Functional programming: tricks and patterns

OCaml PRO – Grégoire Henry

March 9-13 2015

Outline

Warm-up: vocabulary

Accumulator & continuations

List Write a function `filter p l` of type `('a -> bool) -> 'a list -> 'a list` that returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list should be preserved.

Tree Given the following definition of a tree where nodes are labeled with integer, write a function that sum all the label of a given tree.

```
type tree =
| Empty
| Node of tree * int * tree
```

```
let rec filter f xs =
  match xs with
  | [] -> []
  | x :: xs ->
      if f x then x :: filter f xs else filter f xs

let filter f xs =
  List.fold_right
    (fun x acc -> if f x then x :: acc else acc)
     xs []

let filter f xs =
  let may_cons x acc = if f x then x :: acc else acc in
  List.fold_right may_cons xs []
```

```
let filter f xs =
  let rec loop acc xs =
    match xs with
    | [] -> List.rev acc
    | x :: xs -> loop (if f x then x :: acc else acc) xs
  in
  loop [] xs

let filter f xs =
  List.rev @@
  List.fold_left
    (fun acc x -> if f x then x :: acc else acc)
    [] xs
```

```
let rec sum t =
  match t with
  | Empty -> 0
  | Node (t1, i, t2) -> i + sum t1 + sum t2

(* Using the generic "fold" combinator. *)
let rec fold_tree : (int -> 'a -> 'a) -> 'a -> tree -> 'a =
  fun f acc t ->
  match t with
  | Empty -> acc
  | Node (t1, i, t2) ->
      let acc = fold_tree f acc t2 in
      let acc = fold_tree f acc t1 in
      f i acc

let sum t = fold_tree (+) 0 t
let product t = fold_tree ( * ) 0 t
```

Accumulator & continuations

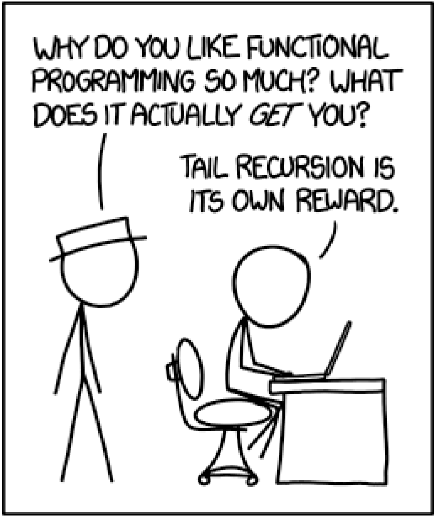

```
let sum t = (* tail rec with concrete continuation *)
  let rec loop cont acc t =
    match t with
    | Empty -> begin
        match cont with
        | [] -> acc
        | t :: cont -> loop cont acc t
      end
    | Node (t1, i, t2) -> loop (t2 :: cont) (i+acc) t1
  in
  loop [] 0 t

let sum t = (* tail rec with functional continuation *)
  let rec loop cont acc t =
    match t with
    | Empty -> cont acc
    | Node (t1, i, t2) ->
        loop (fun acc -> loop cont acc t2) (i+acc) t1
  in
  loop (fun acc -> acc) 0 t
```

Accumulator a common pattern to:

- allow tail recursion and avoid stack overflow;
- define generic iterator like `List.fold_left`.

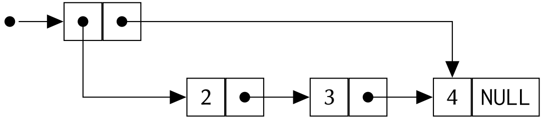
Continuation a functional parameter of a function, that will be applied to the returned value (instead of simply returning the value).


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

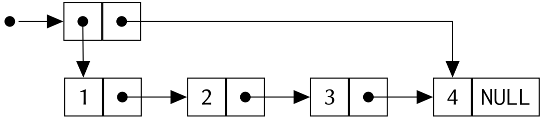
Persistent data structure

Imperative list (for instance in C)

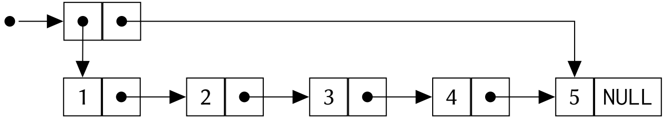
The memory representation of a 3-elements list:



Memory representation after inserting 1 in first position:



Memory representation after inserting 5 at the end:



Mutation is like a master-chef knive:

- a very effective tool, but also
- a very dangerous one.

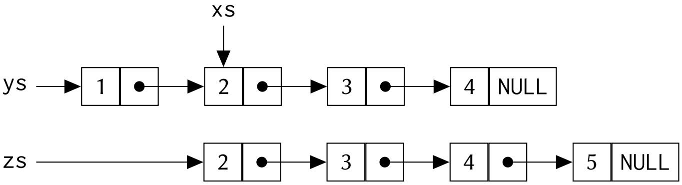
Today, its usage is strictly forbidden!

Do not mutate: copy and share!

- Let's freely[a] copy and share data! Without mutation, this is safe.
- Free your mind of deallocation, the garbage collection is efficient.

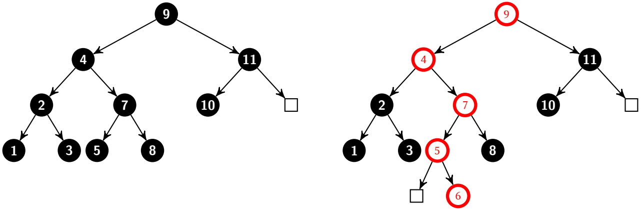
```
let xs = [2;3;4]
let ys = 1 :: xs
let zs = xs @ [5]
```

The resulting memory graph:

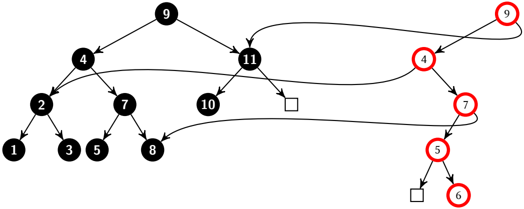


[a] while respecting the intellectual property.

Insertion in a binary search tree:



Sharing in the memory graph:



Do not mutate: copy and share!

For example, operations on lists:

```
val cons: 'a -> 'a list -> 'a list
val append: 'a list -> 'a list -> 'a list
val map: ('a -> 'b) -> 'a list -> 'b list
val fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example, operations on a binary search trees:

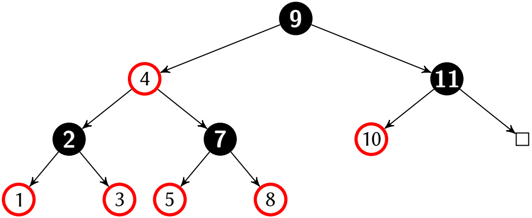
```
type 'a bst =
| Empty
| Node of 'a bst * 'a * 'a bst
val mem: 'a -> 'a bst -> bool
val insert: 'a -> 'a bst -> 'a bst
val union: 'a bst -> 'a bst -> 'a bst
val map: ('a -> 'b) -> 'a bst -> 'b bst
val fold: ('a -> 'b -> 'a) -> 'b bst -> 'a -> 'a
```

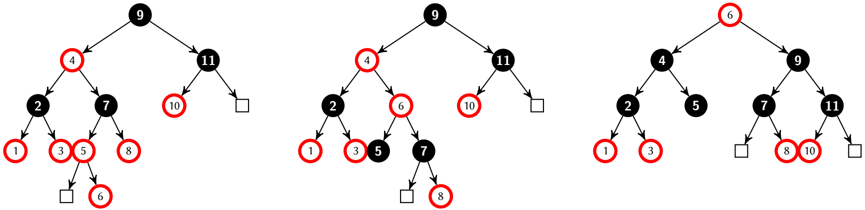
Red-black tree are:

- well-balanced binary search tree, where
- nodes are colored either in red or black.

Invariant 1 No red node has a red child.

Invariant 2 Every path from the root from the root to an empty node contains the same number of black nodes.

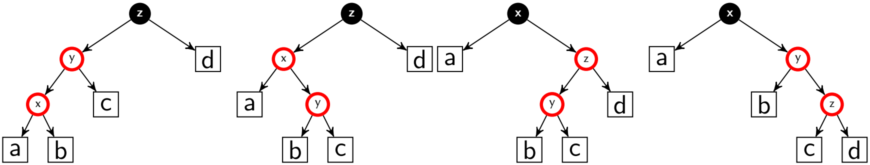




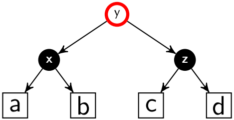
Insertion:

- descent into the tree and insert the new node as a red leaf, respecting the binary search tree property.
- on the path back to the tree root, rebalance all the sequence of two red nodes;
- tag the root in black.

Unbalanced patterns, where a, b, c, and d are valid red-black trees:



Re-balanced tree:



```
type color = R | B
type 'a bst =
  | E
  | N of color * 'a bst * 'a * 'a bst

let mem x t =
  match t with
  | E -> false
  | N (_, l, y, r) ->
     if x < y then mem x l
     else if y < x then mem x r
     else true
```

```
let balance = function
  | B, N(R, N(R, a, x, b), y, c), z, d
  | B, N(R, a, x, N(R, b, y, c)), z, d
  | B, a, x, N(R, N(R, b, y, c), z, d)
  | B, a, x, N(R, b, y, N(R, c, z, d)) ->
      N(R, N (B, a, x, b), y, N (B, c, z, d))
  | c, t1, x, t2 -> N (c, t1, x, t2)

let insert x t =
  let rec ins t =
    match t with
    | E -> N (R, E, x, E)
    | N (c, t1, y, t2) ->
        if x < y then balance (c, ins t1, y, t2)
        else if x > y then balance (c, t1, y, ins t2)
        else t in
  let N (_, t1, x, t2) = ins t in
  N (B, t1, x, t2)
```

"Functional programming combines
the flexibility and power of abstract mathematics
with the intuitive clarity of abstract mathematics."

R. Munroe

Functional-first programming

OCaml is a functional *and* an imperative language. There is some advantage in mixing the two paradigms.

Common patterns:

- keep the *core* of an application to a pure functional settings, or at least, use persistent data structure;
- keep functional interface while using mutation in the implementation (might be tricky to get right!);

Question: how to implement a efficient and persistent FIFO, i.e. without mutation ?

Laziness

Evaluation order

OCaml Strict evaluation: arguments are evaluated *before* a function call.
Haskell Laziness: arguments are evaluated when *first-used* by the function.

```
let rec fact x = if x <= 1 then 1 then x * fact (x-1)
let display x = if Random.bool () then print_int (x + x)
let () = display (fact 1_000_000)
```

With strict evaluation order, the expression `(fact 1_000_000)` is evaluated at each execution of the program[a]. On a lazy settings (or *call-by-need*), it is not evaluated when `Random.bool ()` returns `false` (and, in this case, the program execution time is almost 0s).

[a] This will probably systematically result in a stack overflow...

Explicit laziness in OCaml

OCaml allows explicit laziness.

- A predefined type: `'a lazy_t`.
- A syntactic construction: `lazy expr`, where `lazy` is a keyword
- A function for explicit evaluation: `(Lazy.force: 'a lazy_t -> 'a)`.

```
let rec fact x = if x <= 1 then 1 else x * fact (x-1)
let display : int lazy_t -> unit = fun x ->
  if Random.bool () then
    print_int (Lazy.force x + Lazy.force x)
let () = display (lazy (fact 1_000_000))
```

First call to `Lazy.force x` will compute the factorial and *memoize* the result. Second call will return immediatly the previously computed result.

```
type 'a stream = 'a stream_cell lazy_t
and 'a stream_cell =
  | Nil
  | Cons of 'a * 'a stream
```

```
let rec concat : 'a stream -> 'a stream -> 'a stream =
  fun xs ys ->
  lazy begin
    match Lazy.force xs with
    | Nil -> Lazy.force ys
    | Cons (x, xs) -> Cons (x, concat xs ys)
  end

let rec take : int -> 'a stream -> 'a stream = fun n xs ->
  lazy begin
    if n <= 0 then
      Nil
    else
      match Lazy.force xs with
      | Nil -> Nil
      | Cons (x, xs) -> Cons (x, take (n-1) xs)
  end
```

```
let rec to_list xs =
  match Lazy.force xs with
  | Nil -> []
  | Cons (x, xs) -> x :: to_list xs

let reverse xs =
  let rec rev acc xs =
    match Lazy.force xs with
    | Nil -> acc
    | Cons (x, xs) -> rev (lazy (Cons (x, acc))) xs in
  rev (lazy Nil) xs
```

```
let rec map : ('a -> 'b) -> 'a stream -> 'b stream =
  fun f xs ->
    lazy begin
      match Lazy.force xs with
      | Nil -> Nil
      | Cons (x, xs) -> Cons (f x, map f xs)
    end

(* The infinite stream of all natural *)
let rec naturals = lazy (Cons (0, map succ ints))
```

A lazy list: a stream

```
let tail xs =
  match Lazy.force xs with
  | Nil -> invalid_arg "empty"
  | Cons (_, xs) -> xs

let rec map2 f xs ys =
    lazy begin
      match xs, ys with
      | lazy Nil, _ | _, lazy Nil -> Nil
      | lazy (Cons (x, xs)), lazy (Cons (y, ys)) ->
          Cons (f x y, map2 f xs ys)
    end

(* The infinite stream of the fibonnaci sequence *)
let rec fibs =
  lazy (Cons (1, lazy (Cons (1, map2 (+) fib (tail fib)))))
```

A safe knive?

Internally, the laziness is implemented with side-effects:

- A creation a lazy value, of type `'a lazy_t`, is a *frozen* computation: it is a closure—representing a function of type `unit -> 'a`.
- After the first evaluation, all references to the closure are replaced with the computed value.

Warning: the hidden closure may induce (hard to track) memory leaks.

Persistent FIFO

Signature for persistent FIFO

```
type 'a fifo = ...
val empty : 'a fifo
val push : 'a -> 'a fifo -> 'a fifo
val pop : 'a fifo -> 'a * 'a fifo
```

```ocaml
exception Empty

type 'a fifo = {
  front: 'a list;
  rear: 'a list;
}

let empty = {front = []; rear = [] }
let push x fifo = { fifo with rear = x :: fifo.rear }

let may_reverse fifo =
  match fifo with
  | { front = [] } ->
      { front = List.rev fifo.rear; rear = [] }
  | _ -> fifo
let pop fifo =
  match may_reverse fifo with
  | { front = [] } -> raise Empty
  | { front = x :: front; rear } -> x, { front; rear }
```

A efficient implementation 2 / 2

- The function push is in constant time.
- In best case, the function pop is in constant time.
- In the worst case, the function pop is in $O(n)$ where n is the length of the rear list.
- In average, the function pop seems to be in constant time: the cost of reversing a rear list of length n is shared with n call to the push that precedes the call to the function pop.
- At least, until we do not account for persistency:

```
let fifo = empty
let fifo = push 1 fifo
let fifo = push 2 fifo
let fifo = push 3 fifo
let x, fifo_1 = pop fifo
let x, fifo_2 = pop fifo
let x, fifo_3 = pop fifo
```

```
type 'a fifo = {
  front_length: int;
  front: 'a list;
  lazy_front: 'a list lazy_t;
  rear_length: int;
  rear: 'a list;
}

let empty = {
  front_length = 0;
  front = [];
  lazy_front = lazy [];
  rear_length = 0;
  rear = [];
}
```

```
let check_balance fifo = ...

let push x fifo =
  check_balance
    { fifo with
      rear = x :: fifo.rear;
      rear_length = fifo.rear_length + 1 }

let pop fifo =
  match fifo with
  | { front = [] } -> raise Empty
  | { front = x :: front; lazy_front; } ->
      x,
      check_balance
        { fifo with
          front;
          front_length = fifo.front_length - 1;
          lazy_front = lazy (List.tl (Lazy.force lazy_front))
```

```
let check_head fifo =
  match fifo.front with
  | [] -> { fifo with front = Lazy.force fifo.lazy_front }
  | _ -> fifo

let check_balance fifo =
  let new_fifo =
    if fifo.rear_length < fifo.front_length then
      fifo
    else
      let lazy_front = Lazy.force fifo.lazy_front in
      { front_length = fifo.front_length + fifo.rear_length;
        front = lazy_front;
        lazy_front = lazy (lazy_front @ List.rev fifo.rear);
        rear_length = 0;
        rear = [];
      } in
  check_head new_fifo
```

A efficient implementation for persistency

- The “real” content of the front list is the field `lazy_front`;
- The field `front` is a subpart of `lazy_front` that we have already computed.

- The lazy value created in `check_balance` (and stored in the field `lazy_front`) may be forced either:
  - in check_head (when `front` is empty), or
  - in check_balance (when `rear_length` = `front_length`).
- In both cases, it should not happen before at least *n* interleaved calls to pop or push, where *n* is the value of `rear_length` = `front_length` at the lazy value creation.
- Then, the cost of reversing the list will be shared by all the operation between the lazy creation and its evualation.

- In presence of persistency, the lazy front will be evaluated only once: its cost will be shared by the operation on all ”fork”.

```
type 'a fifo = {
  front: 'a stream;
  rear: 'a list;
  accu: 'a stream;
}

let empty = {
  front = lazy Nil;
  rear = [];
  accu = lazy Nil;
}

let step = ...

let push x fifo = step { fifo with rear = x :: fifo.rear }
let pop fifo =
  match Lazy.force fifo.front with
  | Nil -> raise Empty
  | Cons (x, front) -> x, step { fifo with front }
```

```
let rec rotate fifo =
  match fifo with
  | { front = lazy Nil; rear = [ y ]; accu } ->
    lazy (Cons (y, accu))
  | { front = lazy (Cons (x, xs)); rear = y :: ys; accu } ->
    lazy (Cons (x, rotate { front = xs; rear = ys;
                            accu = lazy (Cons (y, accu)) }))
  | _ -> assert false

let step fifo =
  match Lazy.force fifo.accu with
  | Nil ->
      let front = rotate fifo in
      { front; rear = []; accu = front }
  | Cons (_, accu) -> { fifo with accu }
```

- the field accu is the sub-stream of `front` that has not yet been evaluated
- the function `step` forces the evaluation of a new cell of accu;
- the lazy evaluation being memoized, the function `step` forces *as a side-effect* the evaluation of a new cell of `front`;

- the function `rotate { front; rear; accu; }` produces a stream equivalent to the list `Lazy.force front @ List.rev rear @ Lazy.force accu`.
- the function `rotate` is always called with invariant: the length of `front` is exactly one less than the length of `rear`.

Bibliography

- *Purely Functional Data Structures*, Chris Okasaki, Cambridge University Press.
- Functional Pearls, in Journal of Functional Programming.
  - Gérard Huet, *The Zipper*, Journal of Functionnal Programming, 7(5) :549-554, September 1997.
  - Andrew J. Kennedy, *Pickler Combinators*, Journal of Functionnal Programming, 14(6) :727-739, November 2004.
- *The Functional Approach to Programming with Caml*, Guy Cousineau and Michel Mauny, Cambridge University Press.