Programmation Fonctionnelle
Évaluation paresseuse

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

13 octobre 2014

Plan

Plan

Ordre d'évaluation

Considérez la fonction suivante :

```
inc :: Int -> Int
inc n = n +1
```

Pour evaluer l'expression

```
inc (2*3)
```

nous avons deux possibilités :

```
inc (2 * 3)
 -> inc 6
 -> 6 + 1
 -> 7
```

```
inc (2 * 3)
 -> (2 * 3) + 1
 -> 6 + 1
 -> 7
```

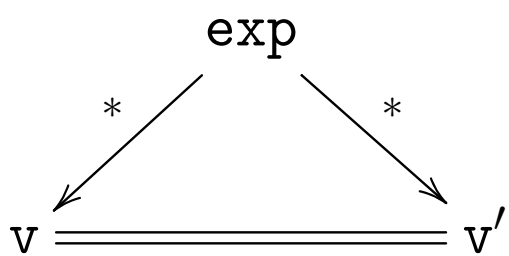
La confluence

On démontre le fait suivant :

en Haskell,
si deux évaluations de la même expression exp
produisent les valeurs v *et* v'*,*
alors $\mathrm{v} = \mathrm{v}'$*.*

On dit que l'évaluation est confluente.

En diagrammes :



La même propriété n'est pas nécessairement vraie si on considère des expressions qui entraînent des effets de bord.

Par exemple, dans un langage où l'on peut affecter la variable *n*, l'expression

```
n + (n:=1)
```

peut s'évaluer à deux résultats différents,
selon que l'on évalue d'abord `n` ou son affectation.

Plan

Redexes

Un *redex* est une sous-expression d'une expression donnée qui peut s'évaluer (donc réduire) à quelque chose d'autre.

Considérez la fonction `mult`

```
mult :: (Int,Int) => Int
mult (x,y) = x * y
```

et l'expression

```
mult (1+2,2+3)
```

Cette expression possede 3 redexes :

```
mult (1+2,2+3) -> 1+2 * 2+3
mult (1+2,2+3) -> mult (3,2+3)
mult (1+2,2+3) -> mult (1+2,5)
```

Évaluation par l'intérieur (innermost evaluation)

1. On évalue un redex qui ne contient pas d'autres redexes.
2. Si plusieurs redexes ont cette même propriété,
   on réduit alors le redex le plus à gauche.

Par exemple :

```
mult (1+2,2+3)
  -> mult (3,2+3)
  -> mult (3,5)
  -> 3 * 5
  -> 15
```

Remarque :

- Cette stratégie d'évaluation assure que les arguments des fonctions sont évalués avant la fonction même, c'est-à-dire ils sont *passés par valeur*.

Evaluation par l'exterieur (outermost evaluation)

1. On évalue un redex qui n'est pas contenu dans d'autres redexes.
2. Si plusieurs redexes ont cette même propriété, on réduit alors le redex le plus a gauche.

Donc :

```
mult (1+2,2+3)
  -> 1+2 * 2+3
  -> 3 * 2+3
  -> 3 * 5
  -> 15
```

β-réduction

Par l'exemple :

```
(\x -> \y -> x + y) 5 6
  ->  (\y -> 5 + y) 6
  -> 5 + 6
  -> 11
```

Règle générale :

$$(\lambda x -> e)arg \rightarrow e[x -> arg]$$

On évalue l'expression

fonction $\lambda x -> e$ appliquée à l' argument arg

en substituant la variable x par arg dans e (le corp de la fonction)

Les λ-expressions sont des valeurs

- En Haskell (et dans tous les autres langages fonctionnels qui se fondent sur le λ-calcul), on ne réduit pas sous une λ-expression.
- Les λ-expressions sont donc traitées comme des valeurs.
- Rationale : les fonctions sont des boites noires qu'on ne peut pas simplifier.

Ainsi

```
(\x -> 1 + 2) 0
  -> 1 + 2
  -> 3
```

est à la fois une évaluation par l'extérieur et par l'*intérieur*.

Call-by-value et call-by-name

- stratégie « *call-by-value* » (appel par valeur) :
  évaluation par l'intérieur,
  + λ-expressions pas évaluées,

- stratégie « *call-by-name* » (appel par nom) :
  évaluation par l'extérieur,
  + λ-expressions pas évaluées.

Plan

Terminaison

Expérimentons avec ce programme :

```
omega :: Int
omega = omega +1

main1 = omega
main2 = fst (0,omega)
```

Haskell utilise la stratégie call-by-name,
le `main2` ne pose pas des problèmes :

```
Prelude> :load "omega.hs"
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main1
  C-c C-cInterrupted.
*Main> main2
0
*Main>
```

En fait, en réduisant par l'extérieur, on a :

```
main2
   -> fst (0,omega)
   -> 0
```

On ne demande pas d'évaluer `omega`.

Nombre de réductions et complexité

Considérez la fonction suivante :

```
square :: Int -> Int
square n = n * n
```

Evaluation de l'interieur :

```
square (1+2)
  -> square 3
  -> 3*3
  -> 9
```

Evaluation de l'exterieur :

```
square (1+2)
  -> (1+2)*(1+2)
  -> 3*(1+2)
  -> 3 * 3
  -> 9
```

Le sur-coût dû à la duplication des expressions est résolu, en Haskell, via le partage des sous-expressions.

Par exemple :

```
square (1+2)
  -> x*x          ou x = 1+2
```

on évalue x, en un seul coup

```
  -> 3 * 3
  -> 9
```

Évaluation paresseuse

En anglais, « *lazy evaluation* » :

call by name + partage	=	évaluation de l'extérieur + λ-expressions sont valeurs + partage

Remarque :

- On suggère de traduire « *lazy* » par "retardée", au lieu que par "paresseuse".

Structures infinies

En Haskell, on manipule des expressions.

Le programme suivant ne pose donc pas de problèmes :

```
-- La liste infinie,
-- qui contient seulement des 1
ones :: [Int]
ones = 1:ones

two_ones = take 2 ones
```

En fait, on a :

```
two_ones
  -> take 2 ones
  -> take 2 (1:ones)
  -> 1:take 1 ones
  -> 1:take 1 (1:ones)
  -> 1:1:take 0 ones
  -> 1:1:[] = [1,1]
```

Le crible d'Eratosthène

- l_0 : la liste de tous les entiers ≥ 2,
- l_{n+1} est obtenu de l_n
  en lui enlevant la tête de l_n et ses multiples,
- les nombres premiers : les têtes des l_n.

2	3	4	5	6	7	8	9	...
	3		5		7		9	...
			5		7			...

Nous pouvons programmer le crible comme suit :

```
premiers :: [Int]
premiers = crible [2..]

crible :: [Int] -> [Int]
crible (p:xs) =
    p:crible [x | x <- xs, x `mod` p /= 0 ]

premiers_400_premiers = take 400 premiers
```

Plan

Programmation modulaire

L'exemple précèdent montre aussi comme l'évaluation paresseuse peut venir en aide à la modularité des programmes.

En fait, le problème d'engendrer tous le nombres premiers :

```
premiers = crible [2..]
```

est scindé du problème d'un prendre un nombre fini :

```
take 5 premiers
```

Un autre exemple I

Comparez le programme suivant :

```
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
  in_handle <- openFile "input.txt" ReadMode
  out_handle <- openFile "output.txt" WriteMode
  mainloop in_handle out_handle
  hClose in_handle
  hClose out_handle
```

Un autre exemple II

```
mainloop :: Handle -> Handle -> IO ()
mainloop inh outh = do
  ineof <- hIsEOF inh
  if ineof then
      return ()
  else
      do
        inpStr <- hGetLine inh
        hPutStrLn outh (processData inpStr)
        mainloop inh outh

processData :: String -> String
processData inpStr = map toUpper inpStr
```

Un autre exemple III

à cet autre :

```
import Data.Char(toUpper)

main :: IO ()
main = do
  inpStr <- readFile "input.txt"
  writeFile "output.txt" (processData inpStr)

processData :: String -> String
processData inpStr = map toUpper inpStr
```

Plan

foldr et *foldl*

Rappelons d'abord les définitions de ces fonctions :

```
foldr :: (a -> b -> b) ->  b ->  [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Des tests : `foldr`

```
foldr (&&) True (map odd [1..30])
  = foldr (&&) True (map odd 1:[2..30])
  = foldr (&&) True (odd 1:map odd [2..30])
  = odd 1 && (foldr  (&&) True (map odd [2..30]))
  = True && (foldr  (&&) True (map odd [2..30]))
  = foldr  (&&) True (map odd [2..30])
  = ...
  = False &&  (foldr (&&) True (map odd [3..30]))
  = False
```

L'évaluation paresseuse de l'expression

`foldr (&&) True (map odd [1..30])`

ne produit pas une liste de valeurs Booléens alternés, de longueur `30`.

...avec `foldl`

```
foldl (&&) True (map odd [1..30])
  = foldl (&&) True (map odd 1:[2..30])
  = foldl (&&) True ((odd 1):map odd [2..30])
  = foldl (&&) (True && odd 1) (map odd [2..30])
  = ...
  = foldl (&&) ((True && odd 1) && odd 2) (map odd [3..30])
  = ...
  = foldl (&&) (...((True && odd 1) && odd 2)... odd 30) []
  = (...((True && odd 1) && odd 2) = ... = False
```

Avec `foldl` et l'évaluation paresseuse, on produit une expression intermédiaire importante à évaluer :

(...((True && odd 1) && odd 2)... odd 30)

Avec l'interprète

```
Prelude> foldl (&&) True (map odd [1..1000000000])
^C^CInterrupted.

Prelude> foldr (&&) True (map odd [1..1000000000])
False

Prelude>
```

Conclusions

- `foldr` est en général meilleure que `foldl` ;
- `foldl` est en général meilleure que `foldr`, si l'évaluation est stricte (appel par nom) ;
- si la fonction binaire n'est pas associative, on peut être obligés d'utiliser `foldl`, et non pas `foldr`.
  Par exemple :

```
implique :: Bool -> Bool -> Bool
impique False  _ = True
implique _ True = True
implique True False = False
```

```
*Main> foldl implique True (map odd [1..100])
False
*Main> foldr implique True (map odd [1..100])
True
*Main>
```