Approches Fonctionnelles de la Programmation

Introduction

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Table des matières

Un paradigme de programmation

« Quoi faire » plutôt que « Comment faire »

Un paradigme ?

- Affecte l'expressivité d'un langage
- Affecte la manière de penser *dans* un langage
- Le concept de paradigme est poreux. . .

Lequel ?

- Expressions
- Définitions (expressions nommées)
- Évaluations (de définitions ou d'expressions)

De l'impératif au fonctionnel

« La somme des carrés des entiers entre 1 et N »

C (impératif)

```
int ssq (int n)
{
  int i = 1, a = 0;

  while (i <= n)
    {
      a += i*i;
      i += 1;
    }

  return a;
}
```

C (récursif)

```
int ssq (int n)
{
  if (n == 1)
    return 1;
  else
    return n*n + ssq (n-1);
}
```

Lisp

```
(defun ssq (n)
  (if (= n 1)
      1
      (+ (* n n) (ssq (1- n)))))
```

Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- Clarté
- Concision

L'impératif vu à l'envers

« La racine carrée de la somme des carrés de a et de b »

C (impératif)

```
float hypo (float a, float b)
{
  float a2 = a*a;
  float b2 = b*b;
  float s = a2 + b2;

  return sqrt (s);
}
```

C (moins impératif)

```
float hypo (float a, float b)
{
  return sqrt (a*a + b*b);
}
```

Haskell

```
hypo :: Float -> Float -> Float
hypo a b = sqrt ((a*a) + (b*b))
```

Lisp

```
(defun hypo (a b)
  (sqrt (+ (* a a) (* b b))))
```

Pour être tout à fait honnête...

Haskell (100% préfixe)

```
hypo :: Float -> Float -> Float
hypo a b = sqrt ((+) ((*) a a) ((*) b b))
```

La fonction : un objet de 1re classe

Christopher Strachey (1916-1975)

. . . du 1er ordre, d'ordre supérieur. . .

- stockage (variables)
- agrégation (structures)
- argument de fonction
- retour de fonction
- manipulation anonyme
- construction dynamique
- . . .

Plus d'expressivité (clarté, concision *etc.*)

Nommage par une variable

Que peut-on manipuler ?

C

```c
/* 3 + 4 */

int plus (int a, int b)
{
  return a + b;
}
/* plus (3, 4); */

typedef int (* foo_f) (int, int);
foo_f func = plus;
/* (*func) (3, 4); */
```

- pointeurs sur fonction

Haskell

```haskell
-- 3 + 4
-- (+) 3 4

func :: Int -> Int -> Int
func = (+)
-- func 3 4
```

Lisp

```lisp
;; (+ 3 4)

(setf func #'+)
;; (funcall func 3 4)
;; (funcall #'+ 3 4)

(setf (symbol-function 'func) #'+)
;; (func 3 4)
```

Scheme

```scheme
;; (+ 3 4)

(define func +)
;; (func 3 4)
```

- fonctions elles-mêmes

Arguments fonctionnels
Mapping et folding

Deux Archétypes du passage d'argument fonctionnel :

- **mapping :** traiter individuellement les éléments d'une liste par une fonction.

Lisp

```
(mapcar #'sqrt '(1 2 3 4 5))
```

Haskell

```
map sqrt [1..5]
```

- **folding :** combiner les éléments d'une liste par une fonction.

Lisp

```
(reduce #'+ '(1 2 3 4 5))
```

Haskell

```
foldr1 (+) [1..5]
sum [1..5]
```

Application

« La somme des carrés des entiers entre 1 et N » II, le retour

- **Autre vision algorithmique (non récursive) :**
  « sommer la liste des carrés de 1 à N »

Lisp

```
(defun sq (x) (* x x))

(defun intlist (n)
  (if (= n 1)
      (list 1)
      (cons n (intlist (1- n)))))

(defun ssq (n)
  (reduce #'+ (mapcar #'sq (intlist n))))
```

Haskell

```
sq :: Int -> Int
sq x = x * x

ssq :: Int -> Int
ssq n = sum (map sq [1..n])
```

- **Gain en abstraction :**
  - définition de *ssq* encore plus concise
  - mise en évidence d'abstraction supplémentaire

Fonctions anonymes

Des littéraux comme les autres. . .

- **Possibilité de *ne pas* nommer les fonctions :**

Lisp

```
(lambda (x) (* 2 x))
```

Haskell

```
\x -> 2 * x
```

- **Utilisation directe (littérale) :**
  au même titre que les `int`, les chaînes de caractères *etc.*

Lisp

```
((lambda (x) (* 2 x)) 4)
```

Haskell

```
(\x -> 2 * x) 4
```

Application

« La somme des carrés des entiers entre 1 et N » III, le retour de la vengeance

- **Économie de nommage :**
  éviter le surpeuplement de petites fonctions.

Lisp

```
(defun ssq (n)
  (reduce #'+ (mapcar ;; #'sq
                      (lambda (x) (* x x))
                      (intlist n))))
```

Haskell

```
ssq :: Int -> Int
ssq n = sum (map -- sq
                 (\x -> x * x)
                 [1..n])
```

Retours fonctionnels

Mais surtout, construction de fonctions à la volée

- **Le retour fonctionnel est inutile...**

C

```c
int plus  (int a, int b);
int minus (int a, int b);

typedef int (* foo_f) (int, int);

foo_f p_or_m (int which_one)
{
  return (which_one == 0)
    ? plus : minus;
}

/* (* p_or_m (0)) (4, 2); */
```

Lisp

```lisp
(defun +/- (which-one)
  (if (= which-one 0)
      #'+
    #'-))

;; (funcall (+/- 0) 4 2)
```

Haskell

```haskell
p_or_m :: Int -> (Int -> Int -> Int)
p_or_m 0 = (+)
p_or_m _ = (-)

-- (p_or_m 0) 4 2
```

Retours fonctionnels

Mais surtout, construction de fonctions à la volée

- **. . . sans construction au vol :**
  D'où l'importance des fonctions anonymes. . .

Lisp

```
(defun adder (n)
  (lambda (x) (+ x n)))

;; (funcall (adder 3) 1)
```

Haskell

```
adder :: Int -> (Int -> Int)
adder n = \x -> x + n

-- (adder 3) 1
```

Lisp

```
(defun twice (f)
  (lambda (x)
    (funcall f (funcall f x))))

;; (funcall (twice #'sqrt) 16)
```

Haskell

```
twice :: (a -> a) -> (a -> a)
twice f = f . f

-- (twice sqrt) 16
```

Application

« La somme des carrés des entiers entre 1 et N » IV, Apocalypse

- **Gain en abstraction considérable :**
  Il devient immédiat de définir d'autres combinaisons utiles.

Lisp

```
(defun foldmap (f m)
  (lambda (x)
    (reduce f (mapcar m x))))

(defun ssq (n)
  (funcall
   (foldmap #'+ (lambda (x)
                  (* x x)))
   (intlist n)))
```

Haskell

```
foldmap :: (c -> c -> c) -> (c -> c)
        -> ([c] -> c)
foldmap f m = foldr1 f . (map m)

ssq :: Int -> Int
ssq n = foldmap (+) (\x -> x*x) [1..n]
```

Pseudo-1er ordre dans les langages impératifs

On fait ce qu'on peut...

- Les structures de contrôle impératives...
  sont des formes fixes de fonctions d'ordre supérieur.

```
if (expression)
{   /* LAMBDA PROCEDURE ! */
  /* blah blah ... */
}
else
{   /* LAMBDA PROCEDURE ! */
  /* blah blah ... */
}
```

Programmation fonctionnelle pure

La fonction au sens mathématique

$$ssq(x) = \begin{cases} 1 & \text{si } x = 1, \\ x^2 + ssq(x-1) & \text{sinon.} \end{cases}$$

Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

Fonction :

- Impératif : **procédure**. Ensemble de calculs à effets de bords avec *éventuellement* retour d'une valeur.
- Fonctionnel pur : calcul d'une valeur de sortie (retour) en fonction de valeurs d'entrée (arguments).

Variable :

- Impératif : représente un stockage d'information qui varie au cours du temps (« mutation »).
- Fonctionnel pur : **constante**. Représente une valeur inconnue ou arbitraire. Chaque occurrence est interprétée de la même manière.

Intérêts de la pureté

Pureté $\Longleftrightarrow$ Sureté

- **Parallélisme**
  - Cf. Erlang
- **Sémantique locale aux fonctions**
  - Tests locaux / Bugs locaux
- **Preuve de programme**

Preuves formelles
Induction mathématique

« Prouvez-moi (s'il vous plaît) que $\forall N, ssq(N) > 0$ »

Fonctionnel pur :

Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- C'est vrai au rang 1
- Supposons que ce soit vrai au rang $N-1\ldots$

Impératif :

C

```
int ssq (int n)
{
  int i = 1, a = 0;

  while (i <= n)
    {
      a += i*i;
      i += 1;
    }

  return a;
}
```

- Euh...

Les limites du formalisme mathématique

Déclaratif *vs.* impératif

Comment exprimer le concept de « racine carrée » ?

$$sqrt(x) = y | \begin{cases} y > 0 \\ y^2 = x \end{cases}$$

Lisp

```
(defun sqrtp (s x)
  (and (> s 0)
       (= (* s s) x)))
```

Lisp

```
(defun sqrt (x) ???)
```

Haskell

```
sqrtp :: Float -> Float -> Bool
sqrtp s x = s > 0 && s*s == x
```

Haskell

```
sqrt :: Float -> Float
sqrt x = ???
```

- Au final, il faut bien expliquer *comment faire*...
- Mais on repousse le problème :
  impératif ou fonctionnel pur ?

Évaluation stricte / lazy

Quand calculer la valeur d'une expression ?

- **Stricte :** Lisp
  Les arguments (expressions) sont évalués d'abord.
- **Lazy** (paresseuse) : Haskell
  Les expressions ne sont évaluées que quand le besoin s'en fait sentir, (idem pour les agrégats).

La paresse : une vertu ?

- **Intérêt :** plus d'abstraction (ex. manipulation de listes infinies).
- **Contrainte :** pureté fonctionnelle requise (on ne peut pas s'appuyer sur l'ordre d'évaluation).

Application

« La somme des carrés des entiers entre 1 et N » V, Rédemption

« À partir de la liste des entiers, calculer la somme des carrés jusqu'à N »

Lisp

```
(defun intlist (s)
  (cons s (intlist (1+ s))))

(defun ssql (l n)
  (if (= (car l) n)
      (* n n)
      (+ (* (car l) (car l))
         (ssql (cdr l) n))))

(defun ssq (n)
  (ssql (intlist 1) n))

;; Coffee time...
;; ^C^C^C^C !!!!!!
```

Haskell

```
intlist :: Int -> [ Int ]
intlist s = s : intlist (s + 1)

ssql :: [ Int ] -> Int -> Int
ssql (x:xs) n = if (x == n)
                then n*n
                else x*x + ssql xs n

ssq :: Int -> Int
ssq n = ssql (intlist 1) n


-- Not coffee time, but...
-- Stack overflow !!
```

Pseudo-paresse dans les langages impératifs

On ne fait toujours que ce qu'on peut...

- Les structures de contrôle impératives...
  sont des formes embryonnaires d'évaluation lazy.

```
if (1)
{   /* COMPUTED */
  /* blah blah ... */
}
else
{   /* NOT COMPUTED ! */
  /* blah blah ... */
}
```

Typage dans les langages fonctionnels
Et dans les autres. . .

Problème orthogonal à la programmation fonctionnelle
Il n'y a qu'à regarder les politiques d'Ada, C, Ruby, PHP. . .

- **Typage statique : Haskell**
  - Les *variables* sont typées
  - Vérification de type à la *compilation*
- **Typage dynamique : Lisp**
  - Les *valeurs* sont typées
  - Vérification de type à l'*exécution*
  - Common Lisp : typage explicite possible
- **Terminologie floue :** typage statique, dynamique, manifeste, fort, latent, doux *etc.*

Typage et polymorphisme
Quelle forme de généricité ?

- **Les contraintes du typage statique...**

Lisp

```
(defun invert (l)
  (unless (null l)
    (append (invert (cdr l))
            (list (car l)))))

;; (invert '(1 2 3 4))
;; (invert '(foo 3.6 "blah" 2))
```

Haskell

```
invert :: [ Int ] -> [ Int ]
invert [] = []
invert (x:xs) = (invert xs) ++ [x]

-- invert [1..5]
```

- **...sont compensées par le polymorphisme :**
  Mais les listes restent homogènes...

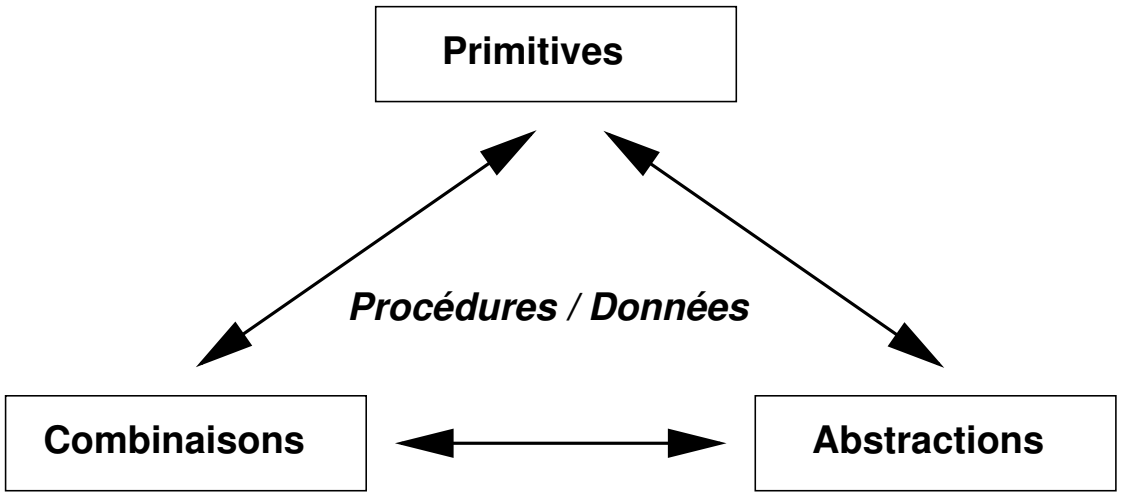
Haskell

```
invert :: [ a ] -> [ a ]
invert [] = []
invert (x:xs) = (invert xs) ++ [x]

-- invert [1..5]
-- invert ["a", "b", "c"]
```

Pourquoi l'approche fonctionnelle est bénéfique

Les 3 caractéristiques des (bons) langages



- Moins de distinction entre procédures et données
- Plus de puissance dans la combinaison
- Plus de puissance dans l'abstraction

Entre Lisp et Haskell

Deux approches fonctionnelles de la programmation

	Fonct.	**Évaluation**	**Typage**	**Autres**
Lisp	impur*	stricte*	dynamique*	. . . *
Haskell	pur	lazy	statique	. . .

* ou pas. . .