Lazy Evaluation and Streams

Ziyan Maraikar

May 5, 2014

Table of Contents

Eager evaluation

- An expression is evaluated at the point it is bound to a variable.
- In the example below, x is evaluated even though it is not used in the subsequent expression.

```
let  x = 1/0 in 1+2 ;;
```

- Common strategy for evaluation in Ocaml.

Lazy evaluation

- An expression is evaluated only at the point its value is needed.
- In the example below, the `then` expression is not evaluated because it is not used in the subsequent expression.

```
let g = if false then 1/0 else 1 ;;
```

- Used only when evaluating conditions and pattern matching in Ocaml.

Call by value vs. call by name

- *Call by value* – a function's arguments are evaluated *before* it is called.
- *Call by name* – arguments are substituted into the function *without* evaluating them. They are evaluated only at the point of use.
- Which of these does Ocaml use?

```
let  f x y = y + 10 ;;
f (1/0) 10 ;;
```

Lazy vs. eager evaluation

- Eager evaluation with call by value is the default in a majority of languages.
- Haskell is the best know language where lazy evaluation is the default. It uses an optimised variation of call by name called *call by need*.
- Lazy evaluation permits interesting features such as computing with infinite data. It can be added onto an eager language.

Table of Contents

Stream processing

Ability to compute with (potentially) infinite sequences of data has a number of uses.

- Defining mathematical sequences: Fibonacci, primes, ...
- Signal processing (audio, video)
- Reading data from a network socket

The stream type

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

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

The second argument to stream `Cons` is a function. This is called a *thunk*.

Stream examples

```
let rec ones =
  Cons (1, fun () -> ones)

let rec from n  =
  Cons (n, fun () -> from (n + 1))
let naturals = from 0

let fibs =
  let rec fibgen a b =
    Cons(a, fun () -> fibgen b (a + b))
  in fibgen 1 1
```

Working with streams

```
(* get the first value *)
let head s  = match s with
  | Cons (x, _) -> x

(* get the tail by evaluating the thunk *)
let tail s = match s with
  | Cons (_, g) -> g ()
```

Note how `tail` lets us *deconstruct* streams. What is its type?

Exercise

```
(* n-th element of a stream *)
let rec nth s n  =
  if n = 0 then head s else nth (tail s) (n - 1)

(* a list of the first n elements of a stream *)
let rec take s n  =
  if n <= 0 then [] else (head s) :: take (tail s) (n
    - 1)
```

MAP ON STREAMS

What type should `map` on stream return?

```
let rec map f s =
  Cons (f (head s), fun () -> map f (tail s))
```

Since we return a stream map on its tail is done lazily.

Exercise: write a `filter` function for streams.

Example: primes

Idea: Take the stream of natural numbers and repeatedly filter out multiples of each prime number encountered from the stream.

```
(* delete multiples of p from a stream *)
let sift p s=
  filter (fun n -> n mod p <> 0)

(* sieve of Eratosthenes *)
let rec sieve s  =
  | Cons (p, g) ->
      Cons (p, fun () -> sieve (sift p (g ())))

let primes = sieve (from 2)
```

Table of Contents

Key ideas in functional programming

Immutability equational reasoning via substitution.

First-class functions higher order functions and data-parallel processing via map-reduce.

Algebraic data types model data structures without worrying about *null*.

Modular programming separating interface from implementation via type abstraction.

Practical impact of FPLs

Functional programming has a long history of contributing ideas to mainstream languages:

- Garbage collection: popularised by LISP.
- Map-reduce: Hadoop, Java 8, C#.
- Stream processing: DSPs, GPGPU computing, realtime distributed computing.