CS 251: Programming Languages Fall 2015
ML Summary, Part 5

These notes contain material adapted from notes for CSE 341 at the University of Washington by Dan Grossman. They have been converted to use SML instead of Racket and extended with some additional material by Ben Wood.

Contents

Introduction to Delayed Evaluation and Thunks

A key semantic issue for a language construct is *when are its subexpressions evaluated.* For example, in ML (and similarly in Racket and most but not all programming languages), given `e1 e2 ... en` we evaluate the function arguments `e2`, ..., `en` once before we execute the function body and given a function `fn ... => ...` we do not evaluate the body until the function is called. This rule ("evaluate arguments in advance") goes by many names, including *eager evaluation* and *call-by-value.* (There is a family of names for parameter-passing styles referred to with *call-by-....* Many are not particularly illuminating or downright confusing names, but we note them here for reference.) We can contrast eager evaluation with how `if e1 then e2 else e3` works: we do *not* evaluate both `e2` and `e3`. This is why:

```
fun iffy x y z = if x then y else z
```

is a function that *cannot* be used wherever you use an if-expression; the rules for evaluating subexpressions are fundamentally different. (We studied this a bit when considering syntactic sugar early in our exploration of Racket.) For example, this function would never terminate since every call makes a recursive call:

```
fun facty n =
  iffy (n=0) 1 (n * (facty (n-1)))
```

However, we can use the fact that function bodies are not evaluated until the function gets called to make a more useful version of an "if function":

```
fun ifok x y z =
  if x then y () else z ()
```

Now wherever we would write `if e1 then e2 else e3` we could instead write

```
ifok e1 (fn () => e2) (fn () => e3)
```

The body of `ifok` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`. (Actually, as we know, ML does not have zero-argument functions – these are actually one-argument functions whose argument is of type `unit`. Since there is only one such value, `()`, we pattern-match on it in the function definition to be explicit: `fn () => ....`) So this function is correct (for non-negative arguments):

```
fun fact n =
  ifok (n=0) (fn () => 1) (fn () => n * (fact (n-1)))
```

Though there is certainly no reason to wrap ML's "if" in this way, the general idiom of using a "zero-argument" function (*i.e.*, a function with argument type `unit`) to *delay evaluation* (do not evaluate the expression now, do it later when/if the function is called) is very powerful. As convenient terminology/jargon, when we use such a function to delay evaluation we call the function a *thunk*. You can even say, "thunk the argument" to mean "use `fn () => e` instead of `e`".

Using thunks is a powerful programming idiom. It is an idiom, not a new language feature, and is not specific to ML or Racket. (When done implicitly in the language instead of using explicit thunking as we have used here, variations on this technique are sometimes called *call-by-name*. We will use "thunking" or "delayed evaluation".)

Lazy Evaluation with Delay and Force

Suppose we have a large computation that we know how to perform but we do not know if we need to perform it. Other parts of the program know where the result of the computation is needed and there may be 0, 1, or more different places. If we thunk, then we may repeat the large computation many times. But if we do not thunk, then we will perform the large computation even if we do not need to. To get the "best of both worlds," we can use a programming idiom known by a few different (and perhaps technically slightly different) names: *lazy evaluation*, *call-by-need*, *promises*. The idea is to use mutation to remember the result from the first time we use the thunk so that we do not need to use the thunk again – it will be called at most once.

One simple implementation in ML would be:

```
signature PROMISE =
sig
    (* Type of promises to produce an 'a. *)
    type 'a t
    (* Make a promise for a thunk. *)
    val delay : (unit -> 'a) -> 'a t
    (* If promise not yet forced, call thunk and save.
       Return saved thunk result. *)
    val force : 'a t -> 'a
end

structure Promise :> PROMISE =
struct

(* Before a promise has been forced, it is just a thunk.  After it has
   been forced, it is a value. *)
datatype 'a promise = Thunk of unit -> 'a
                    | Value of 'a

(* Hide limited mutation inside ADT. *)
```

```
type 'a t = 'a promise ref

(* Wrap the thunk to make a promise. *)
fun delay th = ref (Thunk th)

(* If the promise is already a value, return it.
   Otherwise, call the thunk and save and return its result. *)
fun force p =
  case !p of
      Value v => v
    | Thunk th => let val v = th ()
                      val _ = p := Value v
                  in v end
end
```

We represent a *promise* by a reference cell holding either a Thunk carrying an unevaluating thunk (if the promise has never been "forced") or a Value carrying the value resulting from the evaluation of the thunk when the promise was first "forced".

We can create a thunk and pass it to delay. This returns a Thunk carrying the unused thunk we provided. Then force, if it sees the promise has not yet been forced, calls the thunk and then uses mutation to change the reference to hold the result of the thunk, wrapped as a Value. That way, any future calls to force on the same promise will not repeat the computation. Ironically, while we are using mutation in our *implementation*, this idiom is quite error-prone unless the thunk passed to delay does not have side effects or rely on mutable data, since those effects will occur at most once and it may be difficult to determine when the first call to force will occur.

Consider this silly example where we want to multiply the result of two expressions e1 and e2 using a recursive algorithm (of course you would really just use * and this algorithm does not work if e1 produces a negative number):

```
fun mult 0 y = 0
  | mult 1 y = y
  | mult x y = y + (mult (x-1) y)
```

Now calling mult e1 e2 evaluates e1 and e2 once each and then does 0 or more additions. But what if e1 evaluates to 0 and e2 takes a long time to compute? Then evaluating e2 was wasteful. So we could thunk it:

```
fun mult 0 ythunk = 0
  | mult 1 ythunk = ythunk ()
  | mult x ythunk = (ythunk ()) + (mult (x-1) ythunk)
```

Now we would call mult e1 (fn () => e2). This works well if e1 evaluates to 0, fine if e1 evaluates to 1, and terribly if e1 evaluates to a large number. After all, now we evaluate e2 on every recursive call. So let's use delay and force to get the best of both worlds:

```
open Promise
mult e1 let val p = delay (fn () => e2)
        in (fn () => force p) end
```

Notice we create the delayed computation once before calling mult, then the first time the thunk passed to `mult` is called, `force` will evaluate `e2` and remember the result for future calls to `force p`. A simpler approach to rewrite `mult` to expect a promise rather than a thunk:

```
fun mult 0 _ = 0
  | mult 1 ypromise = force ypromise
  | mult x ypromise = (force ypromise) + (mult (x-1) ypromise)

mult e1 (delay (fn () => e2))
```

Some languages, most notably Haskell, use this approach for all function calls, *i.e.*, the semantics for function calls is different in these languages than in ML, Racket, and most others: If an argument is never used it is never evaluated, else it is evaluated only once. This is called *call-by-need* whereas all the languages we will use are *call-by-value* (arguments are fully evaluated before the call is made).

a.k.a. Suspensions

Promises are also known as *suspended computations* or *suspensions.* SML/NJ (but not the SML language in general) includes a library just like the above, using this alternative terminology: `http://www.smlnj.org/doc/SMLofNJ/pages/susp.html`

Streams

A stream is an infinite sequence of values. We obviously cannot create such a sequence explicitly (it would literally take forever and consume infinite storage resources), but we can create code that knows how to produce the infinite sequence and other code that knows how to ask for however much of the sequence it needs.

Streams are very common in computer science. You can view the sequence of bits produced by a synchronous circuit as a stream, one value for each clock cycle. The circuit does not know how long it should run, but it can produce new values forever. The UNIX pipe (`cmd1 | cmd2`) is a stream; it causes `cmd1` to produce only as much output as `cmd2` needs for input. Web programs that react to things users click on web pages can treat the user's activities as a stream — not knowing when the next will arrive or how many there are, but ready to respond appropriately. More generally, streams can be a convenient division of labor: one part of the software knows how to produce successive values in the infinite sequence but does not know how many will be needed and/or what to do with them. Another part can determine how many are needed but does not know how to generate them.

There are many ways to code up streams; we will take the simple approach of representing a stream as a thunk that when called produces a pair of (1) the first element in the sequence and (2) a thunk that represents the stream for the second-through-infinity elements. (Actually, we will need to tweak this slightly to satisfy the ML type system, but let's try pairs first, as it is instructive.)

We could (almost) take the first 3 elements of a stream as `v1`, `v2`, and `v3` as follows:

```
let val (v1,s1) = s ()
    val (v2,s2) = s1 ()
    val (v3,s3) = s2 ()
in  ...  end
```

Consider the type of `s2`. Since it is called on `()` and matched with `(v3,s3)`, it must be a function of type `unit -> 'a * ...`, where `v3 : 'a`. But if we continue backwards, we see that `s1` must have type `unit -> ('a * (unit -> ('a * ...)))` and `s` must have type

```
unit -> ('a * (unit -> ('a * (unit -> ('a * ...))))
```

Already, it is clear that we cannot give a single type to all the stages of the same stream, since each earlier stage of the stream has a type that is one layer larger. However, if we continue forward, deeper into the stream, it is clear that the `...` here is not accidental: an infinite stream results in an infinitely large type. Furthermore, the types of each stage of the stream are related: the type appears to be (infinitely) recursive. Intuitively, we would like to type streams with:

```
type 'a stream = unit -> ('a * 'a stream)
```

Unfortunately, the ML `type` alias mechanism does not support recursive types, but, as we have seen with lists, `datatype` does. We therefore use the following types to describe streams:

```
datatype 'a scons = Scons of 'a * (unit -> 'a scons)

type 'a stream = unit -> 'a scons
```

The type `unit -> 'a scons` describes a stream: a thunk that returns a pair of element and another stream (*i.e.*, thunk) wrapped in a constructor of the `'a scons` type. It is a bit odd to have a datatype (a "one-of" or "sum" type) with only one constructor, but the key here is that datatype bindings allow recursive type definitions.

Defining thunks to produce streams typically uses recursion. Here are three examples:

```
fun ones () = (1,ones)
val rec ones = fn x => Scons (1, ones)

val nats =
    let fun f x = (x, fn () => f (x + 1))
    in fn () => f 0 end

val powers2 =
    let fun f x = (x, fn () => f (x * 2))
    in fn () => f 1 end
```

Given this encoding of streams and a stream `s`, we could get the first, second, and third elements via:

```
let val Scons (v1,s1) = s ()
    val Scons (v2,s2) = s1 ()
    val Scons (v3,s3) = s2 ()
in  ...  end
```

Usually it is odd to pattern-match a datatype constructor in a val-binding, but here, with only one constructor in the datatype, it is guaranteed to succeed.

We could write a higher-order function that takes a stream and a predicate function and returns the index of the first stream element for which the predicate returns true:

```
fun firstindex stream f =
    let fun consume stream ans =
            let val Scons (v,s) = stream ()
            in
                if f v then ans else consume s (ans + 1)
            end
    in
        consume stream 0
    end
```

As an example, `firstindex powers2 (fn x => x=16)` evaluates to 3.

All the streams above can produce their next element given at most their previous element. So we could use a higher-order function to abstract out the common aspects of these functions, which lets us put the stream-creation logic in one place and the details for the particular streams in another. This is just another example of using higher-order functions to reuse common functionality:

```
fun make_stream step init =
    let fun f x = Scons (x, fn () => f (step x))
    in fn () => f init end
val nats = make_stream (fn x => x + 1) 0
val powers2 = make_stream (fn x => x * 2) 1
```

Memoization

An idiom related to lazy evaluation that does not actually use thunks is *memoization*. If a function does not have side effects, then if we call it multiple times with the same argument, we do not actually have to compute the answer more than once. Instead, we can look up what the answer was the first time we called the function with the argument.

Whether this is a good idea or not depends on trade-offs. Keeping old answers in a table takes space and table lookups do take some time, but compared to repeating expensive computations, it can be a big win. Again, for this technique to be *correct* in the first place requires that, given the same arguments, a function will always return the same result and have no side effects. So being able to use this *memo table* (*i.e.*, do memoization) is yet another advantage of avoiding mutation.

To implement memoization we do use mutation: Whenever the function is called with an argument we have not seen before, we compute the answer and then add the result to the table (via mutation).

As an example, let's consider 3 versions of a function that takes an n and returns the nth Fibonacci number. A natural recursive definition is:

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib (n-1) + fib (n-2)
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `fib 30`, and `fib 40` takes about a thousand times longer than that... We have seen a tail-recursive approach that "counts up" and remembers previous answers during one call, *e.g.*:

```
fun fibtail 0 = 1
```

```
  | fibtail 1 = 1
  | fibtail x =
    let fun f (acc1, acc2, y) =
            if y=x
            then acc1 + acc2
            else f (acc1 + acc2, acc1, y + 1)
    in f (1,1,3) end
```

This takes linear time, but requires a quite different approach to the problem. With memoization we can turn `fib` into an efficient algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you should learn a bit about in CS 231. Below is the version that does this memoization. (We include the definition of an `assoc` function which we have written a couple times previously – it looks up the value corresponding to a given key in an association list. However, this is a linear-time lookup operation. For a real implementation, it would be better to choose a data structure with better lookup times, such as a hash table.)

```
fun assoc x [] = NONE
  | assoc x ((k,v)::rest) = if k=x then SOME v else assoc x rest

val fibmemo =
    let val memo = ref []
        fun f x =
            case assoc x (!memo) of
                SOME y => y
              | NONE => let val y = (case x of
                                       0 => 1
                                     | 1 => 1
                                     | n => (f (n-1)) + (f (n-2)))
                            val _ = memo := ((x,y)::(!memo))
                        in y end
    in f end
```

It is essential that different calls to `f` use the *same* mutable memo table: if we create the table inside the call to `f`, then each call will use a new empty table, which is pointless. But we do not put the table at the top level – that would be bad style since its existence should be known only to the implementation of `fibmemo`. Closures work out very nicely here.

Why does this technique work to make computing larger Fibonacci numbers complete quickly? Because when we evaluate `f (n-2)` on any recursive calls, the result is already in the table, so there is no longer an exponential number of recursive calls. Another nice effect is that calling `fibmemo` a second time on the same number (or any smaller number!) will complete even more quickly since the answer will be in the memo-table.

For a large table, using an association list is a poor choice (try a hash table or tree, perhaps), but it is simple and suffices to demonstrate the concept of memoization.

More Approaches to Memoization

Our memoized `fib` implementation had some nice performance properties, but its code was not as clean as could be. It interleaves the simple definition of *what* `fib` computes with the implementation details of *how* to compute it efficiently. Furthermore, under this approach, every function we wish to memoize has to be converted, even if they will follow a similar pattern. In this section, we present two alternatives to help separate the *what* from the *how* of memoization.

Refactoring Memoization: False Start

Our first approach allows memoization of *any* single-argument function *without modifying the original function* (subject to the determinacy and purity requirements expressed above). It is a simple higher-order function that takes the function to memoize and returns a version wrapped up to first check the memo table before computing the result.

```
fun memotop f =
  let val mem = ref []
  in  fn x =>
        case assoc x (!mem) of
            SOME y => y
          | NONE => let val y = f x
                        val _ = mem := ((x,y)::(!mem))
                    in y end
  end
```

For example, for `fib`, we can define a memoized implementation using the original naturally recursive definition:

```
val fibtop = memotop fib
```

This approach is beautiful in the way that it fully separates the function we compute and the means of memoization, but it is *significantly less efficient* than our original memoized `fib` implementation. Since it intercepts only top-level calls to the function (and not any recursive calls), computing `fibtop n` is still $O(2^n)$ the first time. Future calls on the same input will become simple memo table lookups, but calls on any other input will require the full computation: this method fills only one entry in the memo table per top-level call (not all of 0 through n as before). Its utility is thus questionable.

OPTIONAL: An Elegant *and* Efficient Memoziation Solution.

With a simple change to how we define the functions we wish to memoize, we can support a clean separation of concerns between the functions themselves and the mechanism of memoization (or, for that matter, any other "interjection" function).

Let us begin by defining a version of `fib` in a style called *open recursion*:

```
fun fibopen fib 0 = 1
  | fibopen fib 1 = 1
  | fibopen fib n = fib (n-2) + fib (n-1)
```

The function `fibopen` takes one additional argument: a function to call for recursive calls instead of calling itself. This is a necessary change to the original clean naturally recursive definition, but compared to our first memoized implementation it is minimal and, with good naming choices, the intent remains clear. The `fibopen` function alone is not quite useful yet. Where do we get this first argument? The answer is to use a *fixed-point combinator* to implement recursion, as in the lambda calculus. Here in ML, which already supports recursion, this is much simpler than, *e.g.* the Y combinator in the untyped lambda calculus. We define a simple `fix` function:

```
fun fix f x = f (fix f) x
```

Its definition is exactly the equivalence of a fixed point (not discussed here). Notice its type:

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

It takes an `('a -> 'b) -> 'a -> 'b` function and produces an `'a -> 'b` function. Any open recursive function such as `fibopen` has a compatible type for `fix`'s first argument. By partially applying itself to its first argument, `fix` produces a function of the correct type to pass as the first argument to *its argument function* (*e.g.*, `fibopen`). Furthermore, the resulting function acts just like the one currently being evaluated...

Consider the use of `fix` as partially applied to `fibopen`:

```
val fib = fix fibopen
```

This produces a function that, when called on an integer, `n`, calls `fibopen` and either returns a result immediately in a base case of `fibopen` or calls the first argument to `fibopen` to perform recursion. The key is that this first-argument function in `fibopen` is exactly the partial application of `fix` to `fibopen` — or, a function that acts identically to the one we bound to `fib` imeediately above! Thus we have a recursive implementation of `fib`.

Open recursion is the key to implementing recursion via fixpoint combinators (as in the lambda calculus or elsewhere), but it also appears in key semantic features of object-oriented languages: *late binding* and *dynamic method dispatch*. (We will return to these later in the course.) Open recursion can be used to support a pattern sometimes called *function inheritance*. (Again, the similarity with object-oriented terminology is no accident.)

Now, to introduce memoization, we can define a completely independent memoization function, also in open recursive style. The `make_memo()` thunk, defined below, produces a new open recursive memoization function when call.

```
fun make_memo () =
  let val mem = ref []
      (* In open recursive form: *)
      fun memf f x =
        case assoc x (!mem) of
            SOME v => v
          | NONE => let val v = f x
                        val _ = mem := ((x,v)::(!mem))
                    in v end
  in memf end
```

We could call the result of `make_memo ()` recursively with `fix`, but, with an empty memo table to start, it would never terminate on any input. It is useful only in composition with some other computation, as with `fibopen`:

```
val fibmemo = fix (make_memo () o fibopen)
```

Now, when it finds no value in its memo table, the resulting `memf` function will call `fibopen` to get a result. If `fibopen` does not hit the base case, it will use its first argument to perform recursion. Its first argument is a function that first checks the memo table for its argument before falling back to `fibopen`!

Thus, every recursive level of a call to `fibmemo` inspects the memo table before resorting to the original definition. This approach achieves the performance benefit of the memo table while organizing code in a way that keeps the *separate concerns* of Fibonacci computation, memoization, and their composition *entirely separate code*. It is thus easy to plug in other computations and other interjections (other than memoization).

If desired (it may make more sense while exporing the idea initially), it is also possible to interleave the fixpoint computation with the memoization code as follows, while still keeping the Fibonacci computation separate:

```
fun memoize f = (* difference: f as arg to memo construction *)
  let val mem = ref []
      fun memf x = (* difference: capture f in closure *)
        case assoc x (!mem) of
            SOME v => v
          | NONE => let val v = f memf x (* difference: explicitly fix *)
                        val _ = mem := ((x,v)::(!mem))
                    in v end
  in f memf end

val fibmemo' = memoize fibopen
```

Finally, as an example of another "analysis" to interleave with open recursive computations, consider logging:

```
fun log name arg_to_s result_to_s f =
  let fun wrap indent x =
        let val _ = print (indent ^ name ^ " " ^ arg_to_s x ^ "\n")
            val v = f (wrap ("  " ^ indent)) x
            val _ = print (indent ^ "=> " ^ result_to_s v ^ "\n")
        in v end
  in wrap "" end

val fiblog = log "fib" Int.toString Int.toString fibopen
```

Try it to see what happens! Since it requires another argument to be passed for indentation, this version of logging cannot quite be used with our `fix` as is. Removing the indentation would suffice to allow it. Some additional plumbing could support the current logging function, but we will end our discussion here. Deeper discussions of this topic (using other languages) can be found here:

- `https://www.cs.utexas.edu/~wcook/Drafts/2006/MemoMixins.pdf`. Sections 1, 2, and 2.2 should be accessible to a motivated CS 251 reader. Additonal sections require some extra background beyond what 251 covers, but this background could be great material for a final project...
- `http://matt.might.net/articles/implementation-of-recursive-fixed-point-y-combinator-in-javascript-for-memoization/`
- `http://adriansampson.net/blog/functioninheritance.html`

Integrating Lazy Evaluation into the Language

We have discussed some *programming idioms* that exploit the idea of lazy evaluation by explicitly delaying computation with thunks. Some languages (most notably Haskell) integrate lazy evaluation directly into the language at the level of its semantics. In this section, we consider the implications of this decision briefly.

Essentially, every variable binding (including function calls, where parameters are bound to arguments) in a lazy language such as Haskell *implicitly* creates promises of its arguments rather than evaluating them eagerly. Uses of those bindings then implicitly force these promises. Only if the arguments are needed are they actually evaluated. As Haskell is a functional language similar to ML, this laziness cascades very clearly. Given `f (g x)`, the expression producing the value to be bound to `x` might not be evaluated until the function call `g x` is evaluated (if this is the first time `x` has been used). But `g x` itself will not be evaluated unless needed by `f`, and the call to `f` will not be evaluated unless its result is actually needed, and so on.

This pattern is powerful. For example, streams become much less distinguishable from lists (think infinite lists). This pattern also introduces some complications in reasoning about program behavior. If *everything* is lazily evaluated, it is difficult to reason about the order in which expressions will get evaluated (or whether they will be evaluated at all). In a *pure* setting, without side effects, this is manageable: order of evaluation does not matter. However, just as discussed with promises and memoization, side effects throw a wrench in the works, since their order typically matters. For this reason, side effects are generally banned (or at least managed very carefully) in languages that use lazy evaluation. Regardless of whether side effects are present, laziness can complicate control over space usage. In a language with eager evaluation, it is relatively clear when environments will become unreachable (and thus available for GC). With lazy evaluation, the difficulty in predicting evaluation order means that it is difficult to predict how long large environments linger, still referenced by unevaluated thunks.

We will not have time to explore laziness (including the very interesting mechanism used to deal with side effects in Haskell) beyond this cursory treatment in this course.

Please see Harper, *Programming in Standard ML*, Chapter 15 for more extensive discussion of `lazy` data structures and functions in SML/NJ: `http://www.cs.cmu.edu/~rwh/smlbook/book.pdf`

If you are curious about Haskell, see `https://www.haskell.org/` or `http://learnyouahaskell.com/`.

Any topics extending from here would make great topics for final projects.