CS 321 Programming Languages

Eager vs. Lazy Evaluation

Baris Aktemur

Özyeğin University

Last update made on Wednesday 21st December, 2016 at 11:10.

OCaml is an *eager* (aka call-by-value, strict) language. So are many of the main stream languages such as Java, C, etc. Being eager, informally, means that the function arguments are evaluated to values before evaluating the function body.
Consider this artificial example:

```
# let foo n = 42;;
val foo : 'a -> int = <fun>

# foo (fibonacci(40));;
- : int = 42
```

In this example, 40^{th} fibonacci number is calculated and passed to the function `foo` although `foo` does not use it. In case `fibonacci` is poorly implemented, we may have to spend a significant time for no use.

There are, however, *lazy* (aka call-by-name) features in OCaml. Being lazy, informally, means that an expression is not evaluated unless necessary.
Consider this artificial example:

```
# if true then 42 else 23/0;;
- : int = 42
```

Here, no error occurs although there exists the expression "23/0" which would normally raise a division by zero error. "If" expression in OCaml is lazy for its *then* and *else* branches; a branch is evaluated only if necessary as indicated by the boolean value obtained from the condition.

To make the distinction between eager and lazy evaluation better, now consider this case: To avoid having to type extra characters, I define my own "if" using the following function:

```
# let myif cond thenBranch elseBranch =
    if cond then thenBranch else elseBranch;;
val myif : bool -> 'a -> 'a -> 'a = <fun>

# myif (5>4) 42 23;;
- : int = 42

# myif (5>4) 42 (23/0);;
Exception: Division_by_zero.
```

Because OCaml is an eager language, the function argument "23/0" is evaluated before we execute the function body, even though only the value obtained from the `thenBranch` expression is used.

Simulating lazy evaluation in an eager language

Preventing the evaluation of an expression in an eager language can be done by putting that expression "under a lambda".

```
# fun () -> 23/0;;
- : unit -> int = <fun>

# let g = fun () -> 23/0;;
val g : unit -> int = <fun>

# g();;
Exception: Division_by_zero.
```

Here, we're relying on the fact that a function definition immediately evaluates to a closure, without evaluating the body. The body is evaluated when the function is applied.

Simulating lazy evaluation in an eager language

Now we can re-define "myif" as follows:

```
# let myif cond thenBranch elseBranch =
    if cond then thenBranch() else elseBranch();;
val myif : bool -> (unit -> 'a) -> (unit -> 'a) -> 'a = <fun>

# myif (5>4) (fun() -> 42) (fun() -> 23/0);;
- : int = 42 (* Yuppie, no error! *)
```

The function in the form `fun () -> ...`, which is used for the purposes of delaying a computation, is called a **thunk**. Evaluating the function to get the value of the delayed expression is called **forcing**. Enclosing a computation inside a thunk so that it can be delayed is called **thunking**. Lazy languages such as Haskell automatically do thunking and forcing.

Efficiency of lazy vs. eager

Our first example can be re-written as follows:

```
# let foo n = 42;;
val foo : 'a -> int = <fun>

# foo (fun() -> fibonacci(40));;
- : int = 42
```

This completely avoids computing fibonacci(40) because it is not needed. Therefore, lazy version is more efficient compared to the eager one.

Efficiency of lazy vs. eager

Lazy evaluation, when simulated the way we did, is not always more efficient compared to the eager model. It can avoid unnecessary computations, but it can also repeat computations although not needed. Consider the following artificial example.

```
# let double n = (n, n);;
val double : 'a -> 'a * 'a = <fun>

# double (fibonacci(40));;
- : int * int = (165580141, 165580141)

# let doubleLazy n = (n(), n());;
val doubleLazy : (unit -> 'a) -> 'a * 'a = <fun>

# doubleLazy (fun() -> fibonacci(40));;
- : int * int = (165580141, 165580141)
```

Do you see the difference?

Streams

Streams

A stream is a possibly infinite sequence of data. E.g. key strokes, tweets ffrom Twitter, video/audio stream, location changes on a cell-phone, etc. For instance, the infinite list of natural numbers would be
[0; 1; 2; 3; 4; ...].

A (failing) attempt to define the infinite list of natural numbers could be the following:

```
# let rec naturalsFrom n = n :: naturalsFrom (n+1);;
val naturalsFrom : int -> int list = <fun>

# let naturals = naturalsFrom 0;;
Stack overflow during evaluation (looping recursion?).
```

where `naturalsFrom` is a function that is supposed to return the list of natural numbers starting from a given number `n`. Hence, `naturalsFrom 0` shall return the naturals numbers, however, it goes into infinite recursion.

Streams

Recall that you could manually define the regular OCaml list datatype using the following definition:

```
type 'a mylist =
  | Empty
  | Cons of 'a * 'a mylist
```

Then, the list [1;2;3], which is just syntactic sugar for 1::2::3::[], could be represented as

```
Cons(1, Cons(2, Cons(3, Empty)))
```


Streams

We can define streams using a similar approach. But this time, we use a "thunk" to delay the evaluation of the tail of the stream to avoid the infinite trap. (We also ignore the base case constructor because we are interested in infinite data.)

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

Note that the second element in the Cons constructor above is a closure. That allows us suspend the computation of the tail of a stream. Here is how to define the stream of natural numbers.

```
# let rec naturalsFrom n = Cons(n, fun() -> naturalsFrom(n+1));;
val naturalsFrom : int -> int stream = <fun>

# let naturals = naturalsFrom 0;;
val naturals : int stream = Cons (0,<fun>)
```

Streams

Let us continue by defining useful functions on streams, inspired from the corresponding functions for lists.

```
# let head st =
    match st with
    | Cons(v, _) -> v;;
val head : 'a stream -> 'a = <fun>

# let tail st =
    match st with
    | Cons(_, f) -> f();;
val tail : 'a stream -> 'a stream = <fun>
```

Note how the definition of `tail` above forces the computation of the tail of the stream (i.e. `f()`).

Streams

To make debugging streams easy, a `take` function that takes the first `n` elements of a stream and returns a list would be very useful.

```
# let rec take n st =
    if n = 0 then []
    else (head st)::(take (n-1) (tail st))
val take : int -> 'a stream -> 'a list = <fun>

# take 5 naturals;;
- : int list = [0; 1; 2; 3; 4]
# take 6 (tail (tail naturals));;
- : int list = [2; 3; 4; 5; 6; 7]
```

Streams

Let's define the map function for streams.

```
# let rec map f st =
    Cons(f(head st), fun () -> map f (tail st));;
val map : ('a -> 'b) -> 'a stream -> 'b stream = <fun>

# take 8 (map ((+) 2) naturals);;
- : int list = [2; 3; 4; 5; 6; 7; 8; 9]
# take 6 (map (fun n -> n * n) naturals);;
- : int list = [0; 1; 4; 9; 16; 25]
```


Streams

Here is the filter function.

```
# let rec filter p st =
    if p(head st)
    then Cons(head st, fun () -> filter p (tail st))
    else filter p (tail st);;
val filter : ('a -> bool) -> 'a stream -> 'a stream = <fun>

# let threes = filter (fun x -> x mod 3 = 0) naturals;;
val threes : int stream = Cons (0,<fun>)

# take 5 threes;;
- : int list = [0; 3; 6; 9; 12]
```

Streams

See the sample code for other examples.