# Infinite Data

Michael P. Fourman

February 2, 2010

## 1 Lazy evaluation

When we apply a function in ML, the argument is evaluated *before* we start executing the body of the function. Suppose `f` is declared as `fun f x =` $B$, where $B$ is the *body* of the declaration. To compute the value of `f` $E$, we first compute the value, $v$, of $E$, and then substitute $v$ for the formal parameter, `x`, in $B$. This is called *call-by-value* evaluation order. There are other possibilities.

*Call-by-name* evaluates `f` $E$ by first substituting $E$ for `x` in $B$, and then evaluating the resulting expression. Certain built-in functions, such as addition, need special treatment; to evaluate $E_1 + E_2$ we first need the values of $E_1$ and $E_2$.

Consider, for example, the declaration `fun K x y = x` `K` $n$ is the constant function that always returns $n$, when applied to any argument. In ML, `K 0 (fib 30)` is evaluated by first evaluating `fib 30`, and then returning the result, 0. For this example, call-by-name would be more efficient as it would avoid the unnecessary evaluation of `fib 30`.

Consider the function `fun sqr x = x * x`. To evaluate `sqr(sqr 2)`, call-by-value performs the following steps:

```
sqr(srq 2) = sqr(2 * 2)
           = sqr 4
           = 4 * 4
           = 16
```

Whereas, for call-by-name, we have:

```
sqr(sqr 2) = (sqr 2) * (sqr 2)
           = (2 * 2) * (2 * 2)
           = 4 * (2 * 2)
           = 4 * 4
           = 16
```

The evaluation of `sqr 2` is needlessly repeated; call-by-value is a better strategy for this example.

*Call-by-need* is a variant of call-by name that only evaluates an argument when its value is needed, and also ensures that each argument is evaluated at most once. Call-by-need, which is also called *lazy evaluation* appears to combine the advantages of the other two rules. However, call-by-value is expensive to implement, and, more important, it is difficult to reason about programs in a language using this rule. Nevertheless, it is sometimes useful. In this note, we see how to obtain the effects of call-by-name and call-by-need in ML.

## Delayed evaluation

Sometimes, call-by-value is inappropriate; indeed we could not write any significant program if all our constructs used the call-by-value rule. The factorial function

```
fun fact n = if n = 0 then 1 else n * fact (n-1)
```

depends on the fact that, when we evaluate `fact 0` the expression `n * fact (n-1)` is not evaluated. If we attempted to replace the conditional expression with a function, `ite` (if-then-else),

```
fun ite(c,t,e) = if c then t else e
```

and used this in the declaration of factorial

```
fun fact n = ite( n = 0, 1, n * fact (n-1) )
```

Evaluation of `fact 0` would never terminate.

The conditional expression provides one instance of call-by-name evaluation in ML, the boolean operators `andalso` and `orelse` are also evaluated using the call-by-name rule. More complex conditions may be expressed using pattern-matching, which also has the effect of call-by-name; only one of the expressions on the right of a match is evaluated. However, if we want to abstract a common pattern of computation that relies (for correctness or efficiency) on call-by-name, we need a new idea.

We use a slightly artificial example to make our discussion concrete. Suppose we want to determine whether four values $(a(w), b(x), c(y), d(z))$ form an increasing sequence,

$$a(w) < b(x) \ \& \ b(x) < c(y) \ \& \ c(y) < d(z)$$

(Such a test might occur in an implementation of a card game.) A straightforward ML expression for this test is

```
a w < b x andalso b x < c y andalso c y < d z.
```

If this pattern of computation is repeated in different parts of our code, and if it might be changed to modify the rules of the game, then we would like to write it once, and for all, and “call” it when needed. A simple function taking the values as parameters, would lose the benefits of call-by-name. Instead we pass *delayed computations* as parameters; the idea is that instead of computing a value and passing it as a parameter, we pass a function that may be evaluated to return the value, if needed. We define a function `check`:

```
fun check(e,f,g,h) = e() < f () andalso f() < g () andalso g() < h()
```

A call to `check` has the form

```
check(fn _ => a w, fn _ => b x, fn _ => c y, fn _ => d z);
```

The trick is to replace any expression $e$ whose evaluation is to be delayed by the expression `fn _ =>` $e$. At the point where we realise the value is indeed required we can force the evaluation by applying the argument to `()`. The functions passed as parameters are called *suspensions.* A suspension represents a computation that may be performed (by applying the function to a value) if needed. Using suspensions in this way mimics call-by-name evaluation.

If evaluation of the functions, $a, b, c, d$, is expensive, we might still worry that $b$ and $c$ may be evaluated twice for the same argument. We could modify the body of `check` to avoid this, but we can achieve the same end by modifying or representation of a computation. This is another application of the use of references to secretly change a data representation. Instead of passing a function as a suspension, we pass a reference to that function. Instead of applying the function ourselves, we call a function `$` that applies the function and replaces it by a more efficient implementation

```
fun $ s = let val v = !s () in s := (fn () => v) ; v end
```

We can then write and call `check` as follows:

```
fun check(e,f,g,h) = $e < $f andalso $f < $g andalso $g < $h

check(ref(fn _ => a w),
      ref(fn _ => b x),
      ref(fn _ => c y),
      ref(fn _ => d z));
```

This provides lazy evaluation: $b$ and $c$ will each be applied at most once in any call of `check`.

## Streams

This trick of using suspensions can be used to build streams. These are lists where we only evaluate as much of the list as we need. The 'need' here is driven by functions like `hd` and `tl` that interrogate the structure of the list. We can only ever examine a finite amount of a list, but we can construct infinite lists. The (infinite) part of the list that isn't examined will never be built. Let's make these vague ideas more concrete.

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

A stream is either empty (`Nil`), or consists of a non-empty list with a head and a tail. The tail is a suspension so that the evaluation of the rest of the list is delayed until it is required. Suppose we want to build a singleton list containing the integer `1` for example. The expression `Cons(1, fn _ => Nil)` achieves this. We can build an infinite list of ones as follows

```
let fun ones() = Cons(1, ones) in ones() end
```

A more ambitious example would be the infinite list of integers starting from `k`:

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

How can we extract elements from a stream? We define the following versions of `hd` and `tl` that work on streams:

```
exception Stl and Shd
fun shd(Cons(x,_))  = x
  | stl Nil = raise Stl
and stl(Cons(_,xx)) = xx ()
  | stl Nil = raise Shd
```

We can easily use these to extract any element of a stream. For example, we can take $n$ elements from a stream

```
fun take 0 s = []
  | take n s = shd s :: take (n-1) (stl s);
```

Many of the functions that can be defined on lists can also be defined on streams. The code is just slightly messier due to the need for suspensions. For example, suppose we wanted to take a stream of integers and square each element. The following code achieves this:

```
fun squares Nil = Nil
  | squares(Cons(x:int, xx)) = Cons(x*x, fn _ => squares(xx ()))
```

We can also append two streams:

```
infix 5 @@
fun Nil @@ ss = ss
  | (Cons(x, xx)) @@ ss = Cons(x, fn _ => (xx ()) @@ ss)
```

Note that we will only start evaluating the second list if the first one is finite.

The type-checker can be very helpful when writing functions that manipulate streams as it pin-points places where a stream was required but a suspension was encountered, or vice-versa. It is usually sufficient to just evaluate a suspension, or prefix an expression by `fn _ =>` to achieve the desired result.

## Functions on streams

Functions such as `map` and `filter` are useful for manipulating ordinary lists. We can also define stream versions of these functions:

```
fun smap f Nil = Nil
  | smap f (Cons(x, xx)) = Cons(f x, fn _ => smap f (xx))

fun sfilter pred Nil = Nil
  | sfilter pred (Cons(x, xx)) =
      if pred x then Cons(x, fn _ => lfilter pred (xx ()))
      else sfilter pred (xx ())
```

The definition of `sfilter` illustrates a possible criticism of our stream representation. A stream is evaluated until we have a head and a tail. Consider applying `sfilter` to an infinite lazy list where none of the elements satisfy the predicate. The function would not terminate! We might argue that we should only fail to terminate if we subsequently wished to examine the head of the result. Can you think of an alternative representation of `stream` that has this behaviour?

Once we have defined such higher-order functions we can do some amazing things. For example, here is a program to compute the stream of all prime numbers. . .

```
fun sift p = sfilter (fn n => n mod p <> 0);
fun sieve (Cons(p,pp)) = Cons(p, fn _ => sieve(sift p (pp())))
val primes = sieve (from 2)
```

## Lazy lists

The tail of a stream is a function. Each time we inspect the tail using `stl` a function call has to be repeated. This function call could be arbitrarily expensive and so we would like to avoid it if possible. We can use a reference to make the evaluation of this function lazy. Here is an definition of a lazy list:

```
datatype 'a llist =
    Nil
  | Cons  of 'a * ('a llist) ref
  | Delay of unit -> 'a llist
```

We now redefine `$` so that it takes a reference to a lazy list, and returns a lazy list. If a suspension is encountered then it is evaluated and the reference updated to remember the result.

```
fun $ (lp as ref (Delay lf)) = (lp := lf(); !lp)
  | $ (ref ll) = ll
```
We can now define the functions `lhd` and `ltl` by
```
exception Lhd and Ltl
fun lhd Nil = raise Lhd
  | lhd (Cons(x,_)) = x

fun ltl Nil = raise Ltl
  | ltl (Cons(_,lp)) = $ lp
```
We 'hide' the `Delay` constructor and the creation of the references inside the function
```
fun konz(x,xf) = Cons(x, ref(Delay xf))
```
Functions such as `@@` and `lmap` can then be defined quite simply:
```
fun Nil @@ ll2 = ll2
  | (Cons(x, lp)) @@ ll2 = konz(x, fn _ => ($ lp) @@ ll2)

fun lmap f Nil = Nil
  | lmap f (Cons(x, lp)) = konz(f x, fn _ => lmap f ($ lp))

fun lfilter p Nil = Nil
  | lfilter p (Cons(x, lp))  =
      if p x then konz(x, fn _ => lfilter p ($ lp)
             else lfilter p ($ lp)
```