Lazy Lists

Lists where the nex element is computed on demand.

Also known as *streams* in the literature.

(not to be confused with input/output streams)

- Avoids waste if the entire list is not needed.
- Infinite objects are a useful abstraction.
- Models simple I/O.

Disadvantage: Termination is lost

Lazy Lists in ML

`()` —the unique tuple of length 0 has type `unit`.

A datatype for lazy lists:

```
datatype 'a seq = Nil
  | Cons of 'a * (unit -> 'a seq);
```

`Cons`(h, tf) is the sequence with head h and tail function tf.

Example: the infinite sequence $k, k+1, k+2, \ldots$

```
fun from k =
  Cons(k, fn () => from(k+1));
```

```
val from = fn : int -> int seq
```

Operations on Lazy Lists

```
exception Empty;

fun hdq Nil = raise Empty
  | hdq (Cons(h,_)) = h;

fun tlq Nil = raise Empty
  | tlq (Cons(_,tf)) = tf();

from 1;
> val it = Cons(1, fn) : int seq
- tlq it;
> val it = Cons(2, fn) : int seq
- tlq it;
> val it = Cons(3, fn) : int seq
```

Consuming a Sequence

```
fun takeq (0,s)          = []
  | takeq (n,Nil)        = []
  | takeq (n,Cons(h,tf))=
              h::takeq(n-1,tf());
```

`takeq` takes an integer n and a sequence s and produces a list of the first n elements of s

```
fun squares Nil :int seq = Nil
  | squares (Cons(h,tf)) =
      Cons(h*h,fn () => squares (tf()));
```

```
- squares (from 1);
> val it = Cons(1, fn) : int seq
- takeq (5,it);
> val it = [1, 4, 9, 16, 25] : int list
```

Joining Two Sequences

```
fun appendq (Nil, yf)  = yf
  | appendq (Cons(h,tf),yf) =
      Cons(h, fn() => appendq(tf(),yf));
```

```
appendq: 'a seq * 'a seq -> 'a seq
```

If the first sequence is infinite,

`appendq(x,y) = x`

```
fun interleave (Nil, yf) = yf
  | interleave (Cons(h,tf),yf) =
      Cons(h,fn() => interleave(yf,tf()));
```

Functionals for Lazy Lists

```
fun mapq f Nil  = Nil
  | mapq f (Cons(h,tf)) =
     Cons(f h, fn() => mapq f (tf()));

- mapq (fn x => x*x) (from 1);
> val it = Cons(1, fn) : int seq
- takeq(5,it);
> val it = [1, 4, 9, 16, 25] : int list

fun filterq p Nil = Nil
  | filterq p (Cons(h,tf)) =
     if p h then
      Cons(h,fn() => filterq p (tf()))
     else filterq p (tf());
```

Searching Infinite Trees

Many problem solving applications involve searching a very large search space, structured as a tree.

Usually the tree is not available in its entirety, but the nodes are generated as needed.

We represent such a tree as a function of type

$$\texttt{next} : \alpha \to \alpha\ \texttt{list}$$

Different search strategies may be appropriate.

- *Depth First Search* searches the entire sub-tree rooted at a before proceeding to its sibling.
- *Breadth First Search* searches all nodes at a given level before proceeding to the next level.

Depth First Search

Fast, when it works

Requires space proportional to the *height* of the tree.

If the tree is infinite, it may not find a solution even if one exists.

```
fun depth next x =
 let fun dfs []     = Nil
       | dfs (h::t)=
           Cons(h,fn() => dfs((next h)@t))
 in dfs [x] end;
```

```
fn : ('a -> 'a list) -> 'a -> 'a seq
```

Breadth First Search

Guaranteed to find a solution in finite time, if one exists.

Finds the nearest solution first.

Requires space proportional to the *size* of the tree.

```
fun breadth next x =
 let fun bfs []      = Nil
       | bfs (h::t)=
           Cons(h,fn() => bfs(t@(next h)))
 in bfs [x] end;
```

```
fn : ('a -> 'a list) -> 'a -> 'a seq
```