1 Topics

- Lazy programming (in a call-by-value language)
- Delaying computation
- Demand-driven manipulation of infinite data

As usual, these notes cover some examples that were not yet discussed in class, and some examples from the lecture slides don’t appear here. We encourage you to read notes and slides.
2 Introduction

ML is a call-by-value language: functions evaluate their arguments. So an attempt to build an infinite list, such as the following code fragment, is doomed to failure (actually, will loop forever):

(* repeat : 'a list -> 'a list *)
fun repeat L = L @ repeat L;

Even though ML tells us this function is well-typed, if we apply it to a list we won’t get a value. For example

\[
\begin{align*}
\text{repeat } \text{[0]} & \Rightarrow* \text{[0] @ repeat } \text{[0]} \\
& \Rightarrow* \text{[0] @ ([0] @ repeat } \text{[0]}) \\
& \Rightarrow* \ldots \text{ forever}
\end{align*}
\]

Although this is a silly example, it illustrates a problem. We may need to build and manipulate infinite data structures, and deal with such data in a demand-driven manner: just evaluate as much as we need, while leaving the rest for later if we need it. The call-by-value behavior of functions needs to be side-stepped somehow, if we want to prevent eager evaluation of the rest of the data.

In fact we can simply use functions themselves to delay computation. Functions are values: ML doesn’t evaluate inside a function body until if and when the function is applied to a value.
3 Representing infinite data

In a functional programming language, as we have already seen, one way to represent an infinite list is as a function from integers to reals; we can represent an infinite power series with real coefficients in this way. However, this kind of representation isn’t tailored for lazy demand-driven computation.

Today we’ll introduce another representation for infinite lists, as a datatype with a single constructor (like the :: constructor for finite lists). And we’ll show you how to implement operations on infinite lazy lists that allow you to solve some non-trivial problems that involve demand-driven solutions.

The main idea is that a lazy list of values of type \( t \) has a **head** (a value of type \( t \)) and a **tail**, which is also a lazy list, but which is “deferred” inside the body of a “thunk” (a function of type \( \text{unit} \rightarrow t\ \text{lazylist} \)). Here is the datatype definition:

```plaintext
datatype 'a lazylist = Cons of 'a * (unit -> 'a lazylist);
```

After this definition, ML tells us

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

Incidentally, we could just as easily introduced a lazy list datatype that also included a \texttt{Nil} value representing the empty lazy list, and values of this type would represent finite lazy lists and infinite lazy lists. Since the problems we have in mind only involve infinite lazy lists, we chose not to.

**Examples**

Here are some examples that construct values of type \( \text{int lazylist} \):

```plaintext
(* zeros : \text{unit} \rightarrow \text{int lazylist} *)
fun zeros ( ) = Cons(0, zeros);

(* Zeros : \text{int lazylist} *)
val Zeros = zeros( );

(* mklazy : \text{int list} \rightarrow \text{int lazylist} *)
fun mklazy [ ] = Zeros
| mklazy (x::L) = Cons(x, fn ( ) => mklazy L);
```
(* ones : unit -> int lazylist *)
fun ones ( ) = Cons(1, ones);

(* Ones : int lazylist *)
val Ones = Cons(1, ones);

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

val Nats = nats 0;

(* lazyappend : 'a list * (unit -> 'a lazylist) -> 'a lazylist *)
fun lazyappend ([ ], h) = h( )
| lazyappend (x::xs, h) = Cons(x, fn ( ) => lazyappend(xs, h));

(* repeat : 'a list -> 'a lazylist *)
(* REQUIRES xs not empty list *)
(* ENSURES repeat(xs) terminates *)
fun repeat xs = lazyappend(xs, fn ( ) => repeat xs);

What happens if we evaluate repeat [ ]?

**Viewing a lazy list**

Here is a useful function that shows us a “view” of an infinite lazy list: we can look at the first \( n \) items in the lazy list, as an ordinary finite list.

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

Example: show 5 Nats = [0,1,2,3,4].

Note: “showing” the first few elements of a lazy list “forces” the first few thunks to be applied to their (dummy) argument ( ) to extract data from the tail of the list. This is all purely functional, without side-effect.

If we evaluate show 2 Nats after show 5 Nats we’ll get the result [0,1], not [5,6]. The first show doesn’t change the value of the lazy list.
Using lazy lists of integers

You may ask why such infinite lazy lists are useful. One answer is that we can use them to represent mathematical values. For example, it is obvious that a lazy list of integers could be used to represent a real number. We could use the head to represent the integer part of the real number, and then the tail to represent the decimal digits, with the most significant digit (the tenths part) first, then the hundredths part, and so on. According to this representation, Ones represents the real number whose decimal expansion is 1.11111....

```ml
val X = lazyappend ([5, 8], fn () => repeat [1,4,4]); (* X represents the real number equal to 3227/555 *) (* This number has decimal expansion 5.8144144144. . . *)

val Y = repeat [0,1,2,3,4,5,6,7,9]; (* Y represents 10/81 *) (* The decimal expansion for 10/81 is 0.12345679012345679. . . *)

val Z = repeat [0,9]; (* Z represents 10/11 *) (* The decimal expansion for 10/11 is 0.909090. . . *)
```

A value of type int lazylist is only decimal valid if its tail is a lazy list of decimal digits; the head can be any integer, positive, zero, or negative. And a decimal valid lazy list represents a real number. And we know that a real number can have more than one decimal valid representation. For example, the real number 1.0 is represented in decimal by either of the following:

```
0.99999. . .
1.00000. . .
```

Ones is a decimal valid, but Nats is not.

We cannot define a useful ML helper function for checking if a value of type int lazylist is a valid decimal representation! It would need to do an infinite amount of work! So instead we’ll need to rely on specs and proofs.

We will return to this topic after introducing lazy analogues of some familiar higher-order functions on lists.
Higher-order functions on lazy data

By analogy with the corresponding functions on ordinary lists:

- **lazymap**: \((\texttt{a} \to \texttt{b}) \to \texttt{a} \texttt{lazylist} \to \texttt{b} \texttt{lazylist}\)

  
  \[
  \text{fun lazymap } f \ (\text{Cons}(x, h)) = \text{Cons}(f \ x, \text{fn ( ) } \Rightarrow \text{lazymap } f \ (h( ))) ;
  \]

  If \(f : \texttt{t1} \to \texttt{t2}\) is total, and \(L : \texttt{t1 lazylist}\) has elements \(d_0, d_1, \ldots, d_n, \ldots\), then \(\text{lazymap } f \ L\) has type \(\texttt{t2 lazylist}\), and elements \(f(d_0), f(d_1), \ldots, f(d_n), \ldots\).

- **lazyzip**: \(\texttt{a lazylist} \ast \texttt{b lazylist} \to (\texttt{a} \ast \texttt{b}) \texttt{lazylist}\)

  
  \[
  \text{fun lazyzip } (\text{Cons}(x, h), \text{Cons}(y, k)) \ = \text{Cons}((x,y), \text{fn ( ) } \Rightarrow \text{lazymap } h( ), k( ))) ;
  \]

  If \(A : \texttt{t1 lazylist}\) has elements \(a_0, a_1, \ldots, a_n, \ldots\), and \(B : \texttt{t2 lazylist}\) has elements \(b_0, b_1, \ldots, b_n, \ldots\), then \(\text{lazymap } (A,B)\) evaluates to a value \(Z : (\texttt{t1 \ast t2}) \texttt{lazylist}\), with elements \((a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n), \ldots\).

- **lazyfilter**: \((\texttt{a} \to \texttt{bool}) \to \texttt{a} \texttt{lazylist} \to \texttt{a} \texttt{lazylist}\)

  
  \[
  \text{fun lazyfilter } p \ (\text{Cons}(x, h)) = \begin{cases} 
  \text{if } p \ x \ \text{then } \text{Cons}(x, \text{fn ( ) } \Rightarrow \text{lazymap } p \ (h( ))) \\
  \text{else } \text{lazymap } p \ (h( )) ;
  \end{cases}
  \]

  If \(L : \texttt{t lazylist}\) and \(p : \texttt{t\to bool}\) is total, \(\text{lazymap } p \ L\) computes the lazy list of the elements of \(L\) that satisfy \(p\).

  What happens with \(\text{filter } (\text{fn } _= \Rightarrow \text{false}) \text{ Ones}\)?

  If \(L\) has an element satisfying \(p\), and \(p\) is total, then \(\text{lazymap } p \ L\) evaluates to a lazy list value of form \(\text{Cons}(x, h)\), where \(x\) is the first element of \(L\) that satisfies \(p\); and \(h\) is a thunk for computing the remaining lazy list of elements satisfying \(p\).

It is natural to see the first two functions as “extensions” of the corresponding functions on ordinary lists, because we can prove the following results:
For all non-negative \( n \), and all suitably typed values \( f, L \) and \( R \):

\[
\begin{align*}
\text{show } n \ (\text{lazymap } f \ L) &= \text{List.map } f \ (\text{show } n \ L) \\
\text{show } n \ (\text{lazyzip } (L, R)) &= \text{List.zip } (\text{show } n \ L, \text{show } n \ R)
\end{align*}
\]

The relationship between \texttt{lazymap} and \texttt{List.filter} is more awkward to express, and we leave it as an exercise.

**Equality of lazy lists**

When should we say that two lazy list expressions are “equal”? In what sense are

\[
\begin{align*}
\text{lazymap } (f \circ g) \ L \\
\text{lazymap } f \ (\text{lazymap } g \ L)
\end{align*}
\]
equivalent, assuming as usual that \( f \) and \( g \) are total functions whose types allow them to be composed?

The notion of equality that makes sense for lazy lists is that lazy list values \( X \) and \( Y \) are equal iff, for all \( n \geq 0 \), \( \text{show } n \ X \) and \( \text{show } n \ Y \) are equal (ordinary) lists. This means that the two lazy lists have equal elements, in all positions. This definition also allows for the possibility that an attempt to extract an element may loop forever (as in the \texttt{lazymap} example when there is no element satisfying the predicate). So two lazy lists whose first 42 elements are equal and whose 43rd elements fail to terminate are also regarded as equal.

Again, it is impossible to implement an equality-check for lazy lists. Unlike ordinary lists of integers, we cannot use the built-in = operator of ML to check equality. And to prove that two lazy list expressions evaluate to equal values will typically require an inductive argument, most likely by using induction on \( n \).

Note that according to this notion of equality, the lazy integer lists

\[
\begin{align*}
\text{val One } &= \text{Cons}(1, \text{zeros}) \\
\text{val One' } &= \text{Cons}(0, \text{fn } () \Rightarrow \text{repeat } [9])
\end{align*}
\]

are not equal, even though they both represent the same real number.
Lazy fusion laws

With this as our notion of equality, the following lazy fusion results are provable:

For all types $t_1, t_2, t_3$, all total functions $f : t_2 \rightarrow t_3$ and $g : t_1 \rightarrow t_2$, and all values $L : t_1 \text{ lazylist}$,

$$\text{lazymap (} f \circ g \text{)} L = \text{lazymap } f \ (\text{lazymap } g \ L).$$

The proof uses induction on $n$ to show that (under the given assumptions) for all $n \geq 0$,

$$\text{show } n \ (\text{lazymap (} f \circ g \text{)} L) = \text{show } n \ (\text{lazymap } f \ (\text{lazymap } g \ L)).$$

In fact, we can appeal to the above results linking lazymap and List.map, and the familiar fusion property of List.map, i.e. that

$$\text{List.map (} f \circ g \text{)} (\text{show } n \ L) = \text{List.map } f \ (\text{List.map } g \ (\text{show } n \ L))$$

in deriving the lazy fusion law.

Note that we can paraphrase this lazy fusion law, taking advantage of extensionality. The following is an equivalent statement to the one given above:

For all types $t_1, t_2, t_3$, and all total functions $f : t_2 \rightarrow t_3$ and $g : t_1 \rightarrow t_2$,

$$\text{lazymap (} f \circ g \text{)} = (\text{lazymap } f) \circ (\text{lazymap } g).$$
4 Examples

Now a series of examples illustrating how to program with lazy lists. The art is in the design of thunks (often using recursion), and making the control flow sufficiently lazy that we never prematurely reveal the head-tail structure of part of a lazy list until we need to.

Lazy list of factorials

Useful facts about factorials:

- \( 0! = 1 \)
- \( k! \cdot (k+1) = (k+1)! \) when \( k \geq 0 \).

The following lazy way to generate factorials is based on these useful facts. The first fact tells us what the head of the lazy list should be (1). The second fact tells us that we can obtain the later factorials by zipping each integer \( k + 1 \) with the factorial of its predecessor and multiplying. So we embed a recipe for generating factorials starting from 1 and zipping and multiplying, in the tail thunk.

\[
(* \text{ facts : unit -> int lazy list } *)
\]

\[
\text{fun facts( ) = Cons(1, fn ( ) =>
\quad \text{lazymap(op * )}(\text{lazyzip(facts( ), nats 2 ( ))}))}
\]

\text{facts( ) evaluates to the lazy list of factorials.}

\[
\text{show 5 (facts( )) = [1,2,6,24,120].}
\]

For all non-negative \( n \), \text{show n (facts( )) evaluates to the list}

\[
[0!, 1!, \ldots, n!].
\]

This is the same as saying that

\[
\text{show n (facts( )) = [0!, 1!, \ldots, n!].}
\]

It’s a good exercise to do the proof of this result (use induction on \( n \)).
Lazy list of prime numbers

This code is based on the Sieve of Eratosthenes, an ancient algorithm for enumerating the primes.

(* sift : int -> int lazylist -> int lazylist *)
fun sift x (Cons(y, h)) = 
  if (y mod x = 0) then sift x (h( ))
  else Cons(y, fn ( ) => sift x (h( )));

(* sieve : int lazylist -> int lazylist *)
fun sieve (Cons(x, h)) = 
  Cons(x, fn ( ) => sieve (sift x (h( ))));

val Primes = sieve(nats 2 ( ));

Primes is the lazy list of prime numbers, in ascending order of enumeration.

  show 5 Primes = [2,3,5,7,11]

For all non-negative \(n\), \(\text{show } n \text{ Primes}\) evaluates to the list of the first \(n\) primes. The following facts about primes are relevant in proving this:

- 2 is the first (smallest) prime; there are infinitely many primes.
- For each \(n \geq 0\), the \((n + 1)\)th prime is the smallest integer \(\geq 2\) not divisible by any of the first \(n\) primes.

If \(L = \text{Cons}(x, h)\) is the lazy list of all integers \(\geq 2\) not divisible by any of the first \(n\) primes, then \(x\) is the \((n + 1)\)th prime, and \(\text{sift } x \text{ (h( ))}\) evaluates to the lazy list of all integers \(\geq 2\) not divisible by any of the first \(n+1\) primes.
5 Self-test

1. In these notes we’ve dealt with a datatype of infinite lazy lists. It is also easy to handle finite or infinite lazy lists at the same time, using the following datatype:

   ```ml
   datatype 'a lazylist = Nil | Cons of 'a * (unit -> 'a lazylist)
   ```

   Redefine all of the primitive functions (like `lazymap`, `lazyzip` and so on) to handle finite as well as infinite lazy lists.

2. Prove that for all \( n \geq 0 \), and all suitably typed values \( f, L \),

   ```ml
   show n (lazymap f L) = map f (show n L),
   ```

   where `map` is the usual map function on lists and you can assume that \( f \) is total.

3. Let `badmap` be the function defined by:

   ```ml
   fun badmap f (Cons(x, xs)) = 
       let val R = badmap f (xs ( )) in Cons(f x, fn ( ) => R) end
   ```

   Explain why it is NOT true that for all suitably typed values \( f, L \), and all \( n \geq 0 \),

   ```ml
   show n (badmap f L) = map f (show n L),
   ```

   even when \( f \) is known to be total.

4. Write an ML function

   ```ml
   member : int -> int lazylist -> bool
   ```

   such that whenever \( L \) is an increasing lazy list of integers and \( x \) is an integer, \( member \ x \ L \) returns `true` if \( x \) occurs in \( L \), `false` otherwise.

   Why is it not possible to define a total ML function that satisfies the following specification?
REQUIRES true
ENSURES member x L = true if x occurs in the lazy list L,
member x L = false otherwise.

5. Write a recursive ML function

\[
\text{frac} : \text{int} \times \text{int} \rightarrow \text{int} \text{ lazylist}
\]

such that when \( m \geq 0 \) and \( n > 0 \), \( \text{frac} \ (m, \ n) \) evaluates to a lazy list of
the digits in the decimal representation of the fraction \( \frac{m}{n} \), i.e the decimal
digits of the real number \( \text{real} \ m)/(\text{real} \ n) \). For example, \( \text{frac} \ (3, 4) \)
should evaluate to the lazy list consisting of 0,7,5,0,0,0,… because
\( \frac{3}{4} = 0.75000…. \)

Use your function to find the values of:

- frac(1, 9)
- show 10 (frac (1,9))
- show 20 (frac (3227,555))