Lazy and Eager Evaluation

Recall that evaluation is “call by value” in ML: before performing function application, we have to evaluate the argument. This happens even if the argument isn’t needed for the result, as in

```
fun f(y) = 1
```

Because of this, ML is called an *eager* language. Another terminology for essentially the same thing is that ML is *strict*; this means that applying `f` or any other function to an expression *exp* having no value (because *exp* fails to terminate) gives an expression that has no value.

Some functional languages use *lazy evaluation*, also known as “call by need”, meaning that an expression will not be evaluated unless its value is actually required in order to produce the result. In a lazy language, `f(`*exp*`) = 1`, even if *exp* fails to terminate — the argument to `f` doesn’t need to be evaluated to determine the result. This isn’t done by analyzing the definition of `f`, but rather by delaying evaluation of the argument until its value is required. One *can’t* analyze functions in general to see if they use their arguments — one reason (not the only one!) is that in ML, any function might be the outcome of a complicated computation rather than just text.

Not evaluating an unused argument saves time. If its evaluation doesn’t terminate, not evaluating it saves a *lot* of time!

Here is another example:

```
fun g(x,y,z) = if x<2 then y+3 else z+6
```

Depending on the value of x, either the value of y or the value of z will be required, but not both.

Interesting examples arise with long or even infinite lists used as intermediate results in computations, in cases where only part of the list needs to be computed in order to determine the required result. Computation is *demand-driven*: we only compute as much of the list as we need to get the result.

This seems very sensible. But there are some disadvantages:

- lazy evaluation is difficult to implement efficiently
- the order of evaluation can be counterintuitive. This makes debugging difficult, and doesn’t mix so well with state-change, exceptions, or input/output
- if lazy evaluation is combined with pattern matching, it seems impossible to be really lazy — in some circumstances, unnecessary evaluation may be required.

If you want lazy evaluation in ML, you can *program* it rather than relying on having it be built into the language. The rest of this note explains how.

Non-strict constructs. ML contains a few constructs that are not strict.

`if true then` *exp*1 `else` *exp*2: will not evaluate *exp*2
`true orelse` *exp*, `false andalso` *exp*: will not evaluate *exp*

Suppose

```
fun loop(x) = loop(x+1)
```

and consider

```
fn x => loop(17)
```

When applied to an argument, this function produces no result, but it is itself a value. This shows that the `fn` construct is not strict.

However, if you define

```
fun cond(x,y,z) = if x then y else z
```

then `cond` *will* be strict, like any ML function.

Note that the first of these are abbreviations:

```
true orelse exp  =  if true then true else exp
                 =  case true of true => true | false => exp
                 =  (fn true => true | false => exp)(true)
```

so the the real non-strict construct here is `fn`.

The body of a function is not evaluated until an argument is supplied. Said another way, evaluation of the body is *delayed* until an argument is supplied. Supplying an argument *forces* evaluation.

Force and delay. We can use this idea to explicitly say when expressions are to be evaluated. Suppose we have an expression *exp* : *t*. If we type it in we get its value. Now consider the function value

```
delayed_exp = fn () => exp
```

We get `delayed_exp : unit ->` *t*. This is a "package" containing *exp* which is "waiting" to be evaluated. If we need the value of *exp*, we write the function application:

```
delayed_exp()
```

which gives exactly the value that *exp* would have given originally.

Any type of argument could have been used in `delayed_exp`. We use `():unit` because the value of the argument is completely irrelevant, and values of type `unit` carry no information at all.

We can be a bit more systematic. Define

```
type 'a delayed = unit -> 'a
```

Then `delayed_exp :` *t* `delayed`. Further, define

```
fun force d = d()
```

Then `force : 'a delayed -> 'a`.

Next, we would like to define a function that does the delaying. Let's try:

```
fun delay exp = fn () => exp
```

We get `delay : 'a -> 'a delayed` which is right, and `force(delay(`*exp*`))` evaluates to the same value as *exp*, but the behaviour is wrong: `delay` is a *function* so it evaluates its argument, of course!

So `delay(loop(17))` will loop rather than evaluating to a function that will loop as soon as it is applied. This is exactly what we *don't* want.

It is in fact impossible to write `delay` in ML. It could have been provided as an abbreviation, but it isn't. So we have to write

```
fn () => exp
```

and think of it as `delay(`*exp*`)`.

From call-by-value to call-by-need. Suppose we have a function

```
fun f x = ... x ... x ...
```

where `f : s -> t`. Replace this by

```
fun f x = ... (force x) ... (force x) ...
```

which gives `f : (s delayed) -> t`. Then replace each call `f(`*exp*`)` anywhere in the program by `f(fn () =>` *exp*`)`.

This is in fact *call-by-name* rather than call-by-need. In call-by-need, an expression is evaluated at most once while here it may be evaluated more than once. For example, suppose

```
fun f x => x+x
```

If we transform as above, and then do the application `f(fn () =>` *exp*`)`, *exp* will be evaluated twice if the value of `x` is required both times it appears in the body of `f`. But this is a fine point.

Lazy datatypes. More interesting than transforming call-by-value functions to call-by-need is the ability to create datatypes like infinite lists and infinite trees. The same basic idea applies. We need ML's ability to store functions as components of data structures!

We can't store an entire infinite list explicitly, for obvious reasons. What we can do is to delay the computation of the end part of the list until it is needed.

Recall that ordinary lists are defined like this:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Lazy lists can be defined like this:

```
datatype 'a llist = lnil | lcons of 'a * ('a llist) delayed
```

Here's a function that produces an infinite list of ones:

```
fun lones() = lcons(1,lones)
```

Here's a function that produces the infinite list of integers starting from a given number:

```
fun lfrom n = lcons(n,fn () => lfrom(n+1))
```

To access the components of such lists, one can use pattern matching with explicit applications of `force` where appropriate. Or we can define:

```
fun lhd(lcons(n,_)) = n

fun ltl(lcons(_,l)) = force l

fun lnth(0,l) = lhd l
  | lnth(n,l) = lnth(n-1,ltl l)
```

Prime numbers. We can use the functions above to compute the infinite list of prime numbers.

```
fun lfromto(n,m) =
    if n>m then lnil else lcons(n,fn () => lfromto(n+1,m))

fun lupto n = lfromto(2,n)
fun divides(n,m) = ((m mod n) = 0)

fun lforall(p,lnil) = true
  | lforall(p,lcons(a,dll)) = p(a) andalso lforall(p,force dll)

fun lisprime n = lforall((fn m => not(divides(m,n))),lupto(n-1))

fun lprune(p,lnil) = lnil
  | lprune(p,lcons(a,dll)) =
        if p(a) then lcons(a,fn () => lprune(p,force dll))
        else lprune(p,force dll)

val primes = lprune(lisprime,lfrom 2)
```

To print `primes`, use the following function (you need to type cntl-C to stop):

```
fun lprint(lnil) = ()
  | lprint(lcons(a:int,dll)) =
      (print(Int.toString a); print "\n"; lprint(force dll))
```

A digression: equality types. Consider the membership test on lists:

```
fun member(x,nil) = false
  | member(x,h::t) = (x=h) orelse member(x,t)
```

We do *not* get `member : 'a * 'a list -> bool`! It can't be fully polymorphic because we need to be able to test equality of the elements in the list. (This wouldn't be possible if `'a` is instantiated to `int->int`, for example.) What we get is `member : ''a * ''a list -> bool`. The type variable `''a` can be instantiated by any type that *admits equality.*

Which types admit equality? Types like `int`, `bool` and `string` do. The type *ty* `list` does provided *ty* does. Most user-defined datatypes do. It's easier to say which types do *not* admit equality.

Function types don't admit equality: we can't check (computationally) whether or not two functions are *extensionally* equal. That is, given two functions f, g we can't check if $\forall x. f(x) = g(x)$, since the domain of quantification may be infinite. So we refuse to try.

Therefore, types that *involve* function types don't admit equality. Examples are

```
int->bool * string
(int->bool) list
int llist
```

(the latter doesn't admit equality since the value constructor `lcons` takes an argument of type `int * (int llist) delayed`, i.e. `int * unit -> (int llist)`).

There is another way to write functions like `member`: we can pass the equality function *as a parameter*, rather than relying on ML to find a way of testing equality for us.

```
fun member'(x,p,nil) = false
  | member'(x,p,h::t) = p(x,h) orelse member'(x,p,t)
```

This has type `'a * ('a * 'b -> bool) * 'b list -> bool`. Then instead of `member(a,l)` we write `member'(a,op =,l)`. We can of course supply functions other than `=`, e.g. `<`.