Programmation fonctionnelle, OCaml #4 : gestion d'erreur & exceptions

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Erreurs à l'exécution

Tout programme écrit dans un langage suffisamment puissant peut causer des erreurs (imprévisibles) à l'exécution

Erreurs courantes

- Erreurs arithmétiques (divisions par zéro, dépassement de capacité non souhaités/définis)
- Erreurs d'accès (structure vide, accès hors borne, *fichier* non présent)
- Manque de ressource (dépassement de pile, plus assez de mémoire)

Que faire en cas d'erreur ?

- Rentrer dans un comportement non défini (et on verra…)
- (Essayer de) la détecter et prévenir tout le monde
  - En les forçant plus ou moins à écouter

Table des matières

Options

Type `'a option` : rappel

Définition : `type 'a option = None | Some of 'a`

- Permet de distinguer la *présence* de l'*absence* d'une valeur
- Utilisation *possible* : l'absence de valeur indique que l'évaluation n'a pas pu être terminée
  - Parce qu'un cas d'erreur a été rencontré
- La manipulation d'une `option` se fait nécessairement pas filtrage
  - Un filtrage exhaustif **garantit** que l'on (fait au moins semblant de) traite(r) l'absence de valeur (par ex. les erreurs)

Exemples

- `let head_opt = function x::_ -> Some x | _ -> None`
- `let ckd_div x y = if y = 0 then None else Some (x / y)`

In the wild

Certains types de *nombres à virgule flottante* ont des valeurs non numériques comme `NaN` (proche de `None` par le nom et le rôle)

Variante : *Result*

Type `('a, 'e) result` (pas un rappel)

Définition : `type ('a, 'e) result = Ok of 'a | Error of 'e`

Similaire à un type `option`, mais :

- Explicitement dédié à la gestion d'erreur
- Permet de faire remonter une information en cas d'erreur (par ex. pour en distinguer plusieurs types d'erreurs)
- Pas au programme 🙊

Exemples

- `let head_res = function x::_ -> Ok x | _ -> Error "empty list"`
- ```
  let ckd_div x y = if y = 0 then Error "division by zero"
                             else Ok (x / y)
  ```
  ```

Table des matières

Exceptions

Type exn

OCaml possède un type `exn` de *valeurs exceptionnelles*, ou *exceptions*. Celles-ci se présentent comme des types somme à un constructeur non polymorphe

Définition

On peut définir ses propres exceptions avec la syntaxe :

```
exception Nom (of <type des arguments>)
```

Exemples

La bibliothèque standard prédéfinit un certain nombre d'exceptions, dont :

- `exception Failure of string`
- `exception Invalid_argument of string`
- `exception Match_failure of (string * int * int)`
- `exception Assert_failure of (string * int * int)`
- `exception Not_found`
- `exception Out_of_memory`
- `exception Stack_overflow`

Utilisation

Levée d'exception

Une exception peut être *levée* en lui appliquant la fonction `raise : exn -> 'a`

- Le type de `raise` indique que son évaluation ne termine jamais à moins de casser le système de type, ce qui n'est pas le cas
  - Comme pour une fonction de type `'a -> 'b`
  - D'ailleurs : `((fun x -> raise Not_found): 'a -> 'b)`

Lorsqu'une exception est levée, le flot d'exécution normal du programme est interrompu jusqu'à ce que le programme tout entier termine brutalement, ou que l'exception soit *rattrapée*

Démo

```
exception Ohai
let rec surpraise = function
    | [] -> []
    | x::xs -> if x = 12 then raise Ohai else x::(surpraise xs)
```

Utilisation *bis*

Rattrapage d'exception

Une exception se rattrape avec un filtrage de motif adapté :

`try en with <Ex1> -> ex1 | <Ex2> -> ex2 ...`

qui s'évalue en :

- `en` si aucune exception ne s'échappe lors de son évaluation
- `ex1` si une exception correspondant au motif `<Ex1>` est levée (et pas déjà rattrapée) lors de son évaluation
- en `ex2` si une exception correspondant au motif `<Ex2>` (mais pas au motif `<Ex1>`) est…

Contrainte de typage : `en`, `ex1`, `ex2`… toutes de même type

Exemple

```
try surpraise x with Ohai -> x | e -> raise e
```

- On rattrape l'expression connue `Ohai` avec un traitement adapté, et lève à nouveau (« propage ») toute éventuelle autre exception (pas besoin d'être fait explicitement)

Rattraper une exception… et ensuite ?

- Pas toujours facile de savoir quoi faire d'une exception qui a été rattrapée
  - Souvent, on ne peut guère faire mieux que la lever à nouveau
  - Dans ce cas, autant s'abstenir
- Les rattrapages sont le plus souvent utiles quand :
  - L'exception a été prise en compte lors de la conception (voire est *souhaitable*, ou inévitable), ou
  - Du *nettoyage* est nécessaire en cas d'erreur (par exemple pour libérer des ressources)

Exemples d'exceptions pas toujours rattrapées

- `Failure`
  - Déclenchable à la main ou par `failwith: string -> 'a`
- `Assert_failure`
  - Déclenchable à la main ou (surtout) par l'échec d'une assertion
- `Match_failure`
  - Déclenchable à la main ou (surtout) par l'évaluation d'un filtrage non exhaustif sur un motif non prévu

Intermède : `assert` en OCaml

OCaml fournit une construction `assert` qui

- N'est pas (exactement) une fonction
  - Comme en C
- Prend une expression booléenne en argument
- Ne fait rien (avec type `unit`) si l'expression s'évalue à `true`
- Lève une exception `Assert_failure` si l'expression s'évalue à `false`
  - L'exception contient des informations de localisation

Utilisations typiques

- Pour vérifier/indiquer des invariants (comme en C)
  - Nous verrons un peu plus tard comment facilement ajouter des `assert` dans le flot d'un programme (mais si vous êtes malin, vous pouvez deviner une façon)
  - (Aussi désactivable à la compilation)
- Pour traiter un cas impossible dans un `match with`
  - Aussi un invariant
  - (En C23, correspondrait à `unreachable()`)

Exceptions v. Options pour la gestion d'erreur

Options

- Intégrées au flot d'exécution habituel
- « Visibles » dans le typage
- Forcent la gestion des cas d'erreur (nécessité de filtrage, nécessairement exhaustif)
- Peut être plus lourd à l'utilisation que les exceptions, en particulier quand
  - On peut prouver qu'il n'y a pas d'erreur
  - On ne peut pas traiter les problèmes localement (tout le monde se retrouve avec des options… mais des solutions existent pour écrire ça *joliment*)

Exceptions

- Risque d'erreur à l'exécution en cas de non rattrapage
- Compliquent le flot d'exécution
- Plus léger en cas d'absence d'erreur, de traitement tardif (pas besoin de propager l'exception *à la main* : il suffit de la rattraper au bon endroit)

Exceptions pour le contrôle de flot

Les exceptions peuvent aussi être utilisées pour pallier l'absence de `return` (et `break` / `continue`) en OCaml

Sur un exemple

```
let rec prod = function
    | [] -> 1
    | x::xs -> if x = 0 then 0 else x * (prod xs)

exception Zero
let rec _prod' = function
    | [] -> 1
    | x::xs -> if x = 0 then raise Zero else x * (prod xs)
let prod' x = try _prod' x with Zero -> 0
```

(Plus plus tard, quand on aura des boucles…)