Chapter 8

Sometimes it's good to be lazy

Typically, languages such as OCaml (SML, Lisp or Scheme) evaluate expressions by a *call-by-value* discipline. All variables in OCaml are bound *by value*, which means that variables are bound to *fully evaluated* expressions. For example consider let-expressions: In OCaml, we would write `let x = e1 in e2`. How would this expression be evaluated? We first evaluate `e1` to some value `v1` and then bind the name of the bound variable `x` to the value `v1` before we continue evaluating `e2` in an environment where we have established the binding between `x` and `v1`.

A similar evaluation strategy is applied when evaluating function applications. We first evaluate the argument before it is passed to the function. For example, in the expression `(fun x -> e) e1` we first evaluate the expression `e1` to some value `v1` before we pass the value `v1` to the parameter `x` of the function effectively binding the name `x` to the value `v1`

According to a *call-by-value* strategy, expressions are evaluated and their values bound to variables, no matter if this variable is ever needed to complete execution. For example:

```
1 let  x = horribleComp(345) in  5
```

In this simple example, we will evaluate `horribleComp(345)` to some value `v1` and establish the binding between the variable `x` and `v1`, although it is clearly not needed. For this reason, languages whose evaluation is call-by-value are also called *eager*.

Alternatively, we could adapt a *call-by-name* strategy. This means variables can be bound to *unevaluated expressions*, i.e. *computation*. In the previous example, we would *suspend* the computation of `horribleComp(345)`, until the value for `x` is required by some operation. For example:

```
1 let x = horribleComp(345) in x + x
```

In this example, it is necessary to evaluate the binding for `x` in order to perform the addition `x + x`. Languages that adopt the *call-by-name* discipline are also called

lazy languages, because they delay the actual evaluation until it is required. We also often say evaluation for `horribleComp(345)` is *suspended*, and only *forced* when required.

One important aspect of lazy evaluation is *memoization*. Let us reconsider the previous example. Since the variable `x` occurs twice in the expression `x + x`, it is natural to ask if the expression `horribleComp(345)` is now evaluated twice. If this would be the case, then being lazy would have backfired! In reality, lazy evaluation adopts a refinement of the *call-by-name* principle, called *call-by-need* principle. According to the *call-by-need* principle, variables are bound to unevaluated expressions, and are evaluated only as often as the value of that variable's binding is required to complete the computation. So in the previous example, the binding of `x` is needed twice in order to evaluate the expression `x + x`. According to the *call-by-need* principle, we only evaluate once `horribleComp(345)` to a value `v1`, and then we *memoize* this value and associate the binding for `x` to the value `v1`. In other words, the by-need principle says the binding of a variable is evaluated *at most once*, not at all, if it is never needed, and exactly once, if it is ever needed. Once a computation is evaluated, its value is saved for the future. This is called memoization.

The main benefit of lazy evaluation is that it supports *demand-driven* computation. We only compute a value, if it is really demanded somewhere else. This is particularly useful when we have to deal with *online data structures*, i.e. data-structures that are only created insofar as we examine them.

- Infinite data structures: For example, if we would want to represent *all* prime numbers this cannot be done eagerly. We cannot ever finish creating them all, but we can create as many as we need for a given run of a program!

- Interactive data structures such as sequences of inputs. Users inputs are not predetermined at the start of the execution but rather created on demand in response to the progress of computation up to a given point.

Surprisingly we can model lazy programming with the tools we have seen so far, namely functions and records.

How do we prevent the eager evaluation of an expression? - The key idea is to *suspend computation using functions*. Since we never evaluate inside the body of a function, we can suspend the computation of `3+7` by wrapping it in a function, i.e. `fun () -> 3 + 7`. In general, we will use functions of type `unit -> 'a` to describe the suspended computation of type `'a`. To highlight the fact that a function `unit -> 'a` describes a suspended computation, we tag such suspended computations with the constructor `Susp`.

```
1 type 'a susp = Susp of (unit -> 'a)
```


We can then delay computation labelling it with the constructor `Susp` using the function `delay` and force the evaluation of suspendend computation using the function `force`.

```
(* force:  'a susp → 'a *)
let force (Susp f) = f ()
```

We force a suspended computation by applying `f` which has type `unit -> 'a` to `unit`. For example, the following evaluates using call-by-name.

```
let  x = Susp(fun () -> horribleComp(345)) in  force x + force x
```

Only when we force `x` will be actually compute and evaluate `horribleComp(345)`.

8.1 Defining infinite objects via observations

Finite structures such as natural numbers or finite lists are modelled by (inductive) data types. For example, the data type `'a list` given below

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

encodes the following inductive definition of finite lists:

- `Nil` is a list of type `'a list`.
- If `h` is of type `'a` and `t` is a list of type `'a list`, then `Cons (h,t)` is a list of type `'a list`.

The inductive definition defines how we can construct finite lists from "smaller" lists. In mathematical terminology, the given inductive definition defines a least fix point. We can manipulate and analyze lists (or other inductively defined data) via pattern matching. Key to writing terminating programs is that our recursive call is made on the "smaller" list.

How can we model infinite structures such as streams or processes? - Instead of saying how to construct such data, we define *infinite objects via the observations* we can make about them. While finite objects are intensional, i.e. two objects are the same if they have the same structure, infinite objects are extensional, i.e. they are the same if we can observe them to have the same behavior. We already encountered infinite objects with extensional behavior, namely functions. Given the two following functions:

```
let f x = (x + 5) * 2
let g x = 2*x + 10
```


we can observe that for any input `n` given to `g` the result will be same as `f n`. We cannot however pattern match on the body of the function `f` and `g` analyze their structure/definition. Similarly, when we defined operations on Church numerals we were unable to directly pattern match on the given definition of a Church numeral. Instead, we observed the behavior by applying them.

To summarize, we cannot directly pattern mach on a function; we can only apply the function to an argument, effectively performing an experiment, and observe the result or behavior of the function.

Similarly, we want to define a stream of natural numbers via the observations we can make about it. Given a stream `1 2 3 4 ...`, we may ask for the head of the stream and obtain `1` and we may ask for the tail of the stream obtaining `2 3 4 ...`. Programs which manipulate streams do not terminate - in fact talking about termination of infinite structures does not make sense. However, there is a notion when a program about streams is "good", i.e. we can continue to make observations about streams. We call such programs *productive*. They may run forever, but at every step we can make an observation. This is unlike looping programs which run forever but are not productive.

To define infinite structures via the observations we exploit the idea of delaying the evaluation of an expression. For example, we can define elements of type `'a str` (read as a stream of elements of type `'a`) via two observations: `hd` and `tl`. Asking for the head of a stream using the observation `hd` returns an element of type `'a`. Asking for the tail of a stream using the observation `tl` returns a *suspended* stream. It is key that the stream is suspended here, since we want to process streams on demand. We will only unroll the stream further, when we ask for it, i.e. we force it.

```
1 type 'a str = {hd: 'a  ; tl : ('a str) susp}
```

In some sense a suspended stream is a box; in order to know what it looks like, we need to force it, i.e. open the box, to look inside.

8.2 Create a stream of 1's

Now we are in the position to create an infinite stream of one's for example, i.e. `1 1 1 ...`.

```
1 let rec ones = {hd = 1 ; tl = Susp (fun () -> ones)}
```

The head of a stream of one's is simply `1`. The tail of a stream of one's is defined recursively referring back to its definition. We can see here that it is key that the recursive call `ones` is inside the function `fun () -> ones` thereby preventing the evaluation of `ones` eagerly which would result in a looping program.

Maybe even more interestingly, we can now create a stream for the natural numbers as follows. We first define a stream of natural numbers starting from `n`. Hence the head of such a stream is simply `n`. The tail of such a stream is simply referring back to its definition incrementing `n`.

```
(* numsFrom : int -> int str  *)
let rec numsFrom n =
{hd = n ;
 tl = Susp (fun () -> numsFrom (n+1))}

let nats = numsFrom 0
```

8.3 Power and Integrate series

Additional Examples:

```
(* Sequent of 1 3 9 27 .. *)
let rec fseq n k =
  { hd = n ;
    tl = Susp (fun () ->  fseq (n*k) k) }

(* Sequent 1, 1/2, 1/4, 1/8, 1/16 ... *)
let rec geom_series x =
  { hd = (1.0 /. x) ;
    tl = Susp (fun () -> geom_series (x *. 2.0)) }
```

Next we show how to directly define the power series.

$$1,\ 2/3,\ 4/9,\ 8/27,\ \ldots = series(2^{x_i}/3^{x_i})$$

```
let rec pow n  x = if n = 0 then 1
  else x * pow (n-1) x

let rec power_series x =
{ hd = (float (pow x 2)) /. (float (pow x 3)) ;
  tl = Susp (fun () -> power_series (x+1)) }
```

We can now define how to integrate a series:

```
let integrate_series s =
  let rec aux s n =
    { hd = s.hd /. (float n) ;
      tl = Susp (fun () -> aux (force s.tl) (n + 1) )
    }
  in
  aux s 1
```


8.4 Writing recursive programs about infinite data

It is often useful to inspect a stream up to a certain number of elements. This can be done by the next function:

```
(* int -> 'a str -> 'a list *)
let rec take_str n s = match n with
  | 0 -> []
  | n -> s.hd :: take_str (n-1) (force s.tl)
```

Another example is dropping the first `n` elements of a stream.

```
(* stream_drop: int -> 'a str -> 'a str *)
let rec stream_drop n s = if n = 0 then  s
  else stream_drop (n-1) (force s.tl)
```

8.5 Adding two streams

Next, let us add two streams. This follows the same principles we have seen so far.

```
(* val addStreams : int str -> int str -> int str *)
let rec addStreams s1 s2 =
{hd = s1.hd + s2.hd ;
 tl = Susp (fun () -> addStreams (force s1.tl) (force s2.tl))
}
```

8.6 Fibonacci

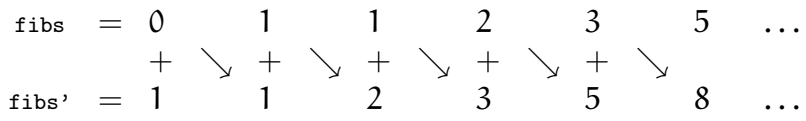
Let us move on to generating some more interesting infinite sequences of numbers such as the Fibonacci number series. The Fibonacci Sequence is the series of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

The next number is found by adding up the two numbers before it.

n	$\mathrm{Fib}(n)$	$+$	$\mathrm{Fib}(n+1)$	$\mathrm{Fib}(n+2)$
0	$0, 1, \ldots$		$1, \ldots$	$1, \ldots$
1	$1, 1, \ldots$		$1, \ldots$	$2, \ldots$
2	$1, 2, \ldots$		$2, \ldots$	$3, \ldots$

The stream in the right column, labelled $\mathrm{Fib}(n+1)$, is the tail of the stream described in the left column, labelled $\mathrm{Fib}(n)$. By adding up the two streams pointwise, we should be able to compute the next element in the stream!



The key observation is that we need to know the n-th and (n+1)-th element to compute the next element. In terms of streams it means to define the next element in a stream we need to know already not only the head of the stream we are defining, but also the head of the tail of the stream!

We will definie the fibonacci series hence mutually recursive as follows:

- The first element of the fibonacci series is `0`.
- The second element (i.e. the head of the tail or the head of `fibs'`) of the fibonacci series is `1`.
- The remaining series (i.e. the tail of the tail or the tail of `fibs'`) is obtained by adding the stream `fibs` and `fibs'`.

This can be implemented using records and suspended functions in OCaml directly.

```
let rec fibs =
{ hd = 0 ;
  tl = Susp (fun () -> fibs') }
and fibs' =
{hd = 1 ;
 tl = Susp (fun () -> addStreams fibs fibs')
}
```

8.7 Map function on streams

How can we write some of the functions we have seen for lists for streams? – Let's start with the map function.

```
(* smap: ('a -> 'b) -> 'a str -> 'b str *)
let rec smap f s =
{ hd = f (s.hd) ;
  tl = Susp (fun () -> smap f (force s.tl))
}
```

Note it is important that we `force` the tail of the stream when making the recursive call, since `s.tl` has type `('a str) susp`, i.e. it is a suspended stream. However, `smap` requires a stream, not a suspended stream.

8.8 Filter

```
(* val filter_str : ('a -> bool) -> 'a str -> 'a str
   val find_hd : ('a -> bool) -> 'a str -> 'a * 'a str susp
*)
let rec filter_str p s =
  let h,t = find_hd p s in
  {hd = h;
   tl = Susp (fun () -> filter_str p (force t))
  }
and find_hd p s =
if  p (s.hd) then (s.hd, s.tl)
  else find_hd p (force s.tl)
```

Note that `find_hd` is not productive. In principle, we could test for a property `p` which is never fullfiled by any element in the stream `s`. We would then continue to call eagerly `find_hd p (force s.tl)`. In fact, productive programs should always suspend their recursive computation. Since `find_hd` is not productive, `filter` is also not necessarily productive.

Using `filter`, we can now define easily a stream of even and odd numbers.

```
let evens = filter_str (fun x -> (x mod 2) = 0) (force (nats.tl))

let odds = filter_str (fun x -> (x mod 2) <> 0) nats
```

8.9 Sieve of Eratosthenes

A great application of lazy programming and streams is computing a stream of prime numbers given a stream of natural numbers. The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers. It does so by iteratively marking or filtering out the multiples of each prime, starting with the multiples of 2.

The idea can be best described by an example. We begin with a stream s of natural numbers starting from 2, i.e. 2 3 4 5 6 ....

1. The first prime number we encounter is the head of the stream s, i.e. 2 when we have a stream 2 3 4 5 6 ....

2. Given the remaining stream 3 4 5 6 ... (i.e. the tail of s), we first remove all numbers which are dividable by 2 obtaining a stream 3 5 .... We then start with 1. again.

To find all the prime numbers less than or equal to 30 (and beyond), proceed as follows. First generate a list of integers from 2 to 30 and beyond

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ...

The first number in the list is 2; this is the first prime number; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e. all the multiples of 2:

3 5 7 9 11 13 15 17 19 21 23 25 27 29 ...

Next prime number after 2 is 3; now cross out every number which can be divided by 3, i.e. all the multiples of 3:

5 7 11 13 17 19 23 25 29 ...

Next prime number is 5; cross out all the multiples of 5:

7 11 13 17 19 23 29 ...

Next prime number is 7; the next step would be to cross out all multiples of 7; but there are none. In fact, the numbers left not crossed out in the list at this point are all the prime numbers below 30.

Following the idea described, we now implement a function `sieve` which when given a stream `s` of natural numbers generates a stream of prime numbers. The head of the prime number stream is the head of the input stream. The tail of the prime number stream can be computed by filtering out all multiples of `s.hd` from the tail of the input stream.

```
let no_divider m n = not (n mod m = 0)

let rec sieve s =
{ hd = s.hd ;
  tl = Susp (fun () -> sieve (filter_str (no_divider s.hd) (force s.tl)))
}
```

8.10 Lazy finite lists

On demand evaluation is not only necessary when we deal with infinite structures, but it can also be useful when processing structures which can possibly be finite. We show here how to define (possibly) finite lists via observations to allow on demand evaluation. This is accomplished by nesting inductive and coinductive definitions. First, we define `'a lazy_list` coinductively via the observation `hd` and `tl`. The tail of a

possibly finite list may be finite, i.e. we have reached the end of the list and the list is empty, or we can continue to make observations, i.e. the list is not empty.

```
type 'a lazy_list = {hd: 'a  ; tl : ('a fin_list) susp}
and 'a fin_list = Empty | NonEmpty of 'a lazy_list
```

We can now re-visit the standard functions for infinite lists and rewrite them for lazy (finite) lists. We show below the implementation of `map` for lazy lists. We split the function into two mutual recursive parts: `map` is a recursive function; if the given list `xs` is `Empty`, then we return `Empty`; if we have not reached the end, we continue to apply lazily `f` to `xs` tagging the result with `NonEmpty`. `map'` applies lazily `f` to a list `s`; this is done by defining what observations we can make about the resulting list given a function `f` and an input list `s`.

```
let rec map f s =
{ hd = f s.hd ;
  tl = Susp (fun () -> map' f (force s.tl))
}

and map' f xs = match xs with
  | Empty -> Empty
  | NonEmpty xs -> NonEmpty (map f xs)
```

8.11 Summary

We demonstrated in this Chapter how to write on-demand programs using observations. This is useful for processing infinite data such as streams but also finite data which may be too large to be processed and generated eagerly. The idea of modelling infinite data via observations (i.e. destructors) is the dual to modelling finite data via constructors. While this view has been explored in category theory and algebra, it has only recently been been picked up and incorporated into actual programming practice up by A. Abel and B. Pientka together with their collaborators [1]. While pattern matching is used to analyze finite data, they introduced the idea of copattern matching to describe observations about infinite data. Based on their work OCaml was extended with direct support for infinite data and copattern matching (see opam package for copatterns; see `ocaml14.04.0+copatterns`).