Programmation Fonctionnelle
Définitions (et construction) des fonctions

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

15 septembre 2014

Plan

Conditionnels, conditions de garde, filtrage

Expressions Lambda

Sections

Plan

Expressions conditionnelles

Comme dans tous les langages, on peut définir des fonctions en utilisant des expressions conditionnelles.

Par exemple :

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

> *`abs` prend un entier n et retourne n si n n'est pas négatif, sinon il retourne $-n$.*

Les expressions conditionnelles peuvent être imbriquées :

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Remarque :

- En Haskell, les expressions conditionnelles possèdent toujours le branche `else`.
  Cela évite l'ambiguïté des conditionnels imbriqués.

Inférence de type : `if .. then .. else ..`

La règle est la suivante :

$$\frac{\texttt{x :: Bool} \qquad \texttt{y :: t} \qquad \texttt{z :: t}}{\texttt{if x then y else z :: t}}$$

Par exemple :

```
> if True then 1 else [1]
... Erreur de type :-(
```

Equations avec conditions de garde

Alternativement au conditionnel, les fonctions peuvent être définies avec des équations avec des conditions de garde (*guarded equations*).

```
abs :: Int -> Int
abs n
    | n >= 0 = n
    | otherwise = -n
```

Comme auparavant,
en utilisant les conditions de garde.

Les équations avec conditions de garde rendent les définitions avec conditions multiples plus lisibles :

```
signump :: Int -> Int
signump n
    | n < 0 = -1
    | n == 0 = 0
    | otherwise = 1
```

Remarque(s) :

- La condition `otherwise` – qui filtre toutes les conditions – est définie dans `prelude.hs` par

```
otherwise = True
```

Filtrage par motifs (pattern matching)

Plusieurs fonctions possèdent une définition assez claire en utilisant le filtrage sur leur arguments.

```
not :: Bool -> Bool
not False = True
not True = False
```

> *`not` envoie `False` vers `True`, et `True` vers `False`.*

En utilisant le filtrage, la même fonction peut se définir de plusieurs façons.

Par exemple :

```
(&&) :: Bool -> Bool -> Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

peut se définir aussi par

```
True && True = True
_ && _ = False
```

La définition

```
True && b = b
False && _ = False
```

est plus efficace.
Elle permet de ne pas évaluer la deuxième expression si la première s'évalue à `False`.

Remarque(s) :

- Le symbole _ (underscore) est un motif qui filtre tout valeur, sans l'assigner à aucune variable.

- Les motifs sont filtrés dans l'ordre. Par exemple, cette définition retourne toujours `False` :

```
_ && _ = False
True && True = True
```

- Les motifs n'ont pas de variables repetés.
  Par exemple, la définition suivante donne un erreur :

```
b && b = b
_ && _ = False
```

Motifs et listes

Internement, toute liste non vide est construite par l'utilisation de l'opérateur `(:)` appelé "cons" qui ajoute un élément au début de la liste.

```
[1,2,3,4]
```

signifie

```
1:(2:(3:(4:[])))
```

Les fonctions sur les listes peuvent être définies en utilisant les motifs de la forme `x:xs`.

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

> *`head` (resp. `tail`) envoie toute liste non-vide vers son premier élément (resp. vers la listes des éléments restants, la queue).*

Remarque(s) :

- les motifs `x:xs` filtrent seulement les listes non vides :

```
> head []
Error
```

- les motifs `x:xs` doivent être parenthésés, car l'application est prioritaire sur `(:)`. Par exemple, la définition suivante produit un erreur :

```
head x:_ = x
```

Motifs (II)

Une définition par motif est utile pour accéder à un morceaux d'information structurée :

```
third :: (a,b,c) -> c
third (x,y,z) = z

produit :: (Float,Float) ->
           (Float,Float) -> (Float,Float)
produit (xre,xim) (yre,yim) =
    (xre*yre - xim*yim, xre*yim +xim*yre)
```

Motifs (III)

- Un motif est un *valeur*
  (c'est-à-dire, expression non évaluable ultérieurement),
  pouvant contenir des variables (non répétés).
- Lors de l'appel d'une fonction
  - on *unifie* l'argument de la fonction avec le motif ;
  - les variables du motif sont instanciés à des morceaux de l'argument.

```
reverse (x:xs) = reverse xs ++ [x]

reverse [1,2,3] -->
    (x := 1, xs := [2,3])
    reverse [2,3] ++ [1]
```

Plan

Expressions Lambda (ou abstraction)

On peut construire des fonctions sans les nommer explicitement.

A ce fin, on utilise la notation λ (lambda).

```
λx -> x + x
```

> *la fonction (sans nom) qui prend un entier* `x` *et renvoie* `x + x` *comme résultat.*

Remarque(s) :

- Le symbole λ est la lettre grecque lambda ;
  elle est tapée au clavier comme backslash, \.
- En mathématique, les fonctions sans noms sont d'habitude dénotées par le symbole $\mapsto$, comme dans

$$x \mapsto x + x \quad .$$

- En Haskell, l'utilisation du symbole lambda pour les fonctions sans nom vient du lambda-calcul (le fondement théorique du langage Haskell).

Pourquoi utiliser la notation lambda ?

Les λ-expressions donnent une signification formelle (c'est-à-dire précise) aux fonctions définies en utilisant la Curryfication.

Par exemple :

```
add x y = x+y
```

signifie

```
add = \x -> (\y -> x+y)
```

Pourquoi utiliser la notation lambda (II) ?

Les λ-expressions sont aussi utiles quand on définie des fonctions qui retournent des fonctions comme résultat.

Par exemple :

```
const :: a -> b -> a
const x _ = x
```

est plus naturellement définie par :

```
const :: a -> (b -> a)
const x = \_ -> x
```

Pourquoi utiliser la notation lambda (III) ?

Les expressions lambda sont utiles pour nommer des fonctions qu'on référence une fois seulement.

Par exemple :

```
odds n = map f [0..n-1]
   where f x = x*2 + 1
```

peut être simplifiée à

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

L'application

Rappel

- Un autre opérateur (associatif à gauche) fondamental qui origine des fonctions est

  l'application d'une fonction à un argument

  dénoté par la juxtaposition :

```
(\x -> \y -> x + y) 1 2
```

- Remplacer les espaces pertinents par des traits permet de localiser les occurrences de cet opérateur.

  On obtient (après parenthèsage) :

```
((\x -> \y -> x + y)_1)_2
```

Règles de typage

Abstraction :

$$\frac{e :: t_2 \qquad x :: t_1}{\lambda x \to e :: t_1 \to t_2}$$

Application :

$$\frac{e :: t_1 \to t_2 \qquad e' :: t_1}{e\ e' :: t_2}$$

Si l'expression e a le type des fonctions $t_1 \to t_2$, et e' a le type des arguments t_2, alors l'expression e e' a le type des valeurs retournés par ces fonctions, c'est-à-dire t_2.

Autrement :
Condition nécessaire afin que e $e' :: t_2$ est qu'il existe un type t_1 tel que $e' :: t_1$ et $e :: t_1 \to t_2$.

Règles de typage

Abstraction :

$$\frac{e :: t_2 \qquad x :: t_1}{\lambda x \to e :: t_1 \to t_2}$$

Application :

$$\frac{e :: t_1 \to t_2 \qquad e' :: t_1}{e\ e' :: t_2}$$

Si l'expression e a le type des fonctions $t_1 \to t_2$, et e' a le type des arguments t_2, alors l'expression e e' a le type des valeurs retournés par ces fonctions, c'est-à-dire t_2.

Autrement :
Condition nécessaire afin que e $e' :: t_2$ est qu'il existe un type t_1 tel que $e' :: t_1$ et $e :: t_1 \to t_2$.

Attention :
si les expressions de type contiennent des variables,
ces règles s'interprètent modulo unification.

Exemple :

$$\texttt{map} :: (a \to b) \to [\texttt{a}] \to [\texttt{b}]$$

$$\texttt{succ} :: \mathit{Int} \to \mathit{Int}$$

$$\texttt{map succ} :: [\texttt{Int}] \to [\texttt{Int}]$$

> *Les types* `a` *et* `b` *sont instanciés par* `Int`.

Règle générale pour l'application :

$$\frac{e : t_1 \to t_2 \qquad e' : t_3}{e\ e' : \sigma(t_2)}$$

où σ est un MGU de (t_1, t_3).

Plan

Sections

Un operateur infixe – écrit entre ses arguments – peut être converti à une fonction Curryfiée, écrite avant ses arguments.

Pour cela on utilise le mettre entre parenthèses.

Par exemple :

```
> 1+2
3
> (+) 1 2
3
```

Cette convention permet aussi d'avoir un argument entre parentheses. Par exemple :

```
> (1+) 2
3
> (+2) 1
3
```

En general, si `@` est un operateur, alors les fonctions de la forme `(@)`, `(x@ )` et `(@y)` sont appellées sections.

Utiliser les sections ?

Plusieurs fonctions se définissent en utilisant les sections.

Par exemple :

- `(1+)` fonction successeur
- `(1/)` réciproque
- `(*2)` double
- `(/2)` moitié

Des fonctions aux operateurs

Une fonction de deux arguments peut être utilisée en tant que opérateur binaire si on le me entre guillaumets arrière.

```
div :: Int -> Int -> Int

deux = 4 `div` 2
```

Exercices I

1. Considerez la fonction `f` suivante :

```
f [] x d = d x
f ((y,z):ws) x d
  | (y x) =  z x
  | otherwise = f ws x d
```

   Quel est le type de `f` ?

2. Proposez une règle de typage pour les définitions par équations avec condition de garde.

Exercices II

3. Considérez la fonction `safetail` qui se comporte exactement comme `tail`, sauf qu'elle envoie la liste vide vers elle même. Rappel : dans ce cas, `tail` produit un erreur.
   Définissez `safetail` en utilisant :
   (a) une expression conditionnelle ;
   (b) des équations avec conditions de garde ;
   (c) le filtrage par motifs.

   Aide : la fonction du prelude

   ```
   null :: [a] -> Bool
   ```

   teste si une liste est vide.

Exercices III

4. Donnez trois possibles définitions de l'opérateur logique `(||)`, en utilisant le filtrage par motifs.
5. Redéfinissez la version suivante de `(&&)` en utilisant les conditionnels au lieu du filtrage :

```
True && True = True
_ && _ = False
```

6. Faites de même pour la version suivante :

```
True && b = b
False && _ = False
```