Lambda Calcul et Programmation Fonctionnelle (LCPF)

Zied Bouraoui

CRIL-CNRS & Univ Artois

bouraoui@cril.univ-artois.fr

http://www.cril.univ-artois.fr/~bouraoui/

Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

La notion de type

Types

Les types aident à vérifier la correction des programmes et à les traduire en code exécutable

Les types ont une importance accrue dans les langages fonctionnels car ils servent de fondement à certains langages, notamment ML et ses descendants (dont Haskell).

En programmation fonctionnelle, un objet appartenant à un type est une constante entièrement déterminée par sa valeur

Langages faiblement et fortement typés

Un langage est **fortement typé** quand un seul type est attribué à toute expression bien formée du langage.

Pour la plupart de ces langages, le type est déterminé et vérifié avant l'exécution du programme (statiquement).

Un langage est **faiblement typé** quand une même expression peut avoir plusieurs types. Pour la plupart de ces langages, le type est déterminé et vérifié au moment de l'exécution du programme (dynamiquement).

Haskell est fortement typé

Haskel est fortement typé. Donc, toute expression a un type. Nous pouvons demander le type d'une expression avec la commande
`:type`

```
Prelude > :type 1
1 :: Num t => t
Prelude > :type 1 + 1.0
1 + 1.0 :: Fractional a => a
```

Nous pouvons aussi utiliser la commende :set +t pour demander que les types des expressions soient toujours affichés :

```
Prelude> :set +t
Prelude> circonf r = 2 * 3.14 * r
circonf :: Fractional a => a -> a
Prelude > circonf 1
6.28
it :: Fractional a => a
```

Langages faiblement et fortement typés

L'approche faiblement typé est plus souple, mais avec cet approche le programme peut mal fonctionner (sans forcement s'interrompre) en raison d'une erreur de type qui n'a pas été détectée plus tôt

L'approche fortement typé est plus contraignant, mais de tels erreurs sont impossibles

Les langages fortement typés incluent donc les notions de constructeur de type ou inférence de type définissant une véritable théorie de types

Constructeurs de type

Types de base

Il y a plusieurs types de base en Haskell. Voici quelques uns :

type	description
`Char`	caractères unicode
`Bool`	valeurs logique
`Int`	entiers (taille fixe)
`Integer`	entiers (taille illimitée)
`Float`	virgules flottante (non recommandé)
`Double`	virgules flottante
`Rational`	rationnels (deux entiers)
`Fractional`	(polymorphe) `Double`, `Float` ou `Rational`
`Num`	(polymorphe) numérique

Listes

Une **liste** est un conteneur de valeurs dont tous les éléments sont du même type. Il y a plus d'une manière de créer de listes en Haskell :

```
Prelude> [1, 2, 3]
[1, 2, 3]
Prelude> 1:2:3:[]
1:2:3:[]
Prelude> []
[]
```

```
Prelude> :type []
[] :: [t]
Prelude > :type [1, 2, 3]
[1, 2, 3] :: Num t => [t]
Prelude> :type 1:2:3:[]
1:2:3:[] :: Num a => [a]
Prelude> :type "chaine"
“chaine" :: [Char]
```

Listes

Nous pouvons utiliser les fonctions *head* et *tail* respectivement pour avoir la tête et la queue de la liste

```
Prelude > head [1, 2, 3]
1
Prelude > tail [1, 2, 3]
[2,3]
Prelude > head (1:2:3:[])
1
Prelude > tail (1:2:3:[])
[2,3]
Prelude> head "toto"
't'
Prelude> tail "toto"
“oto"
Prelude >
```

Exercice

Écrivez la fonction longueur qui calcule la longueur d'une liste.

```
longueur l
| l == [] = 0
| otherwise = 1 + longueur(tail l)
```

Écrivez la fonction concat qui concatène deux listes.

```
concat l1 l2
| l1 == [] = l2
| otherwise = (head l1) : concat (tail l1) l2
```

Filtrage par motif

Le **filtrage par motif** (ou pattern matching) consiste à utiliser un motif à la place d'un paramètre formel pour designer une valeur d'un type structuré.

En Haskel :

```
case e of {
p1 -> e1;
p2 -> e2;
...
pn -> en
}
```

Filtrage par motif

Écrivez la fonction longueur qui calcule la longueur d'une liste.

```
longueur l =
    case l of {
      [] -> 0;
       _:q -> 1 + (longueur q)
}
```

Exercice : Écrivez la fonction concat en utilisant le filtrage par motif.

```
concat l1 l2 =
    case l1 of {
        [] -> l2
        t:q -> t : (concat q l2)
}
```

Constructeurs de type

Un constructeur de type est un opérateur qui prend en argument un ou plusieurs types, et a pour résultat un type.

Il y a **deux constructeurs de type fondamentaux** : **produit et flèche.**

Type produit

Soit A1, . . . An de types bien formés. La notation A1 × ··· ×An est un type bien formé. Les éléments de A1 ×···×An sont des n-uplets (a1,…,an) où chaque ai est du type Ai.

Exemples :

Le type N × N contient les éléments (0, 0), (0, 1), (45, 2029), etc. 1

Le type Z × R × Q contient (−1, π, 0), (−234, 1.34, 3), etc.

```
Prelude > :type (1, 2)
(1, 2) :: (Num t1, Num t) => (t, t1)
Prelude > :type ('a', 1)
('a', 1) :: Num t => (Char, t)
```

Type flèche

Soit A1 et A2 deux types bien formés. La notation A1 → A2 est un type bien formé.

Les éléments de A1 → A2 sont les fonctions de domaine A1 et codomaine A2.

Exemples :

Le type R → R contient les fonctions qu'associent un nombre réel à un nombre réel.

Le type (R × R) → R contient les fonctions qu'associent un nombre réel à tout couple de nombre réels.

```
Prelude> circonf r = 2 * 3.14 * r
circonf :: Fractional a => a -> a
Prelude> somme (a, b) = a + b
somme :: Num a => (a, a) -> a
```

Priorité et associativité

L'opérateur $\times$ est prioritaire sur $\rightarrow$.

$A \times B \rightarrow C = (A \times B) \rightarrow C$

L'opérateur $\rightarrow$ est associatif à droite.

$A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Attention :

l'opérateur $\times$ n'est pas associatif.

$$(A \times B) \times C \neq A \times B \times C$$

$$A \times (B \times C) \neq A \times B \times C$$

$$(A \times B) \times C \neq A \times (B \times C)$$

Fonctionnelles et fonctions d’ordre supérieur

Fonctionnelles

Une **fonctionnelle** est une fonction dont l'argument est une fonction.

Exemple : Une suite arithmétique est une fonction qu'associe un entier naturel à tout entier naturel.

$$\begin{aligned} \mathit{suite} &: \mathbb{N} \to \mathbb{N} \\ n &\mapsto \begin{cases} 1, & \text{si } n = 0 \\ \mathit{suite}(n-1) + 2, & \text{si } n > 0 \end{cases} \end{aligned}$$

Nous voulons maintenant définir une fonctionnelle qui calcule la raison d'une suite arithmétique donnée.

$$\begin{aligned} \mathit{raison} &: (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ f &\mapsto f(1) - f(0) \end{aligned}$$

La fonction raison associe un entier naturel à toute fonction qu'associe un entier naturel à tout entier naturel. Nous avons, par exemple, raison(suite) = 2.

Fonctionnelles

```
suite n
   | n == 0 = 1
   | n > 0 = suite (n - 1) + 2

raison f = (f 1) - (f 0)
```

```
Prelude > suite 0
1
Prelude > suite 1
3
Prelude > raison suite
2
```

Nous pouvons utiliser raison avec d'autres suites arithmétiques :

```
suite2 n
    | n == 0 = 1
    | n == 1 = suite (n - 1) + 5


Prelude> suite2 0
1
Prelude > suite2 1
6
Prelude > raison suite2
5
```

Fonctions d'ordre supérieur

Une **fonction d'ordre supérieur** est une fonction dont le résultat est une fonction

- Une **fonction d'ordre 1** est une fonction dont le résultat n'est pas une fonction.

- Une **fonction d'ordre 2** est une fonction dont le résultat est une fonction d'ordre 1

- Une **fonction d'ordre n** est une fonction dont le résultat est d'ordre n – 1

Fonctions d'ordre supérieur

Exemple : Nous voulons créer une fonction *sign* qu'à tout entier n associe une fonction fn.

$$sign : \mathbb{N} \to \mathbb{R} \to \mathbb{R}$$
$$n \mapsto f_n$$

$$f_n(x) = (-1)^n x$$

```
sign n = \x -> (-1) ^ n * x
```

```
Prelude > (sign 5) 3.5
-3.5
```

```
Prelude > sign 5 3.5
-3.5
```

En Haskell, l'application est associative à gauche. Cela veut dire que : f x z est équivalent à (f x) z.

Fonctionnelles d'ordre supérieur

Nous pouvons calculer cela avec l'approximation de Simpson :

$$\int_b^a f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Nous pouvons définir la fonction intégrale qui, **étant donné une fonction f** intégrable sur R, **revoie la fonction** qui à tout intervalle [a; b] associe une valeur approchée de l'intégrale de a à b de f.

```
integrale f = \ (a, b) =  ((b - a)/6) * ((f a) +
     4 * (f ((a + b) / 2))+
    (f b))
```

```
Prelude> f x = x * x
Prelude> (integrale f) (0, 1)
0.3333333

Prelude> (integrale (\x -> x * x)) (0, 1)
0.3333333
```

Fonctionnelles d'ordre supérieur

La fonction *dérivé* qui, étant donné une fonction f dérivable sur un intervalle de R, renvoie une fonction qui à tout point x0 associe une valeur approchée de f′(x0). Nous pouvons utiliser la définition suivante :

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

En prenant h suffisamment petit, nous avons en Haskell :

```
derivee f = \ x0 -> let h = 0.001 in
                    ((f (x0 + h)) - (f x0)) / h
```

```
Prelude> f x = x * x
Prelude> (derivee f) 1
2.0009999

Prelude> (derivee (\x -> x * x)) 1
2.000999
```

Différentes types de définition

Ces définitions sont équivalentes en Haskell :

```
suiv n = n + 1
suiv = \ n -> n + 1
```

Ainsi que :

```
add a b = a + b
add a = \ b -> a + b
add = \ a -> \ b -> a + b
```

Il y a donc n + 1 façons différentes de définir une fonction d'ordre n.

Application partielle

En programmation fonctionnelle il est légal d'appliquer une fonction d'ordre n à moins d'n arguments

```
Prelude> add= \x ->\y -> x + y
Prelude > suiv = add 1
Prelude > suiv 2
3
```

Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Curryfication

Il y a deux possibilité pour résoudre un problème dont le nombre de donnée à l'entrée est 2.

```
add1 (a, b) = a + b
add2 a b = a + b
```

Ces deux définitions ne sont pas équivalente puisque leurs types sont différents.

```
Prelude > :type add1
add1 :: Num a => (a, a) -> a
Prelude > :type add2
add2 :: Num a => a -> a -> a
```

On dit que la fonction *add2* est la forme curryfiée de *add1*. (Le terme vient du nom du mathématicien Haskell Curry.)

La forme curryfiée présente l'intérêt de permettre l'application partielle.

Exercice

Écrivez une fonction curry qu'à toute fonction non-curryfiée de deux arguments associe sa version curryfiée.

```
curry f = \ a -> \ b -> f (a, b)
```

Écrivez une fonction uncurry qu'à toute fonction curryfiée de deux arguments associe sa version non-currifiée.

```
uncurry f = \ (a, b) -> f a b
```

Notez que maintenant nous pouvons faire :

```
Predlude> add1 (a, b) = a + b
Prelude > add2 = curry add1
```

ou bien :

```
Prelude> add2 a b = a + b
Prelude > add1 = uncurry add2
```

Polymorphisme

Polymorphisme

Du grec, « poly » = plusieurs et « morphê » = formes.

Une expression est polymorphe quand elle peut servir sans modifications dans de contextes différents

Un grand intérêt du polymorphisme est de permettre **d'écrire une seule fonction** qui prend en argument des valeurs appartenant de plusieurs types, plutôt que d'écrire plusieurs fonctions.

Le polymorphisme permet donc de r**endre un programme plus général**

Polymorphisme

Exemple : La fonction identité :

```
identite x = x
```

```
Prelude > identite 1
1
Prelude > identite “oui"
"oui"
```

Exemple : La fonction « swap » :

```
swap (x, y) = (y, x)
```

```
Prelude > swap (1, 2)
(2,1)
Prelude> swap (1, "oui")
(“oui",1)
```

Variable de type

Une variable de type identifie un type quelconque (parmi les types possibles)

Un type est polymorphe si son expression comporte une variable de type non instanciée ou instanciée avec un type polymorphe

```
Prelude > identite x = x
Prelude> :type identite
identite :: t -> t
```

La variable de type t n'est pas instanciée. Donc, le type de la fonction *identite* est polymorphe.

```
Prelude> swap (x, y) = (y, x)
Prelude > :type swap
swap :: (t1, t) -> (t, t1)
```

Les variables de type t et t1 ne sont pas instanciées. Donc, le type de la fonction swap est polymorphe.

Variable de type

```
Prelude > const_un x = 1
Prelude> :type const_un
const_un :: Num t1 => t -> t1
```

La variable de type t1 est instanciée avec un type polymorphe et la variable t ne l'est pas. Donc, le type de la fonction const_un est polymorphe.

```
Prelude > const_a x = 'a'
Prelude> :type const_a
const_a :: t -> Char
```

La variable de type t n'est pas instanciée. Donc, le type de la fonction *const_a* est polymorphe.

Exercice

Un exemple classique de fonction polymorphe est la fonctionnelle qui, étant donné deux fonctions f et g, renvoie f ○ g, c'est à dire, la composée de g par f. Écrivez cette fonction en Haskell.

```
compose f g = \ x -> f (g x)
```

Quell est le type de compose ?

```
Prelude> :type compose
compose :: (t -> t1) -> (t2 -> t) -> t2 -> t1
```

Le type de la variable f est t -> t1 et le type de g est t2 -> t. Le type de retour de g est identique à celui de l'argument de f.

Fonction compose

Maintenant nous pouvons créer des nouvelles fonctions avec compose :

```
Prelude> add1 x = x + 1
Prelude> add2 x = x + 2
Prelude > add3 = compose add1 add2
Prelude > add3 3
6
```

En Haskell, nous avons un opérateur infixé pour faire la composition de fonctions.

L'expression f . g est équivalente à compose f g.

```
Prelude > add3 = add1 . add2
Prelude > add3 3
6
```

Inférence de type

L'inférence de type est un processus implémenté par le contrôleur de types d'un langage fortement typé

Il permet de **déterminer automatiquement et statiquement le type le plus général de toute expression du langage** (sans qu'il soit mentionné explicitement dans le programme).

Nous allons voir l'algorithme W (du système de type HM) qui est utilisé par Haskell et OCaml.

Règles d'inférence de type

Une règle d'inférence de type est de la forme générale suivante :

$$\begin{cases} e_1 : t_1 \\ e_2 : t_2 \\ \cdots \\ e_n : t_n \end{cases} \Rightarrow e_0 : t_0$$

- e_0 est une expression du langage
- les e_i sont de sous-expressions de e_0.
- les t_i sont des expressions de type.

La règle dit que :

«Si e_1 est de type t_1 et…et e_n est de type t_n, alors e_0 est de type t_0»

Les $e_i : t_i$ sont les prémisses et $e_0 : t_0$ est la conclusion.

Quelques règles d'inférence de type

Définition de fonction \ e1 -> e2

$$\begin{cases} e_1 : t_1 \\ e_2 : t_2 \end{cases} \quad \underset{fun}{\Rightarrow} : t_1 \to t_2$$

Application de fonction e1 e2 :

$$\begin{cases} e_1 : t_1 \to t_2 \\ e_2 : t_1 \end{cases} \quad \underset{app}{\Rightarrow} : t_2$$

Conditionnelle if e1 then e2 else e3 :

$$\begin{cases} e_1 : Bool \\ e_2 : t \\ e_3 : t \end{cases} \quad \underset{cond}{\Rightarrow} : t$$

Opérateurs e1 + e2 :

$$\begin{cases} e_1 : Num \\ e_2 : Num \end{cases} \quad \underset{+}{\Rightarrow} : Num$$

Inférence de type

Exemple : Inférence du type de l'expression 1 + 2 :

$$\begin{cases} 1 : Num \\ 2 : Num \end{cases} \Rightarrow 1 + 2 : Num \quad {}^{+}$$

Exemple : Soit l'expression

abs = \ x -> if x > 0 then x else -x.

D'abord nous appliquons la règle pour l'opérateur *let* :

1. $\begin{cases} \texttt{abs} : t_1 \\ \texttt{\textbackslash\ x -> if x > 0 then x else -x} : t_1 \end{cases} \overset{let}{\Rightarrow} : t_1$

Ensuite, la règle pour l'opérateur (\) :

2. $\begin{cases} \texttt{x} : t_2 \\ \texttt{if x > 0 then x else -x} : t_3 \end{cases} \overset{fun}{\Rightarrow} : t_2 \rightarrow t_3$

3. $t_1 = t_2 \rightarrow t_3 \quad (2,3)$

Inférence de type

Maintenant, la règle pour l'opérateur if :

$$4.\quad \begin{cases} \mathtt{x} > \mathtt{0} : t_4 \\ \mathtt{x} : t_2 \\ -\mathtt{x} : t_2 \end{cases} \overset{cond}{\Rightarrow} : t_2$$

Nous avons :

$$5.\quad t_3 = t_2 \qquad (2,4)$$

$$6.\quad t_1 = t_2 \to t_2 \quad (3,5)$$

La règle pour l'opérateur > :

$$7.\quad \begin{cases} \mathtt{x} : t_2 \\ \mathtt{0} : t_2 \end{cases} \overset{>}{\Rightarrow} : \mathtt{Bool}$$

$$8.\quad t_4 = \mathtt{Bool} \quad (4,5)$$

Inférence de type

La règle pour les constantes :

$$9. \quad \{\mathtt{0} : \mathtt{Num}$$

La règle pour l'opérateur – :

$$10. \quad \{\mathtt{x} : \mathtt{Num} \quad \bar{\Rightarrow} : \mathtt{Num}$$

Ceci nous permet d'inférer :

11.	$t_2 = \mathtt{Num}$	(7, 9) et aussi (7, 10)
12.	$t_1 = t_2 \rightarrow t_2$	(3, 11)
13.	$t_1 = \mathtt{Num} \rightarrow \mathtt{Num}$	(11, 12)

Donc, le type de l'expression : `abs = \ x -> if x > 0 then x else -x` est :

$$Num \rightarrow Num$$

Inférence de type

Exercice : Quel est le type de l'expression : `double = \ f -> \ x -> 2 * (f x)`?

1. $\begin{cases} \texttt{double} : t_1 \\ \texttt{\textbackslash\ f -> \textbackslash\ x -> 2 * (f x)} : t_1 \end{cases} \overset{let}{\Rightarrow} : t_1$

2. $\begin{cases} \texttt{f} : t_2 \\ \texttt{\textbackslash\ x -> 2 * (f x)} : t_3 \end{cases} \overset{fun}{\Rightarrow} : t_2 \rightarrow t_3$

3. $\begin{cases} \texttt{x} : t_4 \\ \texttt{2 * (f x)} : t_5 \end{cases} \overset{fun}{\Rightarrow} : t_4 \rightarrow t_5$

4. $\begin{cases} \texttt{2} : \texttt{Num} \\ \texttt{(f x)} : \texttt{Num} \end{cases} \overset{*}{\Rightarrow} : \texttt{Num}$

5. $\begin{cases} \texttt{f} : t_4 \rightarrow t_6 \\ \texttt{x} : t_4 \end{cases} \overset{app}{\Rightarrow} : t_6$

Inférence de type

Nous avons :

6.	$t_6 = \texttt{Num}$	$(4, 5)$
7.	$t_5 = \texttt{Num}$	$(3, 4)$
8.	$t_3 = t_4 \to t_5$	$(2, 3)$
9.	$t_3 = t_4 \to \texttt{Num}$	$(7, 8)$
10.	$t_2 = t_4 \to t_6$	$(2, 5)$
11.	$t_2 = t_4 \to \texttt{Num}$	$(6, 10)$
12.	$t_1 = t_2 \to t_3$	$(1, 2)$
13.	$t_1 = (t_4 \to \texttt{Num}) \to t_3$	$(11, 12)$
14.	$t_1 = (t_4 \to \texttt{Num}) \to (t_4 \to \texttt{Num})$	$(9, 13)$

Donc, le type de l'expression est :

$$(t_4 \to \texttt{Num}) \to t_4 \to \texttt{Num}$$

Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.