Delayed Evaluation

Delayed, or Lazy, evaluation is the evaluation of the expression only when needed. It is often considered in relation to the arguments to a function call, the idea being that the arguments would not be evaluated until their values are needed. Some languages like Haskell promote this idea to every location, even allowing infinitely long lists that are only evaluated up to where they are needed.

There are two important features needed for delayed evaluation:

- The expression must not be evaluated until it is needed.
- The expression must only be evaluated once. So the value must be stored upon first evaluation and recovered each time as requested.
- If the expression raised an exception when called, then it must do so on any future invocations as well.

In OCAML we can "emulate" this behavior using variant types, references and mutation, and the fact that expression are regular values, as follows:

- A type `'a result` holds in a type variant either the unevaluated expression, the resulting value or the raised exception, whichever occured.
- A type `'a lazy` holds a reference to a value of type `'a result`.
- A function `delay` takes as input a "thunk" and stores it in the reference.
- A function `force` takes as input a `'a lazy` value, and if it is still unevaluated then it evaluates it and adjusts the reference accordingly.

Here's how the whole code might look like:

```
type 'a result = Delayed of unit -> 'a | Evaled of 'a | Raised of exn
type 'a lazy = 'a result ref

(* val delay : (unit -> 'a) -> 'a lazy *)
let delay th = ref (Delayed th)

(*val force : 'a lazy -> 'a *)
let force r = match !r with
  | Evaled v -> v
  | Raised ex -> raise ex
  | Delayed th -> try let v = th ()
                      in (r := Evaled v; v)
                  with ex -> (r := Raised ex; raise ex)
```

Every time we want to delay an evaluation, and express the fact, we wrap it using the `delay` function. When we want to get the value out, we use `force`.