Can We Represent Infinite Lists?

Certain collections are infinite, like the set of all natural numbers. We may try

```
fun all_numbers n =
        n :: (all_numbers (n+1))
```

which has type

```
  int -> int list
```

but

```
all_numbers 1
```

does not terminate and hence produces no result.

Lazy Evaluation

Key idea:

- evaluate lists one element at a time
- generate an element only when needed
- the list tail is thus not fully evaluated yet

We may thus attempt

```
fun all_numbers n =
      n :: (fn () => all_numbers (n+1))
```

but this is not quite type correct.

Lazy Lists


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

We define `hd`, `tl` with types

```
hd: 'a lseq -> 'a   tl: 'a lseq -> 'a lseq
```

```
exception Empty
fun hd Nil = raise Empty
|   hd (Cons (x,_)) = x
fun tl Nil = raise Empty
|   tl (Cons (_,sf)) = sf ()
```

We can now define a sequence containing all numbers:

```
fun numbers n =
   Cons(n, fn () => numbers (n+1))
... fn int -> int lseq
```

Converting a List to a Lazy List

We want functionality

```
'a list -> 'a lseq
```

which can be accomplished by

```
fun list2lseq [] = Nil
|   list2lseq (x::xs) =
        Cons (x, fn () => list2lseq xs)
```

Extracting a List from a Lazy List

- we cannot extract all elements
- but would like to extract first n elements

We thus aim at functionality

```
int -> 'a lseq -> 'a list
```

which can be accomplished by

```
fun take 0 ls = []
|   take n Nil = raise Empty
|   take n (Cons(x,sf)) =
      x :: (take (n-1) (sf ()))
```

Running

```
take 7 (numbers 2)
```

thus gives us

```
 [2,3,4,5,6,7,8]
```

Map for Lazy Sequences

Recall that `map` has type

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

We want to define `lmap` with type

```
('a -> 'b) -> 'a lseq -> 'b lseq
```

which can be accomplished by

```
fun lmap f Nil = Nil
|   lmap f (Cons(x,sf)) =
      Cons(f(x),
           fn () => lmap f (sf ()))
```

We can now run

```
take 7
    (lmap (fn (x) => x * 2)
          (numbers 1));
```

```
 [2,4,6,8,10,12,14] : int list
```

Filter for Lazy Sequences

Similarly, we want to define `lfilter` with type

```
('a -> bool) -> 'a lseq -> 'a lseq
```

which can be accomplished by

```
fun lfilter p Nil = Nil
|   lfilter p (Cons(x,sf)) =
      if p(x)
      then Cons(x, fn () =>
                     lfilter p (sf ()))
      else lfilter p (sf ())
```

We can now run

```
take 7
    (lfilter (fn x => x mod 3 = 0)
             (numbers 1));
```

```
 [3,6,9,12,15,18,21]
```

Non-Fair Merge

Recall the append function

```
fun append(nil,ls)    = ls
|   append(n::ns,ls) = n::append(ns,ls);
```

We may want to define

```
fun lappend Nil ls = ls
|   lappend (Cons(x,sf)) ls =
        Cons (x, fn () =>
                  lappend (sf ()) ls)
'a lseq -> 'a lseq -> 'a lseq
```

but when running

```
take 7
     (lappend (numbers 100)
              (numbers 1));
```

the second list is ignored:

```
  [100,101,102,103,104,105,106]
```

Fair Merge

Instead, we repeatedly swap the two lists:

```
fun interleave Nil ls = ls
|   interleave (Cons(x,sf)) ls =
      Cons(x, fn () =>
                interleave ls (sf ()))
```

which still has type

```
'a lseq -> 'a lseq -> 'a lseq
```

and when running

```
take 7
     (interleave (numbers 100)
                 (numbers 1));
```

we do get the desired alternation:

```
 [100,1,101,2,102,3,103]
```

Sieve of Eratosthenes


```
val primes =
  let fun sieve (Cons(p,sf)) = let
        fun p_not_div x = (x mod p > 0)
         in Cons(p, fn () =>
                    sieve (lfilter
                                p_not_div
                                (sf ())))
        end in sieve (numbers 2) end
```

```
- primes;
val it = Cons (2,fn) : int lseq
```

```
- take 10 primes;
val it = [2,3,5,7,11,13,17,19,23,29]
  : int list
```

Giving Change Lazily


```
fun mk_change coin_vals amount = let
    (* sol is solution currently built;
       sf () builds rest of solutions *)
  fun chg _  0 sol sf = Cons(sol,sf)
  |   chg [] n  _  sf = sf ()
  |   chg (cv1::cvs) n sol sf =
        if n < 0 then sf ()
        else chg (cv1::cvs) (n - cv1)
                 (cv1::sol)
                 (fn () =>
                     chg cvs n sol sf)
   in chg coin_vals amount []
          (fn () => Nil)
  end
```

```
int list -> int -> int list lseq
```

Example Changes, I


```
val cg1 = mk_change [5,2] 16
```

We can extract two solutions:

```
take 2 cg1;
```

```
[[2,2,2,5,5],[2,2,2,2,2,2,2,2]]
```

but is there a third?

```
take 3 cg1;
```

```
 uncaught exception Empty
```

Example Changes, II


```
val cg2 = mk_change [25,10,5,1] 46
```

displays the first solution:

```
Cons ([1,10,10,25],fn) : int list lseq
```

We can see that there are 39 solutions:

```
- take 39 cg2;
val it = [[1,10,10,25],[1,5,5,10,25],..]
```

but not 40 solutions:

```
- take 40 cg2;
  uncaught exception Empty
```