Programmation Fonctionnelle
Système de Types

Michael Lampis

2023-2024

Types – Rappel

- Haskell utilise un système de types **strict**.
  - Chaque expression a un type.
  - Quand une fonction est appliquée, les arguments doivent avoir le **bon type** (?).

À comparer (C)

```
int a[3] = {1,2,3}; int b=7; char c='x';
a[2] = b; //OK
a = b; //Error!
b = c; //OK (?)
```

Exemples en Haskell

```
3 + "a" -- Type Error!
head [1,2,3] -- OK
head [] -- Exception mais type est bon!
head [1,2] [3] -- Type Error!
```

Motivation

- Quel est le but d'un système de types ?
  - Idéal : un programme a un bon type **si et seulement si** le programme est correcte.
  - Évidemment pas atteignable...
- Objectif :
  - Rendre automatique la détection d'erreurs de programmation
  - ...sans encombrer le programmeur
  - Idéal : donner une preuve mathématique que le programme satisfait quelques conditions élémentaires.
- Comment équilibrer les deux premiers objectifs est à discuter...

Alternatif 1 – Tout déclarer, Types fixés

- Exemples : C, C++, Java, Pascal, ...
- Principe : pour utiliser n'importe quelle variable ou fonction il faut au préalable informer le compilateur de son type.
- Le compilateur utilise ces déclarations pour détecter des erreurs.
- Types doivent correspondre, avec une certaine souplesse.

```
int g1(int); //declaration
int g2(double);
int f()
{
        int x; //sinon ne peut pas utiliser x
        double y;
        char *t = "Hi!";
        g1(x); g2(x); //OK
        g1(y); //OK (warning?)
        g2(t); //Type error!
}
```

Alternatif 2 – Rien déclarer, types dynamiques

- Exemple : Python
- Principe : compilateur infère le type (sans déclaration)
- Le type d'une variable peut évoluer

```
x=3
print x
x="hi"
print x
if 13>5:
 x=7
print x+5
```

- Avantage : plus de liberté
- Désavantage : moins de garantis
  - si on avait `if 3>5` comme condition ?
  - La validité de la condition n'est pas connue en compile-time
  - $\rightarrow$ programme qui peut terminer avec erreur ou pas!

Alternatif 3 – Qui a besoin de types?

- Exemples : javascript, Lisp
- Principe : ne déclare rien, on fait le maximum de conversions possible pour que le programme marche

```
var x = 5
var y = [1,2,3]
x+y // -> "51,2,3"
x*y // -> NaN
x-y // -> NaN
(x*y)+""+(x*y) // -> "NaNNaN"
```

- Avantage : on évite l'obligation de s'occuper de types
- Désavantage : le programme peut avoir des comportements étranges, difficiles à prévoir
- Approche normalement utilisée par de langage de scripts

Le système de types de Haskell

- Première priorité : programmes corrects
  - Système de types strict.
  - Chaque fonction/variable a un type fixe et précis.
  - Conversions automatiques limitées.
- Deuxième priorité : faciliter la programmation
  - Déclaration de types facultative.
  - Type non-déclaré $\rightarrow$ compilateur infère un type.
- Dans ce cours :
  - **Obligatoire** de déclarer les types de toutes les fonctions.
  - Facultatif pour les variables locales (notamment dans `where`, `let`)

Déclarer les types

- Pour déclarer un type on utilise l'opérateur `::`
- Pour tester le type d'une expression (en `ghci`) on utilise la commande `:t`.

Exemple :

```
f :: Int -> Int
f 0 = 0
f x = x+2
```

Dans `ghci` on a

```
*Main> :t f
f :: Int -> Int
*Main> :t f 3
f 3 :: Int
*Main> :t f [3]
<interactive>:1:3: error:...
```

Les types

Qu'est-ce que un type ?

- Définition mathématique : type $==$ ensemble (potentiellement infini) de valeurs possibles.
  - Entendu : `t` a type `T` $\Rightarrow$ alors `t` a une valeur dans l'ensemble `T`
  - `f` a type $a \to b$ veut dire que `f` est une fonction $a \to b$ (sens math)
- Les types élémentaires en Haskell
  - `Bool`. Ensemble : { `True, False` }
  - `Integer`. Ensemble : $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - `Int`. Ensemble : comme `Integer` mais avec valeur abs max 2^{63}
  - `Char`. Ensemble : { `'a', 'b', 'c', ...` }
  - `String`. Chaînes de caractères.
  - `Float`.
  - `Double`. Nombres réels (représentation comme en C)

Type Checking

- La loi principale de vérification de types en Haskell
- (Rappel : opération principale en Haskell : application d'une fonction)

> Si `x :: a` et `f :: a -> b`
> $\Rightarrow$ alors `f x :: b`

```
f1 :: Int -> Bool
f2 :: [Int] -> Int
f2 [1,2,3] :: Int
f1 (f2 []) :: Bool
f2 ( if (f1 5) then [5] else [6]) :: Int
```

- Vérification : Le compilateur applique la loi de façon exhaustive
- Résultat :
  - Soit un type (programme correct)
  - Soit une opération irréductible (erreur)

Les Listes

- Catégorie de types avec la forme `[a]`, où `a` est un type.
- Représente toutes les listes (ordonnées) dont les éléments ont type `a`.
- Points importants :
  - Tous les éléments d'une liste ont le même type
  - La taille de la liste n'est pas fixée et peut varier de 0 (liste vide) jusqu'à l'infini (grâce à l'évaluation paresseuse)

Exemples :

```
f1 :: [Int] -> [Int]
  -- fonction qui prend liste de Int
  -- et retourne liste de Int
[ "a", "b", "c" ] :: [ String ]
[ ["a","b"], ["c"] ] :: [ [ String ] ]
[ "a", ["b"] ] -- Error !
[ "a" ] ++ [["b"]] -- Error !
[] ++ [["b"]] :: [ [ String ] ] -- OK?!?
```

Les Tuples

- Tuple : vecteur dont chaque élément a un type prédéfini.
- Comme les listes, représentent des collections de données
- Différences :
  - Taille toujours finie et déclarée en avance.
  - Éléments peuvent avoir des types différents.

Exemples :

```
myadd :: (Int, Int) -> Int
myadd (x,y) = x+y
  -- fonction qui prend un tuple de deux entiers
(True, ["a"], 5) :: (Bool, [String], Int)
(5, (2,5), False) :: (Int, (Int, Int), Bool)
(5,4) == (2+3,2+2) :: Bool
  -- donne True
```

- Note : pas de tuples de taille 1. (Pourquoi ?)

Fonctions de plusieurs variables

- On a vu comment définir le type d'une fonction

```
f :: [Int] -> Bool
```

- Note : la déclaration de type n'implique pas forcement que la fonction soit totale

```
head [] -- Type OK mais exception!
```

- Comment déclarer une fonction de plusieurs variables ?
- Alternatif 1 : tuples

```
mysum :: (Int, Int) -> Int
mysum (x,y) = x + y
```

- OK, mais pas trop pratique. (!)

Curryfication

- Le nom de Haskell fait hommage à **Haskell Curry**
  `https://fr.wikipedia.org/wiki/Haskell_Curry`
  - Logicien qui au début du XXéme siècle a posé les bases mathématiques de la programmation fonctionnelle.
- Les fonctions de plusieurs variables sont définies grâce à la notion de la “Curryfication” (Currying)
- Principe :
  - La seule opération de notre langage est l’application de fonctions **d’une variable**
  - Or, on écrit par exemple `(+) 3 5`
  - à savoir, on applique la fonction `(+)` sur `3` et **puis** `5`
  - Interprétation `(+) 3 5` équivaut à `( (+) 3) 5`
  - Donc, le type de `(+) 3` est `Int->Int`
    - Pourquoi ?
  - Quel est donc le type de la fonction `(+)` ?

Curryfication

Curryfication

- Une fonction f qui prend des argument de types `a1,a2,...,an` et retourne une valeur de type `b` et déclarée comme :

```
f :: a1 -> a2 -> a3 -> ... -> an -> b
```

- L'opérateur `->` est right-associative, c'est-à-dire on a

```
f :: a1 -> (a2 -> a3 -> ... -> an -> b)
```

Exemple :

```
myadd :: Int -> Int -> Int
myadd x y = x + y
```

- Pour appeler la fonction on écrit `myadd 5 8`
- ...qui équivaut à `(myadd 5) 8`
- Quels sont les types des expressions
  - `(myadd 5) 8` ?
  - `myadd 5` ?
- Par conséquent le type de `myadd` doit être ..?

Curryfication – Implications

- La curryfication nous donne une base mathématique très solide pour définir notre langage
  - Pas besoin de faire la distinction entre fonctions d'une ou de plusieurs variables
- Un effet secondaire dont on peut se servir : on peut **partiellement appeler** une fonction !

Exemple :

- Rappel : on a vous une fonction `myapply f x = f x`
- Quel est son type ?
- Comment passer `myadd` à cette fonction pour augmenter la valeur de `x` par `2`?

Curryfication – Exemple

```
myadd :: Int -> Int -> Int
myadd x y = x+y

myapply :: (Int->Int)->Int->Int
myapply f x = f x
-- Pourquoi pas Int->Int->Int->Int ??

f1 :: Int -> Int
f1 x = x+2
```

On a

```
myapply f1 5 :: Int -- donne 7
myapply (myadd 3) 5 :: Int -- donne 8
myadd 3 :: Int->Int
myapply (myadd 2) :: Int -> Int
myapply ( (+) 2) 6 :: Int -- donne 8
```

Types Polymorphiques

- Selon les règles de la curryfication le type de `(+)` devrait donc être `Int -> Int ->Int`
- ou peut-être `Integer -> Integer -> Integer` ?
- ou `Float -> Float -> Float` ?
- Oui !

Si on essaye dans `ghci` on a

```
(+) :: Num a => a -> a -> a
```

- `a` est une **variable de type** (ignorez la partie `Num a` pour ce slide)
- Donc, la fonction `(+)` est une fonction qui est définie pour plusieurs types différents
- Pour chaque type `a`, il s'agit d'une fonction qui prend un élément de type `a` et retourne une fonction de type `a->a`
- Autrement dit, c'est une fonction qui prend deux argument du même type et retourn un résultat de ce type.

Types Polymorphiques

- La plupart de fonctions de Haskell qui utilisent des listes sont polymorphiques :

```
head :: [a] -> a
tail :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
(!!) :: [a] -> Int -> a
```

La variable de type est instancié quand on utilise une telle fonction (et le type de l'expression qui en résulte est inféré)

```
head ['a','b'] :: Char
```

- Le compilateur infère que, pour que l'expression soit correcte (== pour que la loi de types soit applicable) il faut que `a == Char`
- Le compilateur essaye toujours d'éviter de préciser le type autant que possible.

```
head [[]] :: [t]
```

Classes de types

- Rappel : un type est un ensemble de valeurs
- Une **classe** est un ensemble de types
  - Idée : une classe est une collection de types qui partagent la capacité d'effectuer une opération.
  - Exemple : l'opération `+` s'applique aux types `Int, Integer, Float, Double,...`, donc on considère que ces types forment une **classe**
- Les principales classes de types de Haskell

Classe	Opérations	Types	Descriptif
Eq	==, /=	Tous sauf fonctions	test égalité
Ord	<, <=, >, >=	Types basiques	comparaisons
Show	`show`	sauf fonctions	affichage
Num	+, −, *	types numériques	opérations de math
Integral	`div,mod`	entiers	
Fractional	/, `recip`	rationnels	

Comment utiliser les classes

- Rappel : fonction polymorphique == fonctionne pour plusieurs types
- Exemple `myadd x y = x+y`
- Or, `myadd` n'a pas comme type `a->a->a`
  - Parce que `myadd` est définie seulement si la fonction `(+)` est définie pour le type `a`
- Pour mettre de pré requis de type on utilise l'opérateur `=>`
- `Num a => a->a->a` veut dire que cette définition s'applique seulement pour les types qui appartient à la classe `Num`

Exemple :

```
mymax (x:[]) = x
mymax (x:xs)
   | x>y = x
   | otherwise = y
     where y = mymax xs

 mymax:: Ord a => [a] -> a
```

Comment définir un type

Trois méthodes de définir un nouveau type :

- `type`
- `newtype`
- `data`

Pourquoi trois méthodes ?

- `type <newtype-name> = <existing-type>`
  - Donne un synonym d'un type existant.
  - Pas de définitions récursives !
- `data <Constructor> = .. | .. | ..`
  - Permet de définir de type plus compliqués
  - Même de structures de données infinies
  - À revoir plus tard. . .
- `newtype <newtype-name> = ..`
  - Comme `type` mais nouveau type pas considéré comme synonym. (Compilateur ne l'accepte pas à la place de l'ancien type)

Définir un type

Exemples d'usage de `type`

```
type String = [Char]
-- Definition de Haskell, String n'est pas type de base

type Coordinates = (Int, Int)
type Move = Coordinates -> Coordinates
-- OK

type Tree = (Int, [Tree])
-- Pas OK!! definition recursive

type Pair a = (a,a)
first :: Pair a -> a
first (x,_) = x

first (3,5) :: Int -- Donne 3
```