Exceptions

General motivations

Important software engineering principles:

- faults and unpredictable misbehavior must be signaled as soon as possible
- faults and unpredictable misbehavior must be handled, but at the right moment
- software crashes must be avoided, whenever possible

Benefits of exceptions

Clear separation between normal behavior and misbehavior

- **normal and abnormal execution**: normal execution flow should be interrupted as soon as a fault/misbehavior is detected
- **values and exceptions**:
  - values are returned only when a computation completes normally
  - exceptions are raised/thrown when a computation **cannot** complete normally
  - when an exception is raised/thrown, no returned value is expected

Benefits of exceptions

High-level constructs to deal with exceptions

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

- exception generation and propagation
- exception handling

Enhanced software reliability

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

Exceptions in OCaml

In a nutshell

- exceptions have general type `exn` and are created with constructors
- exception generation and propagation:
  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 a value is expected

Exceptions in OCaml

Declaration of exception constructors: syntax

```
Dec ::= 'exception' CONS_ID ('of' Type)?
```

Remark: `CONS_ID` must start with an uppercase letter

Declaration of exception constructors: examples

```
exception Fault;; (* constant constructor *)
exception Fault1 of string;; (* a unary constructor *)
exception Fault2 of string*exn;; (* a binary constructor *)
let exc=Fault;;
let exc1=Fault1 "error message";;
let exc2=Fault2 ("msg",exc);;
```

Remarks

- exception constructors behave as expected with the usual laws
- exception constructors are always uncurried

Exceptions in OCaml

Predefined exceptions and functions (a selection)

```
(* self-explanatory, no additional info *)
exception Division_by_zero;;

(* general exception with an error message *)
exception Failure of string;;

(* self-explanatory, associated info: function  name *)
exception Invalid_argument of string;;

(* self-explanatory, associated info: file name, code line and column *)
exception  Match_failure of string*int*int;;

let failwith msg = raise (Failure msg);; (* failwith : string -> 'a *)
```

Exceptions in OCaml

Examples

```
let hd = function
    hd::_ -> hd
  | _ -> failwith "hd";;

(* control flow is changed: x+1 is not evaluated *)
let x=hd [] in x+1;;
Exception: Failure "hd".

(* element at index 4 not found *)
List.nth [1;2;3] 4;;
Exception: Failure "nth".

(* index -1 is not valid *)
List.nth [1;2;3] (-1);;
Exception: Invalid_argument "List.nth".
```

Variant types

In a nutshell

They allow users to define new types with their constructors

Example with only constant constructors

```
type color = Red | Green | Blue;; (* just constant constructors *)

let to_string = function (* to_string : color -> string *)
    Red -> "red"
  | Green -> "green"
  | Blue -> "blue";;

List.map to_string [Red; Blue; Green; Blue];;
- : string list = ["red"; "blue"; "green"; "blue"]
```

Remarks

- type identifiers must start with a lowercase letter
- constructors identifiers must start with an uppercase letter
- constructors cannot be curried

Variant types

Example with constructors of arity > 0

```
type shape = Square of float | Circle of float
      | Rectangle of float * float;;

let perimeter = function  (* perimeter : shape -> float *)
    Square side -> 4.0 *. side
  | Circle ray -> 2.0 *. Float.pi *. ray
  | Rectangle (width,height) -> 2.0 *. (width +. height);;

perimeter (Square 4.0);;
- : float = 16.
```

Standard floating-point numbers

In a nutshell

- predefined type **`float`**
- literals (= constant constructors) with the standard syntax
- standard binary operators `+. -. *. /. **`
- global variables `nan`, `infinity`, `neg_infinity`
- many other features in `Stdlib` (implicitly imported)
- more features in module `Float`

Remarks

- **`int`** and **`float`** not compatible, no implicit conversions
- example

```
(+) : int -> int -> int
(+.) : float -> float -> float
```

```
3.14 * 2;;
^^^^
Error: This expression has type float but an expression was expected of type
    int
```

Variant types

A recursive variant type

```
(* abstract syntax tree implementation *)
type exp_ast = NumLit of int | Sign of exp_ast
      | Mul of exp_ast * exp_ast | Add of exp_ast * exp_ast;;

let rec eval = function (* eval : exp_ast -> int *)
    NumLit n -> n
  | Sign t -> - eval t
  | Mul (t1,t2) -> (eval t1) * eval t2
  | Add (t1,t2) -> (eval t1) + eval t2;;

let ast = Sign(Add(NumLit 40,NumLit 2));;
eval ast;;
- : int = -42
```

Variant types

A polymorphic variant type

```
type 'a option = None | Some of 'a;;

let get = function (* get : 'a option -> 'a *)
    Some v -> v
  | _ -> raise (Invalid_argument "get");;

let find p = (* find : ('a -> bool) -> 'a list -> 'a option *)
    let rec aux = function
        hd::tl -> if p hd then Some hd else aux tl
      | _ -> None
    in aux;;

let v=find ((<) 0) [-1;-2;3];;
val v : int option = Some 3
get v;;
- : int = 3
let v=find ((<) 0) [-1;-2;-3];;
val v : int option = None
get v;;
Exception: Invalid_argument "get".
```

Variant types

A polymorphic and recursive variant type

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

(* member and insert in binary search trees *)

let rec member el = function (* member : 'a -> 'a btree -> bool *)
    Node(label,left,right) ->
        el=label || el<label && member el left || member el right
  | _ -> false;;

let rec insert el = function (* insert : 'a -> 'a btree -> 'a btree *)
    Node(label,left,right) as t ->
        if el=label then t
        else if el<label then Node(label,insert el left,right)
        else Node(label, left, insert el right)
  | _ -> Node(el,Empty,Empty);;
```