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. . .

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

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

Aussi simple que d'écrire `i = j` :

Haskell

```
backwards :: [a] -> [a]
backwards = reverse
```

Scheme

```
(define backwards reverse)
```

Lisp

```
(setf (symbol-function 'backwards) #'reverse)
```

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]
```

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
```

Retours fonctionnels

Avec fonctions anonymes

Aussi simple que `return 42;` :

Lisp

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

Haskell

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

Haskell

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

Lisp

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

Haskell

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

Évaluation stricte / lazy

Pour ou contre ?

La paresse : une vertu ?

- **Intérêt :** plus d'abstraction (ex. manipulation de listes infinies).

Lisp

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

;; KO
```

Haskell

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

-- OK
```

- **Contrainte :** pureté fonctionnelle requise (on ne peut pas s'appuyer sur l'ordre d'évaluation).

Pseudo-paresse dans les langages impératifs

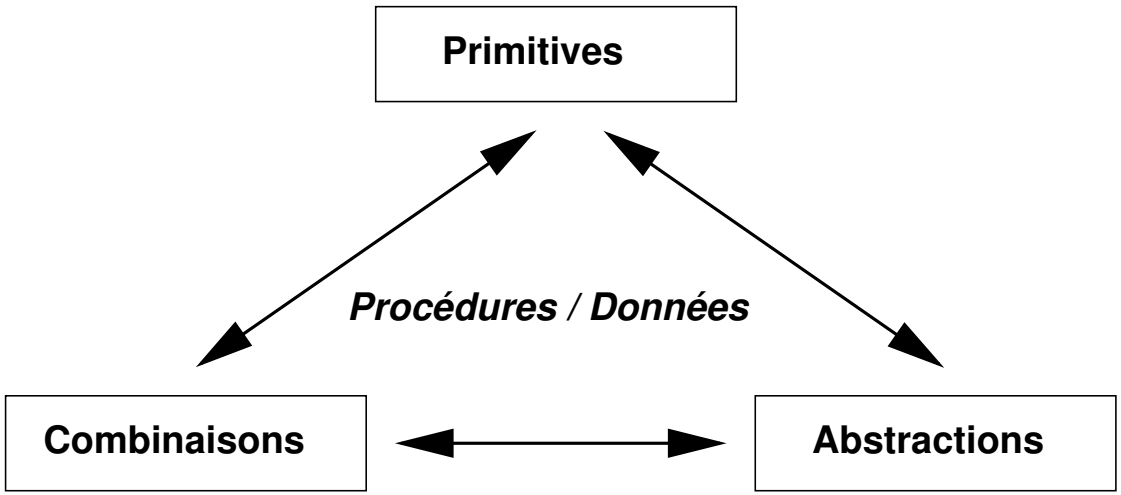
On ne fait toujours que ce qu'on peut...

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

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

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. . .