Lesson 18
LAZY PROGRAMMING

July 25, 2023

Lesson Plan

1 - Lazy Evaluation

Consider the following simple SML function:

```
fun fst (x, y) = x
```

This is usually a pretty standard function to have around, for instance if you want to get out the first component of a tuple without pattern matching (for instance, if you're doing a pipe chain).

Question: What is the behavior of the expression `fst (1, 1 div 0)`?

Answer: It raises `Div`.

Recall that this is due to the fact that SML is a **eagerly evaluated** language, and the arguments to functions are always evaluated to values, if possible, before stepping into the body of the function.

But this is kind of silly, right? We wanted the first component of this tuple. It shouldn't matter whether the second component raises an exception – our code was only concerned with the first part.

The same thing would happen even if the second component didn't loop forever or raise an exception, but instead took a very long time! For instance, `fst (1, horribleComputation 2 4)` would similarly take a long time. [1]

[1] By my reckoning, 3 years.

Or, if we performed `map f L` on a list, but only used the first few elements.

This doesn't stop the fact that the `map` function always fully commits, and applies the function `f` to *each element* of the input list. It doesn't know anything about whether the result will be used, it's simply following the rules of eager evaluation.

The theme is the same – we want not to do work if we can avoid it. We want to only have to do a computation if we *have* to. In other words, we want to be **lazy**.

Def **Lazy evaluation** is a schema of evaluation opposite to that of eager evaluation. In lazy evaluation, instead of evaluating expressions at val bindings and function invocations, we only evaluate expressions when their value is *needed*.

So for instance, the following:

```
val L = [999, 998, 997, 996, 995]
val res =
  List.foldl (fn (x, acc) => fact (fact (x * acc))) 1 L
```

would run almost instantly, if SML were to be a lazy language. This is because under eager evaluation, we would instantly evaluate the right-hand side of `res`, which takes approximately between ludicrously long and forever.

Under lazy evaluation, we have to do no such thing, however! We can instantly bind `res`, by binding it to the verbatim expression of its RHS.

A way to think of the conceptual difference between lazy and eager evaluation is that in an eager language, variables can only be bound to values, which cannot further be simplified. In a lazy language, **variables are bound to expressions**, which are simplified upon **forcing** the expression.

For instance, if we tried to evaluate this code in the context of the previous slide:

```
case res of
  0 => 1
| _ => 2
```

then, we would be forced to evaluate the expression that `res` is bound to, because we want to see what value it has.

Laziness seems great. You only ever need to pay for what you need – wasted computation is impossible. There's some technical details around implementation that warrant a little more special consideration, but otherwise it seems like a good deal.

The problem with laziness comes with a **lack of predictability**.

Consider the following scenario in a lazy programming language. You have implemented an API which returns to a user an `int list`.

A consumer reads your documentation, believes it's just what they need for their project, so they pull down your code and invoke your function and bind the result to a variable. They then proceed on their merry way.

120 production hours later, their web server crashes due to an unexpected error.

The list that you handed them was `[raise Div, raise Div, raise Div]`.

This might seem like a ridiculous example, but the point is that in a lazy language, all data exchange can be completely arbitrary computation!

In an eager language, if you are able to receive an output and bind it to a variable, you are guaranteed that you are holding real data. In the case of the poor consumer, they were handed a list of three ticking time bombs. These time bombs were only revealed 120 hours later, when the elements of the list were presumably actually accessed and forced.

This can make nightmarish bugs arise, because ultimately **computation is unpredictable**. You have no idea when a value will be forced, because it could happen now or many hours later.

It doesn't even need to be a list of `raise Div`, this also has implications for determining performance. You could be handed a list of expressions, all of which take quite a while to evaluate. Now, in applications where you can't necessarily afford to wait around for several seconds, you need to be careful any time you case on an expression.

You also ultimately get that functions like `map f` are constant time, and it's only when they are forced that a linear cost in `f` comes about. This makes it very difficult to give performance a rigorous treatment. [2]

[2] Lazy languages also don't have sum types. This is of absolutely no relevance to anyone in this course, but I'm obliged to say it.

Well, that's a lot of negatives. SML is an eager language anyways, so what's the big whoop? Why are we talking about lazy evaluation?

What does binding a variable to something which encodes a *computation* without actually evaluating it sound like?

Suppose we wanted to mimic something which in a lazy language looked like this:

```
val res = map f L
```

Recall the idea of a lambda function as a *suspension*:

```
val res = fn () => map f L
```

It turns out, we can mimic laziness in an eager language, just by using lambdas.

Def A **thunk**[3] is a value of type `unit -> t`, for some type `t`, also called a **suspension**.

For any expression `e`, we can turn it into a thunk by simply encasing it within a lambda expression. Recall that lambda expressions *freeze* the contents of their bodies, meaning that no evaluation happens until the lambda is given an argument!

Key Lambdas *suspend* the contents of their bodies until the function is actually called.

By convention, that lambda usually just takes in a unit argument, since we don't really care about doing anything with its input, we just want to prevent the computation.

[3] Yes, this is a technical term.

Laziness has a lot of problems, but that is only when it is *on by default*!

Something we will see, in this lecture and the next, is that language features are generally better when they are *opt-in*. Programmers should have full control over what behavior they want, and shouldn't be forced into something contentious.

The great strength of an eager language is that we can *simulate laziness*, by putting computations into a lambda, which is to be evaluated at a later time.

This way, we can take advantage of the benefits of laziness, such as not needing to compute things until we need to and not wasting computation, without suffering the disaster of having it on all the time!

2 - A Lazy Structure

In fact, this idea is so powerful that it will warrant its own module, and type.

```
signature LAZY =
  sig
    (* The type of lazy suspensions of type 'a. These are
       computations which may return an 'a.
     *)
    type 'a t

    val lazy : (unit -> 'a) -> 'a t
    val force : 'a t -> 'a
  end
```

Here, the type of `Lazy.t` is kept abstract, to signal a type-level distinction between this idea of suspended computations versus actual values.

```
structure Lazy :> LAZY =
  struct
    type 'a t = unit -> 'a

    fun lazy f  = f
    fun force f = f ()
  end
```

In reality, the type of `'a t` is exactly equivalent to `unit -> 'a`, so the `lazy` function doesn't "really" do anything.

But to the user of the library, they don't know that. The point is that this kind of conceptual separation is useful, because it helps us segment our thinking.

So for instance, in the following code:

```
val L = List.tabulate (100, fn i => lazy (fn () => i * i))
```

the produced list `L` is a `int Lazy.t list`, where the ith entry is a *suspension* of the ith square number, that can be forced at a later time. This way, we avoid computing all of the squares up front, in case that we don't actually need all of them.

This example is reasonably contrived, but lazy values can be very useful for certain kinds of computations that may be needed later but are costly to do all at once, such as when reading from files.

Usually, lazy values are also **memoized**, which means forcing them only ever computes the expression once, and then saves the result for future forces. This implementation omits that, but this is the general practice.

3 - Infinite Data Structures

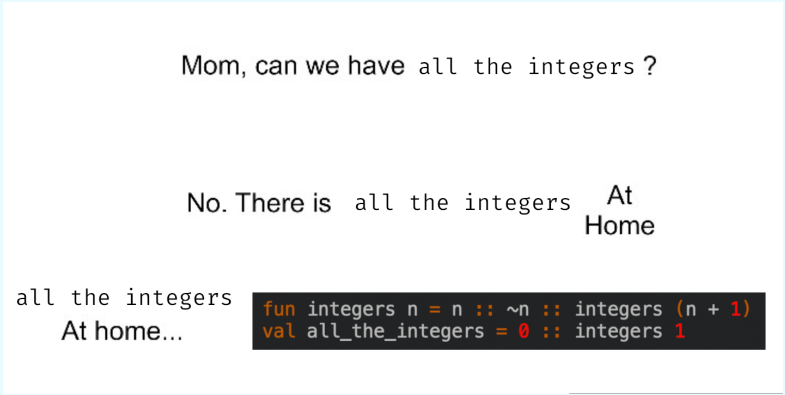
Suppose that the CEO of your company comes into the room, and tells you she wants all the integers.

You tell her, boss, I don't know if you know this, but the integers are infinite.

She says actually yes, she was perfectly aware of that, and was she going crazy or did she hear something which sounded like talking back to her.

You tell her you'll try your best.

The situation is not dissimilar to the following:



The key issue at hand here is that the integers are infinite, and thus computing `all_the_integers` loops forever.

The integers are nicely mathematically defined, however. It's a real shame that we should be limited by silly concerns like "lack of infinite amounts of space", and thus be unable to store all of the integers.

It's impossible to have infinitely many processors too, though.[4] We don't usually let silly things like physical impossibility get in our way – can we figure out something here too?

[4] *citation needed*

This leads us into an idea called **infinite data structures**.

Def An **infinite data structure** is a kind of data structure that may store infinitely many entries, without looping forever.

The memory concern is still a legitimate one, though. We can't possibly store an infinite number of numbers at once, so what can we do?

The key to encoding infinite data structures will be in using **laziness**, to compute entries only as we need them.

Our first data structure will be the **lazy list**. Recall that we could define our own type of lists via the following:

```
datatype 'a list' = Nil | Cons of 'a * 'a list'
```

which would be totally equivalent to our native type of `'a list`.

We can define lazy lists as follows:

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

Whereas a list is either empty or a value and another list, a lazy list is either empty or a value and a *suspension* of another lazy list.

What does that mean? It means that when forming a value of type `'a llist`, we don't ever need to go and come up with the lazy list that comes afterwards – we just need to provide a lambda that computes it.

By our analogy from CPS lecture, we just provide **instructions** that tell us how to make the rest of the lazy list.

Here are some examples of values of type `t llist`:

- `Nil : 'a llist`
- `Cons (1, fn () => Nil) : int llist`
- `Cons (1, fn () => Cons (2, fn () => Nil)) : int llist`
- `Cons (1, fn () => loop ()) : int llist`

Notably, the last value is indeed a value, but trying to get the second element of the list will cause an infinite loop.[5]

[5] That's how it is, sometimes.

Part of the point of lazy lists is that usually, instead of writing them down in a finite amount of space, we will construct lazy lists via recursive functions.

Let's see how we might define the natural numbers using lazy lists:

```
fun natsFrom n = Cons (n, fn () => natsFrom (n + 1))
val nats = natsFrom 0
```

Here, we define first a helper function `natsFrom`, which defines a lazy list of all the natural numbers, from some starting point. Then, we just start at 0.

Note that because the recursive call to `natsFrom` is within a thunk, there is no looping forever here! For that same reason, we don't need a base case, either.

Suppose that we were interested in tabulating a lazy list from an arbitrary function. We might define a function `tabulate` with the following specification:

`lazyTabulate : (int -> 'a) -> 'a llist`
REQUIRES: `true`
ENSURES: `lazyTabulate f` evaluates to the lazy list where the `i`th element is `f i`

```
fun lazyTabulateFrom f i =
  Cons (f i, fn () => lazyTabulateFrom f (i + 1))
fun lazyTabulate f = lazyTabulateFrom f 0
```

Let's try it on a specific use case.

```
val lazy_list = lazyTabulate (fn i => 1024 div i)
```

This lazy list purportedly contains all the results of dividing 1024 by the natural numbers.

Unfortunately, there is one thing working against us, here, which is that 0 is a natural number!

Are we OK, though, since this is a lazy list? It turns out no, because while our list is lazy, attempting to bind `lazy_list` immediately raises `Div`.

It turns out that our notion of lazy lists is indeed lazy, but not lazy enough.

Even if there is an element in the lazy list that should raise `Div` upon being forced, merely constructing the value of `lazy_list` never expresses intent to force that element!

Or, put another way, we **never once** explicitly said we wanted to look at the elements of the lazy list, so why should constructing the lazy list raise an exception on us?

In the next section, we'll see an improvement on lazy lists we call **streams**, which will help with this issue.

4 - Maximal Laziness and Streams

To solve our problem, we need a new kind of data structure.

Def A **maximally lazy** data structure is one which does not compute any element until it is absolutely needed.[6]

We will develop a new type of lazy list called a `stream`, which side steps these issues by being maximally lazy. We can define a stream via two mutually recursive data types:

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

Note that we use the `and` keyword here, which allows two mutually recursive types to see each other, irrespective of the order in which both are declared.

[6] There will be a formal definition of this on the homework which is slightly different, but this definition will suffice to get us through the intuition.

Conceptually, what's the idea behind a stream and a front?

Key **A stream is a delayed front.**
Key **A front is an exposed stream.**

A stream is opaque – the only thing you can do with a stream is force it. This solves our issue from lazy lists, as instead of making lazy lists (which have the first element forced by default), we make streams.

A front is a stream which the user has **expressed intent to force**. The only way to obtain a front is to deliberately try to force a stream. The front has the actual pattern-matching data associated with it though, which may produce another stream.

We will have another structure for dealing with streams. This one will similarly have another an abstract type hiding the inner implementation.

```
signature STREAM =
  sig
    type 'a stream
    datatype 'a front = Empty | Cons of 'a * 'a stream

    (* more... *)
  end
```

These types are implemented exactly the same as the `'a stream` and `'a front` we saw earlier! We just hide the definition of `'a stream` so it simply operates as an abstract idea of some suspended list, which may be forced to look at its contents as a `'a front`.

We will now look at some of the residents of the Stream structure. There are more functions, which are available at the online 150 documentation, but we will not get to all of them.

For the rest of the lecture, assume we are working within the Stream structure, defined as:

```
structure Stream :> STREAM =
  struct
    datatype 'a stream = Stream of (unit -> 'a front)
    and 'a front = Empty | Cons of 'a * 'a stream

    (* ... *)
  end
```

The most essential functions to working with streams are `delay` and `expose`, which mediate the relationship between `stream`s and `front`s.

```
delay : (unit -> 'a front) -> 'a stream
```
REQUIRES: `true`
ENSURES: `delay f` creates the stream associated with the thunk `f`.

```
expose : 'a stream -> 'a front
```
REQUIRES: `true`
ENSURES: `expose S` forces the stream `S`, revealing the resulting front. This does not necessarily need to produce a value!

We can implement them as so.

```
fun delay f = Stream f
```

```
fun expose (Stream f) = f ()
```

We will use these two functions as helpers when we write functions that operate on streams.

Since our `'a stream` type is abstract, we have to provide these helpers, as users of the structure cannot apply the `Stream` constructor directly, or force the thunk. It ends up that using the `delay` and `expose` functions will make it very explicit when we are doing work and when we are suspending.

Let's try to do the natural numbers again, but this time with streams.

We will usually construct streams via **mutual recursion**, which is functions which are allowed to call each other. This is again achieved with the `and` keyword.

```
fun nats  (i : int) : int stream = delay (fn () => nats' i)
and nats' (i : int) : int front  = Cons (i, nats (i + 1))
```

Alternatively, we could have implemented it with a single function:

```
fun nats i = delay (fn () => Cons (i, nats (i + 1)))
```

but it's generally cleaner to separate into two functions, one of which produces a stream, and one of which produces a front. Remember, types guide structure.

We can do `tabulate` again, too, but for streams!

```
fun tabulateHelp  f i = delay (fn () => tabulateHelp' f i)
and tabulateHelp' f i = Cons (f i, tabulateHelp f (i + 1))

fun tabulate f = tabulateHelp f 0
```

Check your understanding Make sure you understand why this version of `tabulate` is more lazy than `lazyTabulate`. What does `tabulate f` return, versus `lazyTabulate`?

Like most data structures, we can write a `map` function on streams.

```
map : ('a -> 'b) -> 'a stream -> 'b stream
```
REQUIRES: `f` is total
ENSURES: `map f S` evaluates to the same stream as `S`, but with `f` applied to each element of the stream

```
map' : ('a -> 'b) -> 'a front -> 'b front
```
REQUIRES: `f` is total
ENSURES: `map' f F` evaluates to the same front as `F`, but with `f` applied to each element of the front

Here, we again define two functions, one for streams and one for fronts. Since `map` operates on a given stream, the corresponding `map'` also takes in a front.

```
fun map  f S = delay (fn () => map' f (expose S))
and map' f Nil = Nil
  | map' f (Cons (x, s)) = Cons (f x, map f s)
```

In this case, our mutual recursive usage of these two functions reflects a difference in intention between the two functions. When "deconstructing" a stream, the function which produces a stream simply sets up the second function – the second function is the only one which actually has license to look at the stream's elements.

The following implementation would fail to be maximally lazy, however.

```
fun map f s =
  case expose s of
    Nil => delay (fn () => Nil)
  | Cons (x, s') => delay (fn () => Cons (f x, map f s'))
```

The process of mapping this stream exposes the input stream, without any outside prompting to force the resulting stream! We want to be able to map it in a way that we never need to expose any elements, until we must.

In general, it's only safe to execute operations on a stream so long as they are locked behind a call to `delay`, so that it only occurs once the stream is exposed.

Recall that exposing a stream means to express intent to look at its elements, and will cause the computation of the next element, if any.

This is dangerous, because that computation might do anything from raise an exception, to loop forever, to take 2 months to run. Exposing a stream is a risky endeavour!

Generally, we like to work with streams that have well-defined contents, such that exposing will always eventually give us another element. We call those streams **productive**.

Def A **productive** stream `S` is one such that `Stream.expose S` always terminates with `Empty` or `Cons (x, S')`, where `S'` is again productive.

Generally, we like working with productive streams. Thus, we need to be careful when using functions like `map`, which could turn a productive stream into a no-longer productive stream.
Productivity will also help us describe the specification of more stream functions.

```
append : 'a stream * 'a stream -> 'a stream
```
REQUIRES: `S1` and `S2` are productive
ENSURES: `append (S1, S2)` is productive

```
append' : 'a front * 'a stream -> 'a front
```
REQUIRES: `S2` is productive and `F` is `Empty` or `Cons` of a productive stream
ENSURES: `append' (F, S2)` is productive

Now, we can implement our function, again employing the technique of two different functions, one for generating `stream`s and one for generating `front`s.

```
fun append  (S1, S2) =
  delay (fn () => append' (expose S1, S2))
and append' (Empty, S2) = expose S2
  | append' (Cons (x, S1), S2) = Cons (x, append (S1, S2))
```

Note that in the case where the input front is `Empty`, `append'` must expose the second stream! This is so that the types match, since `append'` wants to return an `'a front`.

This is fine, in terms of maximal laziness, though, as `append'` is a function operating on a front, meaning that it already has "license to expose" the stream that it is producing. Another way to think about it is that the only way to get to a call to `append'` is to `expose` the output stream.

One more for the road.

```
filter : ('a -> bool) -> 'a stream -> 'a stream
```
REQUIRES: `true`
ENSURES: `filter p S` is the stream of all the elements of `S` satisfying `p`

```
filter' : ('a -> bool) -> 'a front -> 'a front
```
REQUIRES: `true`
ENSURES: `filter' p F` is the front of all the elements of `F` satisfying `p`

```
fun filter  p S = delay (fn () => filter' p (expose S))
and filter' p Empty = Empty
  | filter' p (Cons (x, S)) =
      if p x then Cons (x, filter p S)
      else expose (filter p S)
```

This is an interesting case, because `filter` might produce a non-productive stream from a productive stream, even with a total function `p`!

Note a subtlety in the implementation, here, which is that `filter'` must expose the rest of the stream, to make the types match!

Key `filter` might loop forever upon exposing the next element, if it has to keep searching for an element which never satisfies the predicate

5 - Primes (Bonus)

Some people will tell you that the primes are infinite.[7]

As such, they make a *prime* candidate to study as an application of streams! We can try to hold a stream which has all of the primes at once.

We can do so using an ancient technique called the **Sieve of Eratosthenes**, which tries to find all prime numbers by marking all of the numbers which are not prime.

[7] Those people would be correct.

The idea is that as soon as we see 2, and we know that it is prime, we remember that fact and then disregard any number which is divisible by 2. We will then do that with 3, 5, etc, every prime number that we see in the future.

It can be implemented as such:

```
fun dividedBy x y = y mod x = 0
fun sieve  S = delay (fn () => sieve' (expose S))
and sieve' Empty = raise Fail "this shouldn't ever happen"
  | sieve' (Cons (x, S)) =
      Cons (x, sieve (filter (not o (dividedBy x)) S))

val natsFrom2 = tabulate (fn x => x + 2)
val primes = sieve natsFrom2
```

Thank you!