Programmation Fonctionnelle
Cours 5
Exceptions

Delia Kesner

Table de matières

Motivations

- Interruption de l'exécution normale du programme.
- Se produit par une mauvaise interaction avec l'environnement extérieur ou quand une fonction est appliquée à des valeurs non attendues (par exemple : division par zéro)
- Pas confondre avec les erreurs de typage !
- Toujours préférable à l'envoi des valeurs légales mais artificielles (comme par exemple renvoyer -1 pour le minimum d'une liste vide)
- Arrêt du programme si l'exception n'est pas rattrapée.

Quelques exceptions prédefinies I

```
(* Predefined exceptions *)

1/0;;
# Exception: Division_by_zero.
List.hd [];;
# Exception: Failure "hd".
List.tl [];;
# Exception: Failure "tl".
String.get "hello" 17;;
# Exception: Invalid_argument "index␣out␣of␣bounds".
List.assoc "z" [("a", 5); ("b", 11)];;
# Exception: Not_found.
```

La solution `failwith` I

```
let rec minlist l = match l with
| [] -> failwith "pas de min dans une liste vide"
| [x] -> x
| h::r -> min h (minlist r);;
# val minlist : 'a list -> 'a = <fun>

let _ = minlist [1;2;3];;
# - : int = 1

let _ = minlist [];;
# Exception: Failure "pas de min dans une liste vide".
```

Le type `option`

- Fournit une réponse quand elle existe bien, et indique s'il n'y a pas de réponse adéquate.
- Corrige la sémantique de quelques types de fonction: le type de la fonction `minliste:'a list -> 'a` n'est pas sémantiquement correct (cf liste vide)
- Similaire à une liste qui serait toujours de taille 0 ou 1.
- La gestion de ces options peut vite devenir pesante et lourde à ecrire.

Le type option I

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

let rec minlist l = match l with   (*un type semantiquement correct*)
| [] -> None
| h::r ->  match minlist r with
             | None   -> Some h
             | Some s -> Some (min h s);;
# val minlist : 'a list -> 'a option = <fun>

minlist [1;2;3];;
# - : int option = Some 1
```

Le type option II

```
minlist [];;
# - : 'a option = None

let safe_div x y = if y=0 then None else Some (x/y);;
# val safe_div : int -> int -> int option = <fun>

5 + safe_div 8 2;;
5 + safe_div 8 2;;
    ^^^^^^^^^^^
Error: This expression has type int option
       but an expression was expected of type int
```

Le type `option` III

```
let safe_add x y = match x, y with (*lourdeur de programmation*)
 | Some x', Some y' -> Some (x'+y')
 | _, _ -> None;;
# val safe_add : int option -> int option -> int option =
  <fun>

safe_add (Some 5) (safe_div 8 2);;
# - : int option = Some 9
```

Le type exn I

```
(* quel est le type d'une exception ? *)
Division_by_zero;;
# - : exn = Division_by_zero
Failure;;
Failure;;
^^^^^^^
Error: The constructor Failure expects 1 argument(s),
       but is applied here to 0 argument(s)
Failure "toto";;
# - : exn = Failure "toto"
```

Le type `exn`

- Les exceptions sont des constructeurs d'un nouveau type `exn`,
- Le type `exn` est une sorte de type somme: les constructeurs commencent par une majuscule et peuvent avoir un argument pour transporter des informations sur l'origine de l'exception.

```
type exn=
| Division_by_zero
| Failure of string
| Invalid_argument of string
| ...
```

- Mais:
  - Pas de polymorphisme
  - *Extensibilité* de la somme: on peut toujours définir de nouvelles exceptions
- Les extensions sont introduites par le mot clef `exception`

Premiers exemples de type `exn` I

```
exception Echec;;
# exception Echec
exception Erreur_fichier of string;;
# exception Erreur_fichier of string
exception E1 of int list;;
# exception E1 of int list
exception E2 of 'a list;;
exception E2 of 'a list;;
                ^^
Error: The type variable 'a is unbound in this type declaration.
```

Lever/déclencher une exception

- Mot clef `raise: exn -> 'a`.
- Le fait qu'une exception est levée ne se voit pas dans le type.
- La levée d'une exception arrête l'évaluation de l'expression.

Raise et liaison statique I

```
exception E of int;;
# exception E of int

let f x = if x < 0 then raise (E x) else x+42;;
# val f : int -> int = <fun>

f (-17);;
# Exception: E (-17).

exception E of bool;; (*liaison statique aussi pour les exceptions *)
# exception E of bool

f (-17);;
# Exception: E (-17).
```

Raise et polymorphisme I

```
exception E of int;;
# exception E of int
exception F of string;;
# exception F of string

let f x = if x < 0 then raise (E x) else x+42;;
# val f : int -> int = <fun>
let g x = if x<"abc" then raise (F x) else "bonjour"^x;;
# val g : string -> string = <fun>

f (-17);;
# Exception: E (-17).
g("ab");;
# Exception: F "ab".
```

Rattraper des exceptions

- ```
  try expr with
  | excep 1 -> exp1
  ...
  | excep n -> exprn
  ```

- Les motifs doivent être du type `exn`.
- Les expressions dans le filtrage doivent être du même type, et ce type doit être égal au type de `expr`.
  ```

Exécution du `try` *e* `with`

1. Si l'évaluation de e donne une valeur v et pas d'exception: la valeur est v.
2. Si l'évaluation de e lève une exception x :
   - 2.1 Filtrage de x par les motifs. Si un motif s'applique, l'expression correspondante est évaluée (rattrapage de x).
   - 2.2 Si aucun des motifs s'applique à x : l'exception n'est pas rattrapée localement.
   - 2.3 L'évaluation d'une expression peut elle-même lever une exception.

Premier exemple try I

```
let dico = [("wonderful", "formidable"); ("is", "est")];;
# val dico : (string * string) list =
  [("wonderful", "formidable"); ("is", "est")]

let translate_mot m =
  try List.assoc m dico
  with Not_found -> m;;
# val translate_mot : string -> string = <fun>

let rec translate_phrase  = List.map translate_mot;;
# val translate_phrase : string list -> string list = <fun>

translate_phrase ["Caml";"is";"wonderful"];;
# - : string list = ["Caml"; "est"; "formidable"]
```

Erreur de typage I

```
let f x y =
  try x/y with
    | Division_by_zero -> "Erreur";;
    | Division_by_zero -> "Erreur";;
                          ^^^^^^^^
Error: This expression has type string
       but an expression was expected of type int
```

```
let f x y =
  try x/y with
    | Division_by_zero -> 0;;
# val f : int -> int -> int = <fun>
```

Exception pas rattrapée I

```
exception E of int;;
# exception E of int
let f x = if x=0 then raise (E 0) else
  if x<10 then raise (E 1) else
    if x < 20 then raise (E 2) else x;;
# val f : int -> int = <fun>
let g x =
  try (string_of_int (f x)) with
    | E 0 -> "zero"
    | E 1 -> "petit";;
# val g : int -> string = <fun>

g 0;;
# - : string = "zero"
```

Exception pas rattrapée II

```
g 5;;
# - : string = "petit"
g 17;; (* exception pas rattrapee *)
# Exception: E 2.
g 42;; (* pas d'exception *)
# - : string = "42"
```

Évaluation de l'expression dans le `try` I

```
let f x =
  try x with
    | Division_by_zero -> -1;;
# val f : int -> int = <fun>

f 18;;
# - : int = 18
f (17/0);;
# Exception: Division_by_zero.

let g x =
  try (17/x) with
    | Division_by_zero -> -1;;
# val g : int -> int = <fun>
```

Évaluation de l'expression dans le `try` II

```
g 3;;
# - : int = 5
g 0;;
# - : int = -1
```

Exemple : Test de complétude d'un arbre binaire

Un arbre binaire est dit *complet* (ou, *équilibré*) si

- Il est une feuille ;
- ou bien si ses deux fils sont
  - complets,
  - et de même hauteur.

Solution sans exceptions et avec parcours multiples I

```
type arbre = F | N of arbre * arbre;;
# type arbre = F | N of arbre * arbre

let rec hauteur a =
  match a with
  | F -> 0
  | N(g,d) -> 1 + max (hauteur g) (hauteur d);;
# val hauteur : arbre -> int = <fun>

let rec complet a =
  match a with
  | F -> true
  | N(g,d) -> (complet g) && (complet d)    (* premier parcours de g,d *)
              && (hauteur g = hauteur d);; (* deuxieme parcours de g,d *)
# val complet : arbre -> bool = <fun>
```

Solution sans exceptions et avec parcours multiples II

```
complet (N(N(F,F),N(F,F)));;
# - : bool = true
complet (N(N(F,F),N(F,N(F,F))));;
# - : bool = false
```

Solution avec exceptions et un seul parcours I

```
type arbre = F | N of arbre * arbre;;
# type arbre = F | N of arbre * arbre
exception Incomplet;;
# exception Incomplet

let complet a =
  let rec haut_aux a = match a with
    | F -> 0
    | N(g,d) ->
      let hg = haut_aux g and hd = haut_aux d in
      if hg = hd then (1 + hg) else raise Incomplet
  in try let _ = haut_aux a in true (*haut_aux a n'est pas de type bool!*)
  with Incomplet -> false;;
# val complet : arbre -> bool = <fun>
```

Solution avec exceptions et un seul parcours II

```
complet (N(N(F,F),N(F,F)));;
# - : bool = true
complet (N(N(F,F),N(F,N(F,F))));;
# - : bool = false
```

Solution avec type option et un seul parcours I

```
type arbre = F | N of arbre * arbre;;
# type arbre = F | N of arbre * arbre
let rec hauteur_option a = match a with
| F      -> Some 0
| N(g,d) -> match hauteur_option g, hauteur_option d with
                | Some hg, Some hd -> if hg = hd then
                                          Some (1+hg) else None
                | _                -> None;;
# val hauteur_option : arbre -> int option = <fun>

let complet t =
  match hauteur_option t with
  | Some _ -> true
  | None   -> false;;
# val complet : arbre -> bool = <fun>
```

Solution avec type option et un seul parcours II

```
complet (N(N(F,F),N(F,F)));;
# - : bool = true
complet (N(N(F,F),N(F,N(F,F))));;
# - : bool = false
```

Syntaxe combinant `match` et `try`

Syntaxe combinant `match` et `try` I

```
exception E of int;;
# exception E of int

let f x = if x='a' then raise (E 0) else
  if x='b' then raise (E 1) else
    if x='c' then raise (E 2) else x;;
# val f : char -> char = <fun>

let g x =  match f x with
| 'd' -> "Lettre␣d"
| 'e' -> "Lettre␣e"
| exception E 0 -> "Lettre␣a"
| exception E 1 -> "Lettre␣b"
| exception E 2 -> "Lettre␣c"
| _             -> "autre";;
```

Syntaxe combinant `match` et `try` II

```
# val g : char -> string = <fun>

g 'a';;
# - : string = "Lettre a"
g 'b';;
# - : string = "Lettre b"
g 'c';;
# - : string = "Lettre c"
g 'd';;
# - : string = "Lettre d"
g 'e';;
# - : string = "Lettre e"
g 'h';;
# - : string = "autre"
```

Assertions

- Mot clef `assert`, suivi d'une expression booléenne
- Lève une exception `Assert_failure` quand l'expression donne false, avec nom du fichier, numéro de ligne et colonne.
- Utile pour déclarer des invariants.
- Utile pour tester des invariants pendant l'exécution.
- On peut désactiver les assertions pendant la compilation d'un programme (`ocamlc -noassert`).
- Premier langage avec assertions : `Eiffel`.

Premier exemple avec assert I

```
let f x =
  assert (x>=0.);
  sqrt x;;
# val f : float -> float = <fun>

f 2.;;
# - : float = 1.41421356237309515
f (-5.);;
# Exception: Assert_failure ("//toplevel//", 2, 2).
```

Deuxième exemple avec assert I

```
let rec fac n =
  assert (n>0);
  if n=1 then 1 else n * fac(n-1);;
# val fac : int -> int = <fun>

fac 4;;
# - : int = 24
fac (-4);;
# Exception: Assert_failure ("//toplevel//", 2, 2).
```