Programmation Fonctionnelle (PF)

INFO4

Cours 3 : évaluation, exceptions

Jean-François Monin, Benjamin Wack

POLYTECH GRENOBLE

Questions

Quelle est la différence entre

- $5 + 4$, 3×3, 9 et $\sqrt{81}$?
- false et true && not true
- $157,03 + 89,75 - 4 \times 45,35$, $65,38$ et $\dfrac{(124,63 + 38,82)}{2,5}$?

Réponses

1. Aucune ! (mêmes valeurs)
2. Essayez de faire un chèque de $157,03 + 89,75 - 4 \times 45,35$
3. Essayez de comparer directement $157,03 + 89,75 - 4 \times 45,35$ et $\frac{(124,63 + 38,82)}{2,5}$?

Expression et valeur

$5 + 4$, 3×3, 9, $\sqrt{81}$,
false, true && not true
$157,03 + 89,75 - 4 \times 45,35$, $\dfrac{(124,63 + 38,82)}{2,5}$ et $65,38$
sont des exemples d'*expressions*
Parmi celles-ci, seules 9, false et $65,38$ sont des *valeurs*.

Comment les distinguer ?

Theorem

Deux valeurs sont égales si et seulement si elles sont :
littéralement – syntaxiquement – identiques.

Évaluation

Par définition, *évaluer* $\underline{\underline{\text{déf}}}$ *déterminer la valeur*

Application

Algorithme de comparaison entre deux expressions E_1 et E_2 :

- évaluer $E_1 \rightarrow$ on obtient v_1
- évaluer $E_2 \rightarrow$ on obtient v_2
- comparer syntaxiquement v_1 et v_2

Comment évaluer

On *réduit* progressivement
$\rightarrow$ relation de *réduction* notée $\triangleright$

Exemple

(Syntaxe OCaml pour les flottants)

```
 157.03 +. 89.75 -. 4. *. 45.35
▷
 246.78 -. 4. *. 45.35
▷
 246.78 -. 181.40
▷
 65.38
```

Évaluation d'une expression

Expression =

- valeur $\Rightarrow$ rien à calculer
- fonction appliquée à des arguments (sous-expressions) :
  - évaluation des arguments a_i dans un ordre non spécifié
  - application de la fonction aux valeurs obtenues

Exemple

mult (plus 2 3) (succ (moins 5 (-3)))
▷ mult (plus 2 3) (succ 8)
▷ mult 5 (succ 8)
▷ mult 5 9
▷ 45

Expression conditionnelle

Expression =

- valeur (exemple : un entier)
- fonction appliquée à des arguments : $f\ a_1\ \ldots\ a_n$
- `if` B `then` E_1 `else` E_2
- *autres possibilités vues plus tard*

Évaluation d'une expression conditionnelle

- évaluation de B, i.e. $B \triangleright V$
- $V = \mathsf{true}$, évaluation de E_1, i.e. $E_1 \triangleright V_1$
  (`if` B `then` E_1 `else` E_2) $\triangleright\ V_1$
- $V = \mathsf{false}$, évaluation de E_2, i.e. $E_2 \triangleright V_2$
  (`if` B `then` E_2 `else` E_2) $\triangleright\ V_2$

Expression en **let**

Sous-expressions semblables

$(2 + 37938573 \times 4869582/8576 - 27 \times 31) \times$
$\text{succ}(5 \times 27 - 37938573 \times 4869682/8576)$

- illisible , voire dangereux : on peut se tromper
- inefficace : évaluation effectuée à deux reprises

Solution : nommer des valeurs calculées

let gros $= 37938573 \times 4869582/8576$

$(2 + \text{gros} - 27 \times 31) \times \text{succ}(5 \times 27 - \text{gros})$

Ou introduction d'un nom local

let *gros* $= 37938573 \times 4869582/8576$ `in`

$(2 + \textit{gros} - 27 \times 31) \times \text{succ}(5 \times 27 - \textit{gros})$

Environnement (contexte d'évaluation)

Définition

environnement
$\underline{\underline{\text{déf}}}$
ensemble d'associations nom $\mapsto$ valeur

L'environnement est parfois appelé *contexte d'évaluation*

En programmation fonctionnelle

- associations **définitives** dans la portée considérée
- les valeurs sont **immuables**

Évaluation dans un environnement

Expression =

- valeur (exemple : un entier)
- nom
- fonction appliquée à des arguments : $f\ a_1 \ldots a_n$
- let-expression : **let** nom = *expr1* `in` *expr2* let-expression : let nom = *expr1* in *expr2*

Évaluation avec environnement

- valeur $\Rightarrow$ rien à calculer
- nom x $\Rightarrow$ valeur V trouvée dans l'environnement (l'environnement **doit** contenir $x \mapsto$ V)
- fonction appliquée à des arguments (sous-expressions) :
  - évaluation des arguments a_i dans un ordre non spécifié
  - *un petit quelque chose, voir plus loin*
  - application de la fonction aux valeurs obtenues
- let-expression : cf. suite

Évaluation d'un **let** ... `in` ...

let nom = *expr1* `in` *expr2*

- évaluation de *expr1* : $\triangleright$ *val1*
- environnement augmenté = ancien environnement + nom $\mapsto$ *val1*
- évaluation de *expr2* dans l'environnement augmenté

Évaluation d'un **let** ... in ...

Exemple 1

let $y = 2+3$ in **let** $x = 32 \times 3$ in **let** $z = 75/3$ in f x y z		
$[y \mapsto 5]$	**let** $x = 32 \times 3$ in **let** $z = 75/3$ in f x y z	
$[y \mapsto 5, x \mapsto 96]$	**let** $z = 75/3$ in f x y z	
$[y \mapsto 5, x \mapsto 96, z \mapsto 25]$	f x y z	
$[y \mapsto 5, x \mapsto 96, z \mapsto 25]$	f 96 5 25	
$[y \mapsto 5, x \mapsto 96, z \mapsto 25]$	*...(application de la fonction)*	

Exemple 2

let $y = 2+3$ in **let** $x = 32 \times 3$ in **let** $z = x+y$ in f x y z		
$[y \mapsto 5, x \mapsto 96]$	**let** $z = x+y$ in f x y z	
$[y \mapsto 5, x \mapsto 96, z \mapsto 101]$	f x y z	
$[y \mapsto 5, x \mapsto 96, z \mapsto 101]$	f 96 5 101	
$[y \mapsto 5, x \mapsto 96, z \mapsto 101]$	*...(application de la fonction)*	

Variables locales : let ... in

```
# let x = 44 and y = 2 in x - y ; ;
- : int = 42
# x ; ; Unbound value x
# y ; ; Unbound value y
```

Contrôle de l'ordre d'évaluation

Remarque

L'ordre d'évaluation dans un **let** ... `in` ... est bien déterminé

- sans grande importance dans un cadre purement fonctionnel
- important en cas d'*effets de bord*
  - Entrées/sorties
  - Exceptions

(sera approfondi plus tard)

Encore des valeurs : les fonctions

Exemple : la fonction successeur

Notation ML (Ocaml) : `fun x` $\rightarrow x + 1$
Notation mathématique abrégée : $\lambda x.\ x + 1$

Évaluation

On considère l'application d'une valeur fonctionnelle $\lambda x.$ *corps*
à un argument *arg*
$(\lambda x.$ *corps*$)$*arg* se réduit comme
let $x =$ *arg* `in` *corps*

- *corps* et *arg* sont des expressions
- en OCaml, *arg* est en fait une valeur
  (stratégie dite « stricte »)

Encore des valeurs : les fonctions

Exemple de réduction

$\textbf{let}\ suc = \texttt{fun}\, x \to x + 1\ \texttt{in}\ \textbf{let}\ a = 12\ \texttt{in}\ suc\ (a + a)$	
$[suc \mapsto \lambda x.\ x + 1]$	$\textbf{let}\ a = 12\ \texttt{in}\ suc\ (a + a)$
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12]$	$suc\ (a + a)$
(evaluation de $a + a$ *:* $a + a \triangleright a + 12 \triangleright 12 + 12 \triangleright 24$*)*	
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12]$	$suc\ 24$
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12]$	$(\lambda x.\ x + 1)\ 24$
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12, x \mapsto 24]$	$x + 1$
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12, x \mapsto 24]$	$24 + 1$
$[suc \mapsto \lambda x.\ x + 1,\ a \mapsto 12, x \mapsto 24]$	25

Retour sur l'évaluation

Expression =

- valeur (exemple : un entier)
- nom
- fonction appliquée à des arguments : $f\ a_1\ \ldots\ a_n$
- let-expression : let nom = *expr1* in *expr2*

Évaluation

- valeur $\Rightarrow$ rien à calculer
- nom x $\Rightarrow$ valeur V trouvée dans l'environnement
- fonction appliquée à des arguments (sous-expressions) :
  - évaluation des arguments a_i dans un ordre non spécifié
  - *évaluation de f*
  - application de la fonction *obtenue* aux valeurs obtenues
- **let** x = *expr1* `in` *expr2* : évaluation de *expr2* avec $x \mapsto V_1$

Définition récursive de fonction

Exemple

```
let rec fact = fun n →
    if n = 0 then 1 else n * fact (n − 1)
```

Évaluation

- valeur $\Rightarrow$ rien à calculer (rem : `fun` y $\rightarrow$ *expr* est une valeur !)
- ...
- **let** x = *expr1* `in` *expr2* : évaluation de *expr1* $\triangleright V_1$, puis de *expr2* dans l'environnement augmenté de $x \mapsto V_1$
- **let** `rec` x = *expr1* `in` *expr2* : évaluation de *expr2* dans l'environnement augmenté de $x \mapsto$ *expr1* OK si *expr1* est de la forme `fun` y $\rightarrow$ *expr*

Exemple simple

let `rec` *x* = *expr1* `in` *expr2* :
évaluation de *expr2*
dans l'environnement augmenté de $x \mapsto$ *expr1*

Comment évaluer :

```
# let rec x = 0 :: x in x;;
```

$[x \mapsto 0 :: x]$ $\quad 0 :: x$

Exemple simple

Que fait OCaml ?

```
# let rec x = 0 :: x;;
val x : int list =
[0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; ...]
(profondeur de l'affichage)
```

```
let rec long l =  match l with [] -> 0
                             | x :: l -> 1 + long l;;
```

```
# long x;;
Stack overflow during evaluation (looping recursion ?).
```

Exemple fact 3

let `rec` $fact$ = `fun` n $\rightarrow$ `if` $n = 0$ `then` 1 `else` n * $fact\ (n-1)$
`in` fact 2 ; ;

$\triangleright$ [$fact \mapsto \lambda\ n.$ `if` $n = 0$ `then` 1 `else` n * $fact\ (n-1)$] fact 2

évaluation de fact 2 dans l'environnement [$fact \mapsto \lambda\ n. \ldots$]

fact 2
$\triangleright$ [$n \mapsto 2$] `if` $n = 0$ `then` 1 `else` n * $fact\ (n-1)$
$\triangleright$ [$n \mapsto 2$] n * $fact\ (n-1)$
$\triangleright$ [$n \mapsto 2$] 2 * fact 1
$\triangleright$ [$n \mapsto 2,\ n \mapsto 1$] 2 * (`if` $n = 0$ `then` 1 `else` n * $fact\ (n-1)$)
$\triangleright$ [$n \mapsto 2,\ n \mapsto 1$] 2 * (n * $fact\ (n-1)$)
$\triangleright$ [$n \mapsto 2,\ n \mapsto 1$] 2 * 1 * fact 0
$\triangleright$ [$n \mapsto 2,\ n \mapsto 1,\ n \mapsto 0$] 2 * 1 * (`if` $n = 0$ `then` 1 `else` n * $fact\ (n-1)$)
$\triangleright$ [$n \mapsto 2,\ n \mapsto 1,\ n \mapsto 0$] 2 * 1 * 1
$\triangleright$ 2

Que faire en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

# List.hd [ ];;
```

Une solution : renvoyer un couple (valeur, booléen)

- (*a*, `true`) signifie : « la fonction a réussi et son résultat est *a* »
- (*a*, `false`) signifie : « la fonction a échoué »
  (et *a* n'a pas de sens)

Limite : propagation.

Que faire en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

  Exception: Division_by_zero.

# List.hd [ ] ;;

  Exception: Failure "hd".
```

Solution OCaml : les exceptions

failwith

```
# failwith;;
- : string -> 'a = <fun>
```

Exemple

```
# let tete = fun l -> match l with
    []   -> failwith "tete vide"
  | x::s -> x ;;

val tete : 'a list -> 'a = <fun>

# tete [];;
Exception: Failure "tete vide".
```

Autre possibilité : exception spécifique

raise

```
exception ListeVide

let tete = fun l → match l with
    | x :: s → x
    | [] → raise ListeVide
```

```
# tete []
Exception: ListeVide.
```

Interrompt l'ordre habituel de l'évaluation
Est propagée dans les fonctions appelantes

Le type des exceptions

ListeVide : de type exn
raise ListeVide : de type déterminé en fonction du contexte

exn est :

- un type somme particulier
- étendu dynamiquement :
  exception ListeVide introduit un nouveau constructeur de exn.

Important

Une exception peut transporter de l'information :

exception Alarme of `int *bool`

Syntaxe des exceptions

```
# exception Erreurfatale of string;;
exception Erreurfatale of string
# raise (Erreurfatale "crash compilateur");;
Exception: Erreurfatale "crash compilateur".
```

Attention à la Majuscule

Deux lanceurs d'exceptions prédéfinis

failwith "toto" $\equiv$ raise (Failure "toto")

invalid_arg "toto" $\equiv$ raise (Invalid_argument "toto")

Récupération d'exception (évaluation)

T = `try` E `with`
| ListeVide $\rightarrow E_1$
| Alarme $(n,\ b) \rightarrow E_2(n, b)$
| Not_found $\rightarrow E_3$

(une expression)

Évaluation

E est évalué en premier, puis :

- si aucune exception levée : valeur de T = valeur de E
- si exception e levée, on la compare aux motifs du `with`
  - si filtrage réussi au i^{e} motif : valeur de T = valeur de E_i
  - sinon l'exception e est transmise à la fonction appelante (comme s'il n'y avait pas de `try`)

Exemple

Écrire une fonction prenant une liste d'entiers et renvoyant la somme des ses éléments ; si la liste contient un négatif alors renvoyer -1.

Un premier programme

```
let rec somme = fun l -> match l with
 | []    -> 0
 | x :: r -> if x < 0 then -1 else x + (somme r)
```

ne fonctionne pas

Solution 1 : sans exception

```
let somme = fun l ->
  let rec testpos = fun l -> match l with
    | []     ->  true
    | x :: r ->  x > 0  &&  (testpos r)
  and let rec somme_aux = fun l -> match l with
    | []     ->  0
    | x :: r ->  x  +  (somme_aux r)
  in if  testpos l  then  somme_aux l  else  -1 ;;
```

Problème : 2 parcours de la liste

Solution 2 : sans exception

```
let rec somme = fun l -> match l with
    | []     ->  0
    | x :: r ->  let s = somme r
                 in if  s = -1 || x < 0
                    then  -1
                    else  x + s
```

Problème : mélange entre code fonctionnel et gestion d'erreur
test `s = -1` systématique

Solution 3 : avec exception

```
exception Negatif;;

let somme = fun l ->
  let rec somme_aux = fun l -> match l with
    | []                -> 0
    | x::_ when x < 0 -> raise Negatif
    | x::r              -> x + somme_aux r
  in try
    somme_aux l
  with
    Negatif -> -1;;
```

Séparation du code fonctionnel et de la gestion d'erreur

Application : utilisation d'exceptions en mise au point

Objectif : se ramener à des problèmes plus petits traités un à un

```
exception PasEncoreDef of string *string

let rec oppose = fun i →
  match i with
  | Ent (e) →  raise PasEncoreDef ("oppose", "Ent")
  | Reel (r) →  raise PasEncoreDef ("oppose", "Reel")
  | Cplx (r,i) →  raise PasEncoreDef ("oppose", "Cplx")
```

On peut alors

- tester le programme global dès le départ
- proposer le vrai code pour chaque cas pris séparément

Application : tester ses programmes

Un test = une entrée possible de la fonction + le résultat attendu

```
let _ = assert (f entree = resultat)
```

Couverture

- tests représentatifs de toutes les données acceptables
- permettant d'observer tous les résultats possibles
- ne pas oublier les cas extrêmes : liste vide, 0, etc.

Les données incorrectes (mauvais type, entier négatif...) relèvent de la *robustesse*, moins importante ici.

Conserver les tests et les réévaluer à chaque modification

- Non régression
- Identification plus aisée des bugs