Approches Fonctionnelles de la Programmation

Évaluation et Scoping

Didier Verna

EPITA / LRDE

didier@lrde.epita.fr

lrde/~didier


@didierverna


didier.verna


in/didierverna

Plan

Techniques d'Évaluation

- Évaluation Stricte / Ordre Applicatif
- Évaluation Paresseuse / Ordre Normal
- Mérites Comparés

Formes de Scoping

- Structure de Blocs / Localité
- Scoping Lexical *vs.* Dynamique

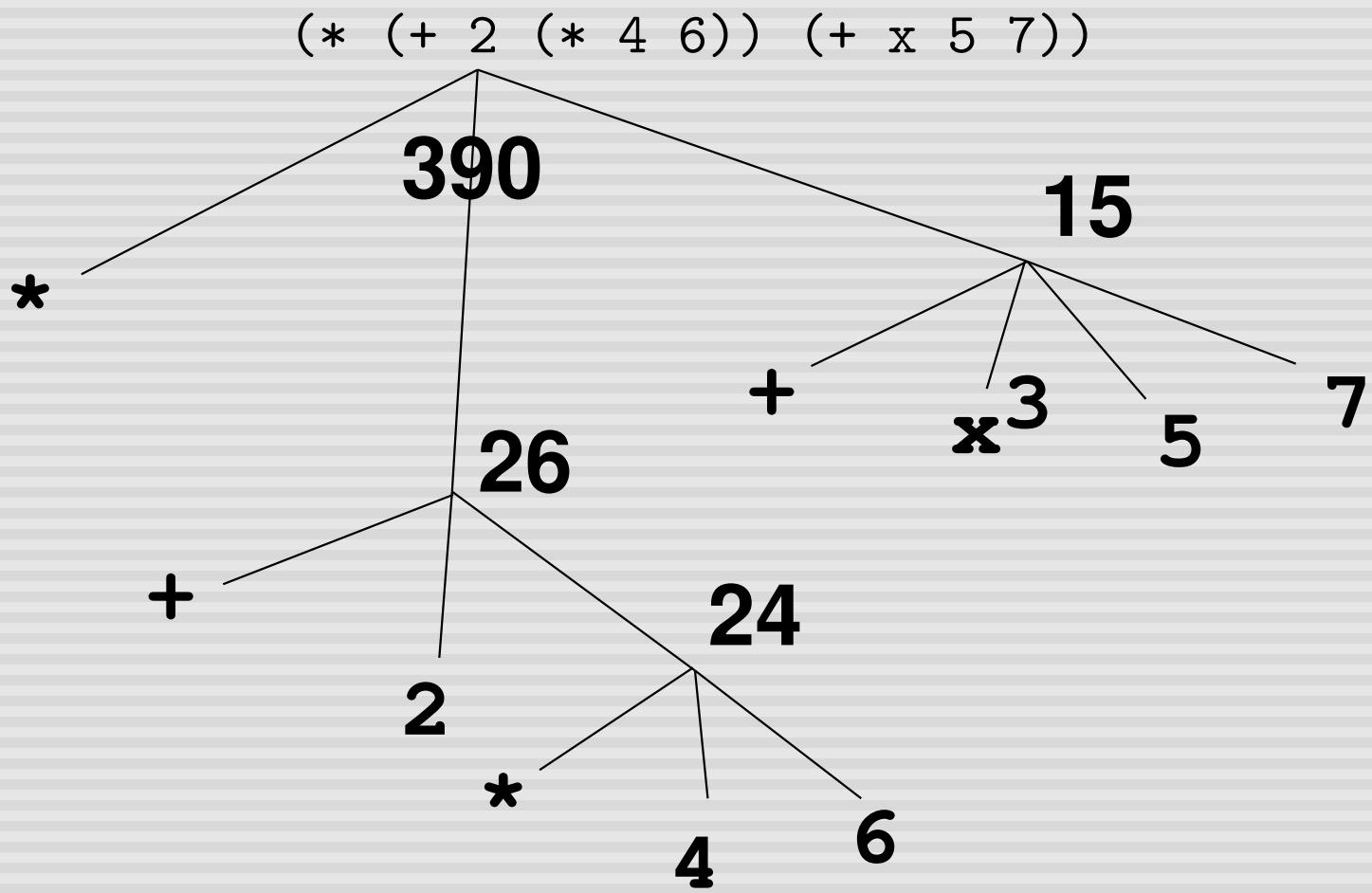
Plan

Évaluation Stricte (Lisp)

- *A.k.a.* « ordre applicatif »
- **Processus**
  1. Évaluer les sous-expressions de gauche à droite
  2. Appliquer l'opérateur à ses arguments

  Processus récursif (« tree accumulation »)
- **Feuilles :** littéraux / opérateurs primitifs
- **Environnement**
  - Valeurs d'expressions abstraites
  - Cas particulier : opérateurs primitifs

Exemple



Modèle de Substitution

- **Pour les expressions abstraites**
  - Identique au cas précédent
  - *Substitution* (étape préalable)
    Remplacement des paramètres formels par les arguments correspondants
- **Remarques**
  - Modèle théorique
  - Substitution $\Rightarrow$ environement local
  - Complexe ! *Cf.* λ-calcul, α-conversion, collision de noms

Exemple

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

(defun ssq (x y) (+ (sq x) (sq y)))

(defun f (a)
  (ssq (+ a 1) (* a 2)))
```

```
(f 5)
(ssq (+ a 1) (* a 2))
(ssq (+ 5 1) (* 5 2))
(ssq 6 (* 5 2))
(ssq 6 10)
(+ (sq x) (sq y))
(+ (sq 6) (sq 10))
(+ (* x x) (sq 10))
(+ (* 6 6) (sq 10))
(+ 36 (sq 10))
(+ 36 (* x x))
(+ 36 (* 10 10))
(+ 36 100)
136
```

Opérateurs Spéciaux

- **Problème :** idiotismes non stricts
  Branchements conditionnels, logique booléenne, etc.
- **Solution :** opérateurs *primitifs* spéciaux
  Technique d'évaluation spécifique
- 25 en Lisp (`setq`, `if`, `quote`, `function`, *etc.*)
- Test : `(special-operator-p 'if)`

Évaluation Paresseuse (Haskell)

- *A.k.a.* « ordre normal »
- **Processus**
  - Modèle de substitution identique
  - Mais évaluation *à la demande*
- **Performance :** modèle inefficace

  Travail sur un graphe d'évaluation (« mémoization »)
- **Contrainte :** fonctionnel pur uniquement

Exemple

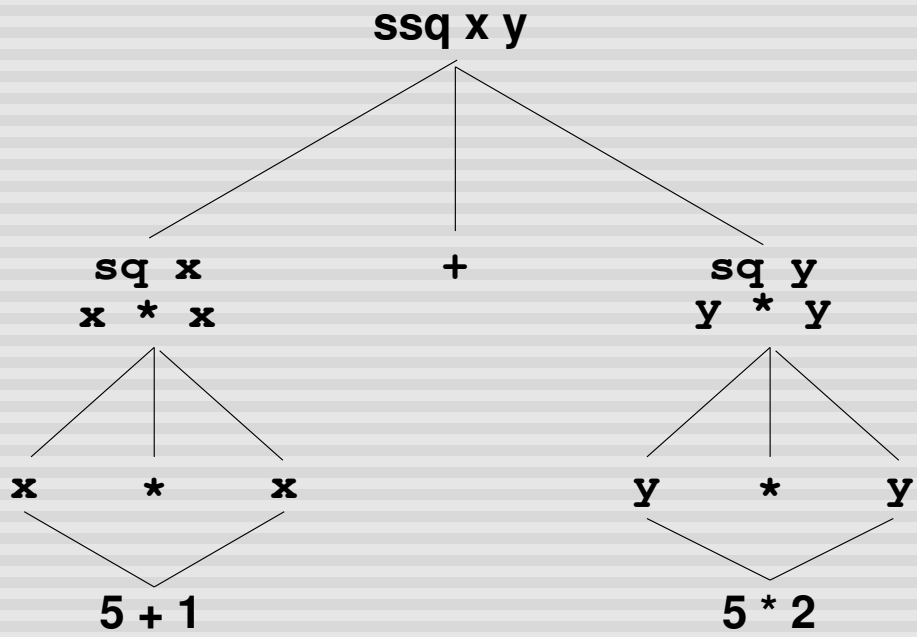
```haskell
sq :: Float -> Float
sq x = x * x

ssq :: Float -> Float -> Float
ssq x y = sq x + sq y

f :: Float -> Float
f a = ssq (a + 1) (a * 2)
```

```
f 5
ssq (a + 1) (a * 2)
ssq (5 + 1) (5 * 2)
sq x + sq y
sq (5 + 1) + sq (5 * 2)
(x * x) + (y * y)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
6 * (5 + 1) + (5 * 2) * (5 * 2)
6 * 6 + (5 * 2) * (5 * 2)
36 + (5 * 2) * (5 * 2)
36 + 10 * (5 * 2)
36 + 10 * 10
36 + 100
136
```

Graphe d'Évaluation / Mémoization




```
ssq (5 + 1) (5 * 2)
sq (5 + 1) + sq (5 * 2)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
6 * 6 + 10 * 10
```

Évaluation d'Équations

```
f p1 p2 p3 ... = e1
f q1 q2 q3 ... = e2
...
```

- **Équation utilisée :** premier « match »
- **Matching**
  - Seuls les arguments nécessaires à la prise décision
  - Seuls les *morceaux* d'arguments nécessaires à la prise décision

Exemples

Évaluation minimale des arguments

```
foo :: Float
foo = 2 * foo

prod :: Float -> Float -> Float
prod x 0 = 0
prod 0 y = 0
prod x y = x * y
```

```
prod 3 4 => (3) 12.0
prod 4 0 => (1)  0.0
prod foo 0 => (1) 0.0
prod 0 foo
=> Stack overflow
```

Évaluation partielle des arguments

```
sh :: [Int] -> [Int] -> Int
sh [] ys = 0
sh (x:xs) [] = 0
sh (x:xs) (y:ys) = x + y
```

```
sh [1..3] [5..8]
(1) => sh (1:[2..3]) [5..8]
(2) => sh (1:[2..3]) (5:[6..8])
(3) => 1 + 5
    => 6
```

Évaluation des Guardes

```haskell
max3 :: Int -> Int -> Int -> Int
max3 m n p
    | (m >= n) && (m >= p) = m
    | (n >= m) && (n >= p) = n
    | otherwise = p
```

```
max3 (2+3) (4-1) (3+9)
(1) => (2+3)>=(4-1) && (2+3)>=(3+9)
    =>     5>=(4-1) &&     5>=(3+9)
    =>     5>=3     &&     5>=(3+9)
    =>     True     &&     5>=(3+9)
    =>     True     &&     5>=12
    =>     True     &&     False
    => False
(2) => 3>=5  && 3>=12
    => False && 3>=12
    => False
(3) => 12
```

nrsqrt IV, Apocalypse

- **Rappel**

Lisp

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

Haskell

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

- **Nouvelle vision :** travailler directement sur la suite (infinie) de Newton-Raphson

Haskell

```
limit :: [ Float ] -> (Float -> Bool) -> Float
limit (y:ys) found
  | found y = y
  | otherwise = limit ys found

nrsqrt :: Float -> Float -> Float
nrsqrt x delta = limit (nrlist x 1) (nrfound x delta) -- partial application!

nrlist :: Float -> Float -> [ Float ]
nrlist x yn = yn : nrlist x ((yn + x / yn) / 2)

nrfound :: Float -> Float -> Float -> Bool
nrfound x delta yn = abs (x - yn * yn) <= delta
```

Évaluation Stricte *vs.* Paresseuse

- **L'évaluation stricte est plus efficace**
  Pas de redondance de calcul, mais *cf.* ordre normal + mémoization
- **L'évaluation paresseuse est plus expressive**
  P.ex. structures de données infinies, mais *cf.* macros de Lisp

Haskell

```
ifnot :: Bool -> a -> a -> a
ifnot test e1 e2 = if test then e2 else e1
```

Lisp

```
(defmacro ifnot (test e1 e2)
  (list 'if test e2 e1))
```

- **Pas de paresse sans pureté, pas d'impureté sans déterminisme**
  Mais l'évaluation stricte n'est pas la seule forme déterministe
- **Church-Rosser** (« confluence globale », *cf.* λ-calcul)
  Unicité de la forme canonique, quelle que soit la méthode de réduction (fonctionnel pur uniquement)

Évaluation Stricte *vs.* Paresseuse

- Quizz sémantique des différences

Lisp

```
(setq bar (sqrt (- (* 3 (+ 6 7)) 8)))
```

```
'(foo bar baz) ≠ (list foo bar baz)
```

```
(let* ((a (* x x)) ;; order important
       (b (+ a 1)))
  (+ (/ a b) (/ b a)))
```

Haskell

```
bar = sqrt (3 * (6 + 7) - 8)
```

```
[foo, bar, baz]
```

```
let b = a + 1 -- order not important
    a = x * x
in a / b + b / a
```

Pseudo-Paresse en Impératif

- **Branchements conditionnels, *etc.***

C

```c
if (something_that_turns_out_to_be_true)
{
  /* Computed */
}
else
{
  /* Not computed */
}
```

Plan

Structure de Blocs / Localité

- **Bloc :** ensemble de liaisons (« bidings ») $\{nom \rightarrow expression\}$
- **Environnement d'évaluation :** blocs imbriqués
- **Blocs explicites *vs.* implicites**
  - `let`, `where`, `flet`, *etc.*
  - Déclarations globales, paramètres d'appels de fonctions
- **Problèmes théoriques** (*cf.* α-conversion)

Lisp

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

```
(defun sq (x) (* x x))
(defun f  (x) (sq (* 2 x)))
```

Haskell

```
sq x = x * x
sq y = y * y
```

```
sq x = x * x
f  x =  sq (2 * x)
```

Notion de Scoping

- **Contexte**
  - Variable *liée* (« bound ») : définie dans le bloc local
  - Variable *libre* : non définie localement
- **Scoping**
  - Recherche d'une liaison dans le bloc le plus « proche »
  - La notion de « proximité » reste à définir...

Scoping et Collision

- Les liaisons locales sont prioritaires

Lisp

```
(defvar x 5)

(defvar y              ;; y = 8
  (+ (let ((x 3)) x)   ;; x = 3
     x))               ;; x = 5
```

Haskell

```
x :: Int
x = 5

y :: Int                -- y = 8
y = (let x = 3 in x)    -- x = 3
    + x                 -- x = 5
```

Scoping et Collision

- **Mais attention à la sémantique des langages !**
  - Lisp : valeurs locales calculées à l'*extérieur* de `let`

Lisp

```
(defvar x 2)

(defvar y
  (let ((x 3)          ;; x = 3
        (z (+ x 2)))   ;; z = 2 + 2
    (* x z)))          ;; y = 3 * 4
```

  - Aucun sens en Haskell (plus proche du `let*` de Lisp)

Lisp

```
(defvar x 2)

(defvar y
  (let* ((x 3)          ;; x = 3
         (z (+ x 2)))   ;; z = 3 + 2
    (* x z)))           ;; y = 3 * 5
```

Haskell

```
x :: Int
x = 2

y :: Int
y = let x = 3      -- x = 3
        z = x + 2  -- z = 3 + 2
        in x * z   -- y = 3 * 5
```

Formes de Scoping

- **Rappel :** recherche de la liaison la plus « proche »
  Ne concerne que les variables libres
- **Lexical (statique) :** recherche dans l'environnement de *définition*
- **Dynamique :** recherche dans l'environnement d'*appel*
- **Remarque :** `let` fait deux choses différentes à la fois !

Lisp

```
(let ((x 10)).              ;; lexical scope
  (defun foo () x))

(let ((x 20))
  (foo))                    ;; => 10
```

Lisp

```
(defvar x 10)               ;; dynamic scope
(defun foo () x)

(let ((x 20))
  (foo))                    ;; => 20
```

Notion de Fermeture Lexicale

- *A.k.a.* « lexical closures »
- **Définition**

  Combinaison d'une fonction avec son environnement de définition

  Valeurs des variables libres au moment de la définition
- **Intérêt**
  - Sous-tend *tout* le fonctionnement de l'ordre supérieur !
  - *Cf.* λ-calcul / chapitre 2

Exemples

- **Arguments fonctionnels**

Lisp

```
(defun list+ (lst n)
  (mapcar (lambda (x) (+ x n)) lst))
```

Haskell

```
(+++) :: [Int] -> Int -> [Int]
(+++) lst n = map (\x -> x + n) lst
```

- **Retours fonctionnels**

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

```
makeAdder :: Int -> Int -> Int
makeAdder n = \x -> x + n
```

- **Localité + mutation**

```
(let ((cnt 0))
  (defun get-uid () (incf cnt))
  (defun reset-uids () (setq cnt 0)))
```

Scoping Dynamique (Lisp)

- Forme historique en Lisp
- Depuis `Scheme` : scoping lexical
- Common Lisp : scoping lexical par défaut, dynamique possible
  - Variables globales (`defvar`, *etc.*)
  - Variables locales déclarées « spéciales »

```
(let ((x 10)).              ;; lexical scope
  (defun foo () x))

(let ((x 20))
  (foo))                    ;; => 10
```

```
(defvar x 10)               ;; dynamic scope
(defun foo () x)

(let ((x 20))
  (foo))                    ;; => 20
```

```
(let ((x 10))
  (defun foo ()
    (declare (special x)) ;; dynamic scope
    x))

(let ((x 20))
  (foo))                    ;; => 20
```

Pour ou Contre le Scoping Dynamique ?

- **Intérêt**
  - Certains paradigmes (Programmation Orientée-Contexte, *etc.*)
  - Gestion des exceptions (handlers)
  - Variables globales (*p.ex.* options utilisateurs, *cf.* Emacs)

```
(defvar *default-background* 'blue) ;; note the earmuffs!

(defun create-42-red-windows ()
  (let ((*default-background* 'red))
    (loop :repeat 42 :do (create-window))))
```

- **Inconvénient**
  - Inadapté à l'ordre supérieur
  - Source de bugs très difficiles à pister
    Problème de la collision de noms (« name clash »)
  - Le tout premier exemple de fonction d'ordre supérieur donné par Mc Carthy était faux !

Le (Mauvais) Exemple de McCarthy

- Première fonction de mapping de l'histoire

```
(defmacro while (test &rest body)
  `(do () ((not ,test))
     ,@body))

(defun my-mapcar (func lst)
  (let (elt n)
    (while (setq elt (pop lst))
      (push (funcall func elt) n)) ;; <- name clash on n!!
    (nreverse n)))

(defun list+ (lst n)
  (my-mapcar (lambda (x)
               (declare (special n)) ;; Do that and...
               (+ x n)) ;; ... barf, name clash on n!!
             lst))
```