Recursion and efficiency

Example 1

```
let rec sum = function (* computes the sum of the elements of a list *)
    hd::tl -> hd + sum tl (* inductive case *)
  | _ -> 0;; (* base case [] *)

let l=List.init 100_000 (fun x->x+1) (*creates [1;2;...;100_000]*)
in sum l;;
- : int = 5000050000

let l=List.init 1_000_000 (fun x->x+1) (*creates [1;2;...;1_000_000]*)
in sum l;;
Stack overflow during evaluation (looping recursion?).
```

Module `List`

- `List` is a predefined OCaml module
- `List.init` is a function defined in `List`
- The documentation of `List` is available at
  `https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

Recursion and efficiency

Example 2

```
let rec reverse = function
    hd::tl -> reverse tl @ [hd] (* inductive case *)
  | _ -> [];; (* base case [] *)

let l=List.init 10_000 (fun x->x+1) (*creates list [1;2;...;10_000]*)
in reverse l;; (* it takes time! *)
```

Time complexity

- `tl @ [hd]` linear in the length of `tl`
- `reverse l` quadratic in the length of `l`!

Recursion and efficiency

Example 3

```
(* Fibonacci numbers *)
let rec fib n = if n<=1 then n else fib(n-2)+fib(n-1);;

(* binomial coefficients *)
let rec bin n k = if n=k||k=0 then 1 else bin(n-1)(k-1)+bin(n-1) k;;
```

Time complexity

- `fib n` is exponential in `n`!
- `bin n n/2` is exponential in `n`!

Accumulators

Can we simulate a loop?

```
(* example with imperative programming, this is not OCaml ! *)
sum(l){
  acc=0; (* initial value of the accumulator *)
  while(l matches with hd::tl){
    acc=acc+hd;
    l=tl;
  }
  (* when l is empty the value in acc is returned *)
  return acc;
}
```

In OCaml

```
let rec loop acc l = match l with (* loop : int -> int list -> int *)
(* if l=hd::tl then increment acc by hd and try the next "loop" with tl *)
    hd::tl -> loop (acc+hd) tl
(* if l=[] then return acc *)
  | _ -> acc
(* loop is called with the initial value of acc *)
in loop 0;; (* loop 0 :  int list -> int *)
```

Tail recursion

Tail recursion in a nutshell

- the recursive application is always the last performed operation
- it can be implemented with a real loop, no stack needed

This is not tail recursion

```
let rec sum = function
    hd::tl -> hd + sum tl (* last operation: addition *)
  | _ -> 0;;
```

This is tail recursion

```
let rec loop acc = function
    hd::tl -> loop (acc+hd) tl (* last operation: recursive application *)
  | _ -> acc
in loop 0;;
```

Accumulators and tail recursion

Full example 1

```
let acc_sum =
  let rec aux acc = function
      hd::tl -> aux (acc+hd) tl
    | _ -> acc
  in aux 0;;

let l=List.init 1_000_000 (fun x->x+1) (*creates [1;2;...;1_000_000]*)
in acc_sum l;;
- : int = 500000500000
```

Remarks

- `aux` is tail recursive with an accumulator
- `aux` is instrumental to the definition of `acc_sum`
- `aux` is an hidden implementation detail
- `acc_sum` applies `aux` and decides the initial value of `acc` (0 in this case)

Accumulators and tail recursion

Full example 2

```
let acc_rev l = (*parameter l needed to get a polymorphic function*)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] l;;

let l=List.init 10_000 (fun x->x+1) (*creates list [1;2;...;10_000]*)
in acc_rev l;;
```

Time complexity

- `hd::acc` takes constant time
- `acc_rev l` is linear in the length of `l`

Remark

Linear time reverse available in module `List`: `List.rev`

Polymorphic functions

Example

```
let acc_rev l = (* acc_rev : 'a list -> 'a list *)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] l;;

acc_rev [1;2;3];;
- : int list = [3; 2; 1]

acc_rev [true;true;false];;
- : bool list = [false; true; true]
```

Remarks

- `acc_rev` has a polymorphic type
- it can be applied to values of different types
- example: `int list` $\neq$ `bool list`

Polymorphic functions

Example

```
let no_poly_rev = (* no_poly_rev : '_weak1 list -> '_weak1 list *)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] ;;

no_poly_rev [1;2;3]
- : int list = [3; 2; 1]

no_poly_rev [true;true;false]
Error: This expression has type bool but an expression was expected of
    type int
```

Remarks

- `no_poly_rev` is **not** polymorphic
- problem with the limitations of the type inference algorithm of OCaml

Strings in OCaml

In a nutshell

- primitive type `string` supported
- standard literals (the only constructors)
  - `""` is the empty string, `"hello world"` is a non-empty string
- concatenation `^`: left-associative, lower precedence than application
- predefined module `String` (`https://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html`)

Examples

```
let s="hello"^" "^"world";;
val s : string = "hello world"
(^);;
- : string -> string -> string = <fun>
String.length s;;
- : int = 11
String.uppercase_ascii s;;
- : string = "HELLO WORLD"
String.lowercase_ascii "HELLO WORLD";;
- : string = "hello world"
```

Generic functions in `List`

Function `map`

- `List.map : ('a -> 'b)-> 'a list -> 'b list`
- `List.map` f `[`x_1`;...;`x_n`]=[`$f\ x_1$`;...;`$f\ x_n$`]`

A possible efficient definition with tail recursion

```
let map f l=
  let rec aux acc = function (* acc contains a list *)
      hd::tl -> aux (f hd::acc) tl (* put f hd on the head of acc*)
    | _ -> acc
  in aux [] (List.rev l);; (* reverse the list *)
```

Remarks

- `::` allows time complexity to be linear in the length of the list (if f is $O(1)$)
- but the list needs to be reversed before or after application of `aux`
- this is a general pattern for functions on lists with accumulator

Generic functions in `List`

Examples of use of function `map`

```
map (fun x->x+1) [1;2;3];;
- : int list = [2; 3; 4]

map String.length ["apple"; "orange" ];;
- : int list = [5; 6]

map String.uppercase_ascii ["apple"; "orange" ];;
- : string list = ["APPLE"; "ORANGE"]
```

Generic functions in `List`

Function `fold_left`

- generic pattern for functions on lists with accumulator
- `List.fold_left : ('a -> 'b -> 'a)-> 'a -> 'b list -> 'a`
- `List.fold_left` f a_0 `[`x_1`;...;`x_n`]` = a_n where:
  - a_0 is the initial value of the accumulator
  - $a_1 = f\ a_0\ x_1$
  - $a_2 = f\ a_1\ x_2$
  - ...
  - $a_n = f\ a_{n-1}\ x_n$
- f `: 'a -> 'b -> 'a` is used to combine
  - the current value of the accumulator (`acc` of type `'a`)
  - the current element of the list (`hd` of type `'b`)

  to get the new value of the accumulator (of type `'a`)

Generic functions in `List`

A possible efficient definition with tail recursion

```
let fold_left f =
  let rec aux acc = function
      hd::tl -> aux (f acc hd) tl
    | _ -> acc
  in aux;;
```

Remarks

The function is tail recursive

Generic functions in `List`

Examples of use of function `fold_left`

```
let sum_list = fold_left (+) 0;; (* (+):int -> int -> int *)
val sum_list : int list -> int = <fun>

sum_list [1;2;3;4];;
- : int = 10

let prod_list = fold_left ( * ) 1;; (* ( * ):int -> int -> int *)
val prod_list : int list -> int = <fun>

prod_list [1;2;3;4]
- : int = 24

let square_list = fold_left (fun acc hd -> acc+hd*hd) 0
val square_list : int list -> int = <fun>

square_list [1;2;3;4]
- : int = 30
```