Accumulators

A standard loop to accumulate a result

```
(* example with imperative programming, this is not OCaml ! *)
sum(ls){
  acc=0; (* initial value of the accumulator *)
  while(true){
    match ls with
       hd::tl -> {acc=acc+hd; ls=tl;}
    |  [] -> return acc
  }
}
```

Simulation in functional programming with OCaml

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

Tail recursion

Definition of tail recursion

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

`sum` is not tail recursive

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

`aux` is tail recursive

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

Accumulators and tail recursion

Efficient definition of sum

```
# let acc_sum =
  let rec aux acc = function
      hd::tl -> aux (acc+hd) tl
    | _ -> acc
  in aux 0;;
val acc_sum : int list -> int = <fun>

# let ls=List.init 10_000 (fun x->x+1)(* ls = [1;2;...;10_000] *)
in acc_sum ls;;
- : int = 50005000
```

Remarks

- `aux` is tail recursive thanks to the accumulator `acc`
- `aux` hides the implementation details of `acc_sum`
- `acc_sum` calls `aux` and passes the initial value of `acc`: 0 in this case

Accumulators and tail recursion

Efficient definition of reverse

```
# let acc_rev ls = (* parameter ls needed to get a polymorphic function *)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] ls;;
val acc_rev : 'a list -> 'a list = <fun>

# let ls=List.init 10_000 (fun x->x+1)(* creates list [1;2;...;10_000] *)
in acc_rev ls;;
- : int list = [10000; 9999; 9998; ...]
```

Time complexity

- `hd::acc` is $O(1)$: constant time
- `acc_rev ls` is $O(n)$: linear in the length n of `ls`

Remark

Efficient reverse defined in module `List`: `List.rev`

Polymorphic functions

Example

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

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` and `bool list` are different types

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
- this is due to the limitations of the type inference algorithm of OCaml

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

Time complexity with respect to the length n of the list

- $O(n)$, if constructor `::` is used and if f is computed in $O(1)$
- but the list needs to be reversed

Generic functions in `List`

Examples of use of function `map`

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

map ((<)0) [0;1;2];; (* remark: (<) 0 equivalent to fun x -> 0<x *)
- : bool list = [false; true; true]

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 defined on lists with an 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` in the next slide)
  - the current element of the list (`hd` of type `'b` in the next slide)

  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;;
```

Remark

Function `aux` is tail recursive thanks to the accumulator `acc`

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
```

Exceptions

Software engineering principles

- faults and unpredictable misbehavior should be reported as soon as possible
- faults and unpredictable misbehavior should be handled at the right moment
- software crashes should be avoided, whenever possible

Motivation

Enhanced software reliability

- more effective way to detect bugs
- better support for fault tolerance

Exceptions

Separation between normal and abnormal behavior

- normal and abnormal execution: normal execution flow should be interrupted as soon as a fault or misbehavior is detected
- values and exceptions:
  - values are the results of computations that complete normally
  - exceptions are special values used to report that a computation cannot complete normally
  - when an exception is raised/thrown, no result is expected
- terminology: to raise/throw an exception means "to report a fault or misbehavior"

Exceptions

High-level constructs to deal with exceptions

Two kinds of constructs to change the control flow in case of exceptions:

- exception generation
- exception handling

Exceptions in OCaml

In a nutshell

- exceptions have general type `exn` and are created with constructors
- exception generation:
  predefined function `raise : exn -> 'a`
- exception handling:
  `try` e **`with`** p_1 `->` e_1 `| ... |` p_n `->` e_n

Remarks

- `raise` does not actually return any value
- the returned type `'a` allows `raise` to be used in any context