converts a finite int list to an infinite int lazylist by padding with 0’s
A value of type \( t \) lazylist represents an infinite lazy list of values of type \( t \).

\[
L : t \text{ lazylist} \text{ represents } v_1, v_2, \ldots v_n, \ldots \\
\text{iff for all } n \geq 0, \text{ show } n \ L = [v_1, v_2, \ldots, v_n]
\]

- **Zeros** represents 0,0,0,0,…
- **Nats** represents 0,1,2,3,…
- **nats 42** represents 42,43,44,45,…
- **mklazy \([1,2,3]\)** represents 1,2,3,0,0,0,…
lazy append

append : 'a list * (unit -> 'a lazylist) -> 'a lazylist

fun append ([ ], h) = h( )
|  append (x::xs, h) = Cons(x, fn ( ) => append(xs, h))

append([0,1], fn ( ) => nats 2)
represents
0,1,2,3,4,…
evaluation

fun append ([ ], h) = h( )
1  append (x::xs, h) = Cons(x, fn ( ) => append(xs, h))

append([0,1], fn ( ) => nats 2) =>* Cons(0, h₁)
   where h₁( ) =>* Cons(1, h₂)
   and so on…

Not true that
append([0,1], fn ( ) => nats 2) =>* Cons(0, fn ( ) => Cons(1, h₂))
lazy append

If

\[ h( ) \]
represents \( y_0, y_1, y_2, y_3, \ldots \)

then

\[ \text{append } ([x_0, \ldots, x_k], h) \]
represents \( x_0, \ldots, x_k, y_0, y_1, y_2, y_3, \ldots \)
repeat

repeat : 'a list -> 'a lazylist
REQUIRES k > 0
ENSURES repeat [x₁,…,xₖ]
    represents x₁,…, xₖ, x₁,…, xₖ,…

fun repeat xs = append(xs, fn ( ) => repeat xs)

repeat [0] represents 0, 0, 0, …
evaluation

repeat [0]  =>*  Cons(0, h_0)

where  h_0( )  =>*  Cons(0, h_1)
and  h_1( )  =>*  Cons(0, h_2)

and so on

repeat [0]  represents  0,0,0,0,0,...
We say that values $A, B : t \text{ lazy list}$ are equal iff they represent the same lazy infinite lists, i.e.

- For all $n \geq 0$, $\text{show } n A = \text{show } n B$.

*Expressions of type $t \text{ lazy list}$ are equal* iff they both fail to terminate, or they raise the same exception, or they evaluate to equal values.
lazy equality

Same as saying that \( e_1, e_2 : t \text{ lazylist} \) are equal iff

- there are values \( v_1, v_2 : t \) and \( h_1, h_2 : \text{unit} \to t \text{ lazylist} \) such that
  \[
  e_1 \to^* \text{Cons}(v_1, h_1) \text{ and } e_2 \to^* \text{Cons}(v_2, h_2)
  \]
  and \( v_1 = v_2 \) and \( h_1 = h_2 \)

- or both fail to terminate

- or both raise the same exception

This looks like a circular definition, so it’s usually easier to work with the show-based definition.
referential transparency

- With this definition of equality
  we still have referential transparency

\[
\text{nats } 0 = \text{Cons}(0, \text{fn} (\ ) \Rightarrow \text{nats } (0+1)) \\
= \text{Cons}(0, \text{fn} (\ ) \Rightarrow \text{nats } 1) \\
= \text{Cons}(0, \text{fn} (\ ) \Rightarrow \text{Cons}(1, \text{fn} (\ ) \Rightarrow \text{nats } 2))
\]

by def of nats

since 0+1 = 1

since nats 1 = Cons(1, fn ( ) => nats 2)

In each case, for all \( n \geq 0 \),

show n LHS = show n RHS = \([0,1,2,...,n-1]\)
According to this definition of “lazy equality” we have

\[
\text{zeros}(\ ) = \text{repeat} \ [0] \\
\text{nats} \ n = \text{Cons}(n, \ \text{fn} \ ( ) => \text{nats}(n+1)) \\
= \text{Cons}(n, \ \text{fn} \ ( ) => \text{Cons}(n+1, \ \text{fn} \ ( ) => \text{nats}(n+2))) \\
\text{nats} \ 0 = \text{append}([0,1], \ \text{fn} \ ( ) => \text{nats} \ 2)
\]

In each case, the expressions evaluate to equal values.
comments

• Lazy lists have many features in common with (finite) lists

• But they can be used to implement lazy, demand-driven computation

• And show how to deal with potentially infinite data while avoiding infinite regress
lazy map

map : ('a -> 'b) -> 'a lazylist -> 'b lazylist

fun map f (Cons(x, h))
    = Cons(f x, fn ( ) => map f (h( )))

map (fn x => x+1) Zeros

If \( f \) is total and \( L \) represents \( x_0, x_1, x_2, \ldots \)
then \( \text{map} f L \) represents \( f(x_0), f(x_1), f(x_2), \ldots \)
eager map

map : ('a -> 'b) -> 'a lazylist -> 'b lazylist

fun map f (Cons(x, h)) =
  let
    val L = map f (h( ))
  in
    Cons(f x, fn ( ) => L)
  end

map (fn x => x+x) Ones

eagerness is not necessarily a virtue
map is lazy List.map

If \( f : t \rightarrow t' \) is total,

then for all \( L : t \) lazylist, and all \( n \geq 0 \),

\[
\text{show } n \ (\text{map } f \ L) = \text{List.map } f \ (\text{show } n \ L)
\]

PROOF? Use induction on \( n \)

\[
\begin{align*}
\text{map } f \ (\text{Cons}(x, h)) & = \ \text{Cons}(f \ x, \text{fn ( ) } \Rightarrow \text{map } f \ (h( ))) \\
\text{List.map } f \ (x::xs) & = (f \ x) :: \text{List.map } f \ xs \\
\text{show } 0 \ _ & = \ [ ] \\
\text{show } n \ (\text{Cons}(x, h)) & = x :: \text{show } (n-1) \ (h( ))
\end{align*}
\]
map is lazy \textbf{List.map}

\begin{align*}
P(n): \text{For all } L : \text{t lazylist, all total } f, \\
\text{show } n \ (\text{map } f \ L) &= \text{List.map } f \ (\text{show } n \ L)
\end{align*}

\textbf{THEOREM} \ \forall n \geq 0. \ P(n)

\textbf{PROOF} \ \text{By induction on } n

Base: \ n=0. \ \text{Both sides are } [ ] .

Inductive step: \ n > 0. \ \text{We show that } P(n-1) \ \text{implies } P(n).

Assume IH: \ P(n-1). \ \text{Let } L = \text{Cons}(x, h) \ \text{and } v = f \ x.

\begin{align*}
\text{show } n \ (\text{map } f \ L) &= \text{show } n \ (\text{map } f \ (\text{Cons}(x, h))) \\
&= \text{show } n \ (\text{Cons}(f \ x, \text{fn ( ) } => \text{map } f \ (h( )))) \\
&= \text{show } n \ (\text{Cons}(v, \text{fn ( ) } => \text{map } f \ (h( )))) \\
&= v :: \text{show } (n-1) \ (\text{map } f \ (h( ))) \\
&= v :: \text{List.map } f \ (\text{show } (n-1) \ (h( ))) \\
&= \text{List.map } f \ (x :: (\text{show } (n-1) \ (h( )))) \\
&= \text{List.map } f \ (\text{show } n \ (\text{Cons}(x, h))) \\
&= \text{List.map } f \ (\text{show } n \ L)
\end{align*}

So \ P(n) \ \text{holds.}
lazy map fusion

If \( f \) and \( g \) are total and composable, then for all suitably typed lazy lists \( L \),

\[
\text{map } (f \circ g) \ L = \text{map } f \ (\text{map } g \ L)
\]

\[
\text{same as}
\]

\[
\text{map } (f \circ g) = (\text{map } f) \circ (\text{map } g)
\]
lazy zip

\[
\text{zip} : \ 'a \text{ lazylist} \times 'b \text{ lazylist} \rightarrow ('a \times 'b) \text{ lazylist}
\]

\[
\text{fun} \quad \text{zip} \ (\text{Cons}(x, h), \text{Cons}(y, k)) \\
\quad = \text{Cons}((x, y), \text{fn} (\ ) \Rightarrow \text{zip} \ (h( ), k( )))
\]

recursive call is deferred

If \( A \) represents \( a_0, a_1, a_2, \ldots \)
and \( B \) represents \( b_0, b_1, b_2, \ldots \)
then \( \text{zip}(A, B) \) represents \( (a_0, b_0), (a_1, b_1), (a_2, b_2), \ldots \)
**zip is lazy List.zip**

For all $A : t_1$ lazylist and $B : t_2$ lazylist and all $n \geq 0$,

$$\text{show } n \ (\text{zip } (A,B)) = \text{List.zip } (\text{show } n \ A, \text{show } n \ B)$$

**PROOF?** Use induction on $n$

\[
\begin{align*}
\text{zip } (\text{Cons}(x, h), \text{Cons}(y, k)) &= \text{Cons}((x, y), \text{fn } (\ ) \Rightarrow \text{zip } (h( ), k( ))) \\
\text{List.zip } (x::xs, y::ys) &= (x, y) :: \text{List.zip } (xs, ys) \\
\text{show } 0 \ _ = [ ] \\
\text{show } n \ (\text{Cons}(x, h)) &= x :: \text{show } (n-1) \ (h( ))
\end{align*}
\]
lazy filter

filter : ('a -> bool) -> 'a lazylist -> 'a lazylist

fun filter p (Cons(x, h)) =
  if p x then Cons(x, fn ( ) => filter p (h( )))
  else filter p (h( ))

if x doesn’t satisfy p, look in tail

if x satisfies p, x is head of filtered list, defer further filtering until needed
What happens when we evaluate

```
filter (fn x => x<0) (repeat [1])
```
specification

If $p$ is total, $L$ represents $a_0, a_1, a_2, \ldots $ and *infinitely many* $a_i$ *satisfy* $p$

then $\text{filter } p \ L$ represents the lazy list of all the $a_i$ that satisfy $p$
fun nats n = Cons(n, fn () => nats (n+1))

val Evens = filter (fn x => (x mod 2 = 0)) (nats 0)

show 1 Evens = [0]
show 2 Evens = [0,2]
show 3 Evens = [0,2,4]
show 4 Evens = [0,2,4,6]
show 5 Evens = [0,2,4,6,8]

Evens represents 0,2,4,6,8,10,….

the lazy list of all non-negative even integers
a case study

• A problem and two solutions
  • one is eager
  • one is lazy

• Contrast the designs and their efficiency

“...I will always choose a lazy person to do a difficult job, because he will find an easy way to do it.”
Bill Gates
the mex sequence

(0,0), (1,2), (3,5), (4,7), (6,10), (8,13), (9,15), (11,18), ...

• An infinite sequence of pairs of natural numbers
• The 0th pair is (0, 0)
• For \( n \geq 1 \) the \( n^{th} \) is \((m, m+n)\) where
  
m is the minimum excluded number so far

For \( n=1 \):
  so far only 0;
  1 is minimum excluded number;
  we get (1, 2)

For \( n=2 \):
  so far only 0, 1, 2;
  3 is minimum excluded number;
  we get (3, 5)
eager solution

- Maintain a (sorted) (finite) list \( L \), of the integers seen so far, and keep track of \( n \)

- To find the next pair, search for the smallest non-negative integer not seen yet, say \( m \)

- Insert \( m \) and \( m+n \) into accumulator list and “return” the pair \((m, m+n)\)
ins : int * int list -> int list
REQUIRES L sorted
ENSURES ins(x,L) = sorted perm of x::L

fun ins(m, [ ])  = [m]
| ins(m, x::L) = case Int.compare(m, x) of
| LESS         => m::x::L
| EQUAL      => x::L
| GREATER => x::ins(m,L)

mem : int * int list -> bool
REQUIRES L sorted
ENSURES mem(i,L) = true if i is in L, false otherwise

fun mem(i, [ ])  = false
| mem(i, x::L) = case Int.compare(i, x) of
| LESS => false
| EQUAL => true
| GREATER => mem (i, L)
eager helpers

mex : int list -> int
REQUIRES L sorted
ENSURES mex L = smallest non-negative integer not in L

fun mex L =
let
  fun loop i = if mem(i, L) then loop(i+1) else i
in
  loop 0
end
eager mexlister

mexlister : int -> int list
REQUIRES k >= 0
ENSURES mexlister k = the first k mex pairs

fun mexlist (L, n) k =
  if k=0 then [ ] else
  let
    val m = mex L
  in
    (m, m+n) :: mexlist (ins(m, ins(m+n, L)), n+1) (k-1)
  end;

fun mexlister k = mexlist ([ ], 0) k
lazy solution

• Maintain a (sorted) lazy list $L$, of the integers not used yet, and keep track of $n$

• To find the next pair, we already have the smallest integer not used yet, say $m$

• Delete $m$ and $m+n$ from the lazy list and “return” the pair $(m, m+n)$
helpers

fun del (m, Cons(x, h)) =
    case Int.compare(m, x) of
        LESS  => Cons(x, h)
        |    EQUAL      => h( )
        |    GREATER => Cons(x, fn ( ) => del (m, h( )))

del : int * int lazy list -> int lazy list
REQUIRES L sorted,
    represents an infinite list
ENSURES del(m, L) sorted,
    represents same infinite list except for m
lazy mexlister

lazy_mexlister : int -> int list
REQUIRES k >= 0
ENSURES lazy_mexlister k = the first k mex pairs

fun lazy_mexlist (Cons(m, h), n) =
    Cons((m, m+n), fn () => lazy_mexlist (del(m+n, h()), n+1));

fun lazy_mexlister k = show k (lazy_mexlist (nats 0, 0))
results

- mexlister 10;

val it = [(0,0),(1,2),(3,5),(4,7),(6,10),(8,13),(9,15),(11,18),(12,20),(14,23)] : (int * int) list

- lazy_mexlister 10;

val it = [(0,0),(1,2),(3,5),(4,7),(6,10),(8,13),(9,15),(11,18),(12,20),(14,23)] : (int * int) list

- mexlister 2000;

SLOW val it = [(0,0),(1,2),(3,5),(4,7),(6,10),(8,13),(9,15),(11,18),(12,20),(14,23), ......] : (int * int) list

- lazy_mexlister 2000;

FAST val it = [(0,0),(1,2),(3,5),(4,7),(6,10),(8,13),(9,15),(11,18),(12,20),(14,23), ......] : (int * int) list