Notes on Lazy Evaluation versus Eager Evaluation

Gopalan Nadathur

The expressions that need to be evaluated in a functional programming language can be uniformly depicted as ones of the form

$$(e_1\ e_2).$$

In expressions of this sort, both e_1 and e_2 represent expressions that may have to be evaluated at some time in the computation. Different evaluation methods arise from considering the question of which of these expressions to evaluate first and how much of them to evaluate at a point when they are chosen for evaluation.

OCaml uses what is known as call-by-value semantics or, alternatively, is based on *eager evaluation*. This means that evaluation proceeds as follows:

(i) The expression e_2 is evaluated first. Let us call the value that is produced v.

(ii) The expression e_1 is evaluated next. A common point with all evaluation orders in functional programming languages is that evaluation stops when either an object like `[]` or 1 or `1::2::[]` that cannot be further simplified is obtained or a function is exposed. In the present context e_1 must have a function type and so a function must be exposed. Let us call this function f.

(iii) The function f is applied to the object v. To be specific, suppose that f is of the form `fun x -> body`. Then, first the result of replacing `x` in `body` by v is computed and then the resulting expression is evaluated using the same evaluation rules as the ones being described. In the case that f is a built-in function, the result of applying f to v is system defined.

Before we move on, we should note that the evaluation of e_2 and e_1 can be done in reverse order and it will still be call-by-value or eager evaluation. The main requirement for this style of evaluation is that the argument must be fully evaluated *before the function is applied to it.*

An alternative to the *call-by-value* semantics that has received much attention is what is referred to as *call-by-name* semantics. The semantics entails the following sequence of steps in evaluation:

(i) As with call-by-value semantics, the expression e_1 is first evaluated. Once again, a function object must be exposed by the process. Let us call this function f.

(ii) The expression e_2 is evaluated *only to the extent needed to allow f to be applied to it.* Notice that some evaluation of e_2 may be necessary because the definition of f may be based, for instance, on the arguments satisfying certain patterns as in OCaml. Let us refer to this (only slightly evaluated) form of e_2 as e_2'.

(iii) The function f is applied to the object e_2'. As before, if f has the form `fun x -> body`, then this step consists of replacing `x` in `body` by e_2' and evaluating the resulting expression using the rules being described.

Thus in call-by-name semantics, the evaluation of arguments takes place in a demand driven fashion. Using call-by-name semantics naively could result in repeated computations. For example, suppose we have the definition

```
let double x = plus x x
```

and consider the evaluation of the expression

```
(double (4 * 5)).
```

In the naive scheme, two copies of the expression `(4 * 5)` will be created and evaluated separately. However, a smarter implementation would try to "share" one copy of this expression between the two occurrences of `x` that get substituted for by it and would thereby evaluate it only once. When call-by-name semantics is implemented with such sharing, it is called *lazy* or *call-by-need* evaluation.

In comparing eager and lazy evaluation, one question to consider is which of them is guaranteed to produce the correct result. In answering this question, it is necessary to note that there are two different kinds of interpretations that are accorded to functions, generally referred to as the *strict* and *non-strict* interpretations. Under the first, a function has a defined result only when its arguments are defined. This is in consonance with the view of functions generally prevalent in mathematics—functions are sets of ordered pairs, and there is no sense in asking for the second component of an ordered pair whose first component is undefined! The non-strict view of functions, on the other hand, allows the result of applying them to arguments to be defined even if the arguments themselves are not. This view is in agreement with the view of functions as rules or methods for computing as opposed to sets of ordered pairs. Thus, the rule could produce a well-defined value from the ill-defined input if it does not have any use for the input.

An example would be useful to fix these notions. Consider the following definitions in an ML-like language:

```
let g x = 5
let h x y = h y x
```

Now consider the expression `(g (h 3 3))`. What should its value be? Under strict semantics for functions, the argument—`(h 3 3)`—is undefined and so the expression itself is undefined. Under non-strict semantics, the value is `5`—the function `g` when looked at as a rule has no real use for its argument, so why does it matter if it is a value such as `3` or an undefined value?

Now, which of eager and lazy evaluation is correct? If the strict interpretation of functions is used, then clearly lazy evaluation produces incorrect results in certain instances. On the other hand if the non-strict interpretation is adopted, eager evaluation is only an approximation of the correct behavior since it fails to produce a result in certain cases where a value exists. To see this, try to compute `(g (h 3 3))` using eager evaluation.

At a pragmatic level, we are willing to ignore these theoretical differences (and discussions) and would like to understand if there are significant behavioral differences between lazy and eager evaluation. Some of the differences worth mentioning are the following:

(1) Often it is necessary to interleave the steps in the obvious approach to solve a problem to obtain the most efficient computation. Lazy evaluation has the characteristic of supporting the needed interleaving automatically. Under eager evaluation, it is necessary for the user to build in the interleaving into his/her program.

(2) It is sometimes useful to be able to represent infinite objects in programs. Lazy evaluation permits these to be represented as a finite evaluated part and an infinite as yet unevaluated part. Under eager evaluation, there is a danger that evaluation may try to expose the entire object, and some coding tricks have to be used to avoid this.

(3) On the positive side for OCaml, efficient implementation of eager evaluation is a somewhat easier matter to handle as also is the question of how to introduce "non-functional" features like assignment into the framework. Part of the reason for this is that the compilation of sharing and incremental evaluation of expressions is a more difficult problem as also is the problem of predicting the order in which side-effecting operations are executed in a situation where the evaluation of subexpressions may take place in an arbitrary order. Contrast the efficiency aspect with the first point above: one has to think a bit more to encode a solution in an efficient way in a language based on eager evaluation, but then the actual execution of this solution may be faster.

It is beyond the scope of this handout to expand on the last of the points above. However, we can consider the first two points in some more detail. We do this by considering some example programs.

Example 1

As the first example, suppose that we are interested in comparing the frontiers of two binary trees to see if they are identical. In case you haven't seen this notion before, the frontier of a tree is the information stored in the leaves.

So let us suppose that our tree data structure is given by the following type declaration in an ML-like language:

```
type 'a tree =
  | Empty
  | Leaf of 'a
  | Node of ('a tree) * ('a tree)
```

Thus, two trees that have the same frontiers are

```
Node (Node (Leaf 3, Node (Leaf 4, Empty)), Node (Leaf 5, Leaf 6))
```

and

```
Node (Node (Leaf 3, Leaf 4), Node( Node (Leaf 5, Empty), Leaf 6)).
```

Now, how do we compare two given trees to see if their frontiers are equal? Well, one way is the following: we simply generate two lists containing the items that constitute the leaves of these trees and then compare these lists. Using this idea, we can write the following program in an OCaml-like language; the equality relation on lists is available as a primitive in OCaml, but I have encoded it here because this is relevant to understanding the nature of the computation under lazy evaluation.

```
let rec equal l1 l2 =
 match (l1,l2) with
 | ([],[]) -> true
 | (h1::t1,h2::t2) -> h1 = h2 && equal t1 t2
 | (_,_) -> false

let rec frontier t =
   match t with
```

```
   | Empty -> []
   | Leaf x -> [x]
   | Node (t1,t2) -> (frontier t1) @ (frontier t2)

  fun samefrontier x y = equal (frontier x) (frontier y)
```

The problem with this "obvious" solution is that it can be rather inefficient under the OCaml evaluation rule when trees with distinct frontiers are being compared. To see this, consider the evaluation of the expression

```
   samefrontier (Node (Leaf(3*2), E)) (Node (Leaf (3*3), F))
```

assuming that `E` and `F` represent some potentially large trees. We can conclude that these trees do not have the same frontier simply by examining their first leaves. However, in the OCaml program, the frontiers of the two trees will be (eagerly) computed using `frontier` before the comparison process even gets started. Thus, a lot of unnecessary effort will be expended.

Now consider what happens under lazy evaluation with the same expression. The attempt to evaluate this expression results in the need to evaluate the expression

```
  equal (frontier (Node (Leaf(3*2), E))) (frontier (Node (Leaf(3*3), F))).
```

Now the arguments of these expressions will be evaluated, but *only to the extent necessary for determining which of the cases in the definition of* `equal` *is to be used*. Working through the definition of `frontier` and `@` (i.e. *append*), it is seen that this results in the expression

```
  (equal (6::([] @ (frontier E))) (9::([] @ (frontier F))))
```

being produced. Using the second clause in the definition of equal, this expression now evaluates to

```
  (6 = 9) && (equal ([] @ (frontier E)) ([] @ (frontier F)))
```

Now, using the evaluation rule for `&&`, this expression right away evaluates to `false`, *without further computing the frontiers of the two trees.*

Thus, we see that at least in the case of this problem, lazy evaluation engenders the kind of computation we would desire based on the encoding of the obvious solution in OCaml-like syntax. But can we get the same kind of behavior out of a perhaps more complicated program in an eager language? The answer is yes. The way we get this behavior is by explicitly interleaving the computation of the frontier and the equality checking into our program and by incorporating laziness in our computation via the selective or conditional evaluation that goes with the `&&` operator. A program for checking the equality of frontiers for two trees that utilizes these ideas is the following:

```
let rec compforests f1 f2 =
   match (f1,f2) with
   | ([],[]) -> true
   | ((Empty::x),y) -> compforests x y
   | (x,(Empty::y)) -> compforests x y
   | (((Leaf a)::x),((Leaf b)::y)) ->
      (a=b) && (compforests x y)
```

```
  | (((Node (l,r))::x),y) -> compforests (l::r::x) y
  | (x,((Node (l,r))::y)) -> compforests x (l::r::y)
  | (_,_) -> false

let samefrontier t1 t2 = compforests [t1] [t2]
```

To make sure you understand the comments about this way of structuring the computation, work through the calculations involved in evaluating the expression

```
samefrontier (Node (Leaf (3*2), E)) (Node (Leaf (3*3),F)).
```

Focus especially on the last clause for the `compareforests` and on how the special evaluation rule for `&&` permits certain computations to be delayed till it is discovered that they, in fact, need not be performed.

Example 2

The second example is that of representing infinite objects. Consider, for example, the following definition:

```
let rec nat n = n :: (nat (n+1))
```

The intention here is to define an infinite sequence of natural numbers starting with the number n that is provided as a parameter. Now, of course, we do not want the entire sequence to be produced for us at any one time. Rather, we think of producing this incrementally and using some finite portion of the sequence in computations. Thus, we might think of generating the first `m` numbers in this sequence using the following function that picks out a specified number of initial elements for a given list:

```
let rec firstm n l =
 match (n,l) with
 | ((0,_) | (_,[])) -> []
 | (m,(a::l)) -> a::(firstm (m-1) l)
```

To be precise, we might think of doing this through the evaluation of expressions of the form

```
(firstm 5 (nat 1))
```

This computation is meaningful in the context of a lazy language. The basic observation is that the infinite sequence of natural numbers that results from the evaluation of `(nat 1)` in this expression is always represented as some initial segment that is used in the computation together with a suspended computation that will produce more numbers when there is a demand for them. In the particular computation, the first five numbers of the sequence will be actually produced, and the remaining part will be left in "suspended" in the expression `(nat 6)`.

Notice, however, that this computation is *not* meaningful in OCaml. The reason for this is that in trying to evaluate the expression `(firstm 5 (nat 1))`, one of the first things that will have to be done is to evaluate `(nat 1)`. Unfortunately an eager evaluation of this last expression leads to an infinite computation for obvious reasons.

Is there a way to encode infinite objects in OCaml then? The answer is yes. The trick is to make use of the fact (mentioned at the outset) that the evaluation of an expression stops as

soon as a function structure is exposed for an expression. Thus, whenever we want to suspend the computation of an infinite object, we can "hide" this object under a "dummy" function. The dummy function will be one that takes the `()` object as an argument.

Using this approach, the infinite stream of natural numbers and the function for generating the first `m` of these can be defined in ML as follows:

```
type 'a stream =
   | Empty
   | Stream of 'a * (unit -> ('a stream))

let rec nat n () = Stream(n, nat (n+1))

let rec firstm n l =
  match (n,l) with
 | ((0,_) | (_,Empty)) -> []
 | (m,Stream (a,rest)) -> a :: firstm (m-1) (rest())
```

Here the `type` definition identifies a type constructor for encoding a potentially infinite sequence (known as a stream) of objects of some given type. In the simple case, this is an `Empty` stream. In the other case, the stream has a first element and a remaining part that is represented as a dummy function; in more detail, these two components are represented as arguments of a value constructor called `Stream`, much like the head and tail of a list form the arguments of the constructor `::`. The function `nat` that is to generate a stream starting at a certain value now becomes a function of a dummy argument in addition to that of a starting value. When it is given both arguments, it produces the first item in the stream and returns the remainder of the stream as a suspended function that is waiting, so to speak, for a dummy argument that signals the desire for the next item in the stream. Given this basic structure for representing an infinite stream, it should be clear how the stream is to be generated incrementally. The definition of `firstm` in ML provides an illustration of this process. As a concrete example of the use of the natural number stream together with the function `firstm` that is a "consumer" for the stream, consider the following:

```
# firstm 5 (nat 1 ());;
- : int list = [1; 2; 3; 4; 5]
#
```

Trying to define and do things with infinite objects can be challenging and also a lot of fun. A slightly more elaborate example of the use of the notions described above is provided by the following code that is intended to generate the primes:

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

let mkStream f = Stream f
let nextStream (Stream f) = f ()

let rec sift a s () =
     let (x,rst) = nextStream s in
```

```
      if (x mod a = 0) then (sift a rst ())
      else (x, Stream (sift a rst))

let rec siftStream a s = mkStream (sift a s)

let rec sieve s () =
     let (x,rst) = nextStream s in
       (x, Stream (sieve (siftStream x rst)))

let sieveStream s = mkStream (sieve s)

let primesStream = sieveStream (fromNStream 2)

let rec take n s =
    match n with
    | 0 -> []
    | n ->
      let (x,rst) = nextStream s in
        (x :: take (n-1) rst)
```

Here, `primeStream` represents the infinite sequence of primes in a finite way; note also that we have changed our view of streams in that they represent only a never-ending sequence. We can extract some finite initial segment of these primes using the `take` function as the following interaction demonstrates:

```
# take 7 primesStream;;
- : int list = [2; 3; 5; 7; 11; 13; 17]
#
```

Try to understand the various definitions in this collection some of the ideas underlying the algorithm for constructing the stream of primes, which is based on the method of Eratosthenes, are also explained in the lecture slides.