Programmation Fonctionnelle
Définition de types et de classes

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

13 octobre 2017

Plan

Nouveaux noms pour les types : `type`

Nouveaux types : `data` (et `newtype`)

Les types récursifs

Les records

Définition des classes

Exercices

Plan

Les déclarations `type`

En Haskell, un nouveaux *nom* d'un type peut se définir par le mot clé `type`.

```
type String = [Char]
```

`String` est un synonyme pour le type `[Char]`.

Les déclarations de type sont utilisées pour rendre les autres types (et surtout les types des fonctions) plus faciles à lire.

Par exemple, étant donnée la déclaration

```
type Position = (Int,Int)
```

nous pouvons écrire

```
origine :: Position
origine = (0,0)

allerAGauche :: Position -> Position
allerAGauche (x,y) = (x-1,y)
```

D'autres exemples – des TPs et TDs

```
-- nombres complexes
type Complexe = (Float,Float)

-- images
type Pixel = Int
type Ligne = [Pixel]
type Image = [Ligne]
type Voisinage = [Pixel]
type Effet = (Pixel,Voisinage) -> Pixel
appliquerEffet :: Effet -> Image -> Image

-- nim
type Echiquier = [Int]
```

Déclaration de types avec paramètres

Les déclarations de type peuvent avoir des paramètres.

Par exemple, étant donnée la déclaration

```
type Paire a = (a,a)
```

nous pouvons écrire

```
mult :: Paire Int -> Int
mult (m,n) = m*n

copy :: a -> Paire a
copy x = (x,x)
```

Les déclarations de type peuvent être incrémentielles :

```
type Position = (Int,Int)
type Transition = Position -> Position
```

Par contre, elle ne peuvent pas être récursives :

```
type Tree = (Int,[Tree])
```

NO!!!

Plan

Les déclarations `Data`

On définit *nouveau type* par le mot clé `data`.

Par exemple, on trouve dans `Prelude` la déclaration suivante :

```
data Bool = False | True
```

> *`Bool` est un nouveau type,*
> *avec exactement deux (nouveaux) valeurs, `False` et `True`.*

Les constructeurs

Remarque(s) :

- Les deux valeurs `False` et `True` sont appelés *constructeurs* du type `Bool`.
- Les noms des *types* et des *constructeurs* débutent toujours par une majuscule.
- Les déclarations `data` spécifient ceux qui sont les éléments du nouveau type. Les constructeurs d'un type servent construire les expressions, les valeurs, et les motifs ayant ce type.
- ...voir plus tard... Cf. les grammaires hors-contexte, qui spécifient les expressions bien formées d'un langage.

Les constructeurs

Remarque(s) :

- Les deux valeurs `False` et `True` sont appelés *constructeurs* du type `Bool`.
- Les noms des *types* et des *constructeurs* débutent toujours par une majuscule.
- Les déclarations `data` spécifient ceux qui sont les éléments du nouveau type. Les constructeurs d'un type servent construire les expressions, les valeurs, et les motifs ayant ce type.
- . . . voir plus tard . . . Cf. les grammaires hors-contexte, qui spécifient les expressions bien formées d'un langage.

Différence entre `type` et `data`

Mot clé	Quoi est neuf ?	Type déclaré
`type`	le nom	préexistant
`data`	le type	est crée

Les valeurs d'un nouveau type s'utilisent comme les valeurs des autres types.

Par exemple, étant donnée la déclaration

```
data Reponse = Oui | Non | Inconnu
```

nous pouvons définir les fonctions suivantes :

```
reponses :: [Reponse]
reponses = [Oui,Non,Inconnu]

flip :: Reponse -> Reponse
flip Oui = Non
flip Non = Oui
flip Inconnu = Inconnu
```

`data` : les paramètres des constructeurs.

Les constructeurs dans une déclaration `data` peuvent avoir des paramètres.
C'est-à-dire : on construit à partir d'objets d'autres types.

Par exemple, étant donné la déclaration

```
data Shape = Circle Float | Rect Float Float
```

nous pouvons définir

```
square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Remarque(s) :

- Syntaxe : dans

```
data Shape = Circle Float | Rect Float Float
```

  - `data` est le mot clés,
  - `Shape` est le nom du type,
  - `Circle` est le nom du premier constructeur du type `Shape`, dont les paramètres sont de type `Float`,
  - `Rect` est le nom du deuxième constructeur du type `Shape`, dont les deux paramètres sont de le type `Float`.

- Sémantique : `Shape` a valeurs de la forme `Circle r` où `r` est un valeur flottant, et de la forme `Rect x y` où `x` et `y` sont des valeurs flottants.

- `Circle` and `Rect` peuvent se voir comme des *fonctions qui construisent des valeurs* de type `Shape` :

```
Circle :: Float -> Shape
Rect :: Float -> Float -> Shape
```

Les types déclarés avec `data` peuvent aussi être paramétrés.

Par exemple, on trouve dans `Prelude` :

```
data Maybe a = Nothing | Just a
```

d'où :

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

Le type `Maybe` offre une première façon de traiter les erreurs :
on retourne `Nothing` si erreur.

`newtype`

- définit un nouveau type,
- avec un seul constructeur,
- implémentation dans le compilateur optimisée par rapport à `data`.

Exemple avec `newtype`

```
newtype Complexe = Paire (Float,Float)
```

- Comme pour `data` :

  `Complexe` est un nouveau type

- mais optimisé de façon interne :

  car les deux types, `(Float,Float)` et `Complexe`, sont isomorphes
  (c'est-à-dire, les éléments des deux types sont en bijection).

- Attention : en Haskell, quand un type a un seul constructeur, on a l'habitude à nommer ce constructeur et le type de la même façon :

```
newtype Complexe = Complexe (Float,Float)
```

Plan

Types récursifs

Les nouveaux types – déclarés par `data` – peuvent se définir en terme de soi-mêmes. On dit alors qu'ils sont *récursifs*.

Exemple :

```
data Nat = Zero | Succ Nat
```

> *`Nat` est un nouveau type, avec constructeurs `Zero :: Nat` et `Succ :: Nat -> Nat`.*

Utilisation du type `Nat`

Remarque(s) :

- Un valeur de type `Nat`
  - soit il est `Zero`,
  - soit il est de la forme `Succ n`,
    où `n` est un valeur de type `Nat`.

- C'est-à-dire, `Nat` contient la suite infinie de valeurs suivants :

```
Zero
Succ Zero
Succ (Succ Zero)
...
```

- Nous pouvons considérer les valeurs de type `Nat` comme les *nombres naturels* (entiers non-négatifs), où
  - `Zero` représente `0`,
  - `Succ` représente la fonction successeur $\lambda x \to x + 1$.

- Par exemple, la valeur

```
Succ (Succ (Succ Zero))
```

représente le nombre naturel

$$1 + (1 + (1 + 0)) = 3$$

C'est facile – en utilisant filtrage et récursion – de convertir les valeurs de type `Nat` vers `Int`, et vice-versa :

```
nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n -1))
```

et ainsi les additioner :

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

Par ailleurs, c'est plus intéressant (et efficace) de définir l'addition directement sur les nombres naturels par récursion :

```
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Par exemple :

```
add (Succ (Succ Zero)) (Succ Zero)
  = Succ (add (Succ Zero) (Succ Zero))
  = Succ (Succ (add Zero (Succ Zero))
  = Succ (Succ (Succ Zero))
```

Remarque(s) :

- La définition récursive de `add` corresponde aux lois

$$0 + n = n, \qquad (1 + m) + n = 1 + (n + m).$$

- La définition de la fonction `add` utilise les deux motifs :

  `Zero` `Succ m`

  (à la gache du symbole d'égalité)

Remarque

- La syntaxe des déclarations `type`, en particulier celles *récursives*, est semblable à celle des grammaires hors-contexte.
- Les grammaires hors-contexte spécifient les expressions bien formées d'un langage.
- De façon analogue, les déclarations `data` spécifient ceux qui sont les valeurs du type déclaré.

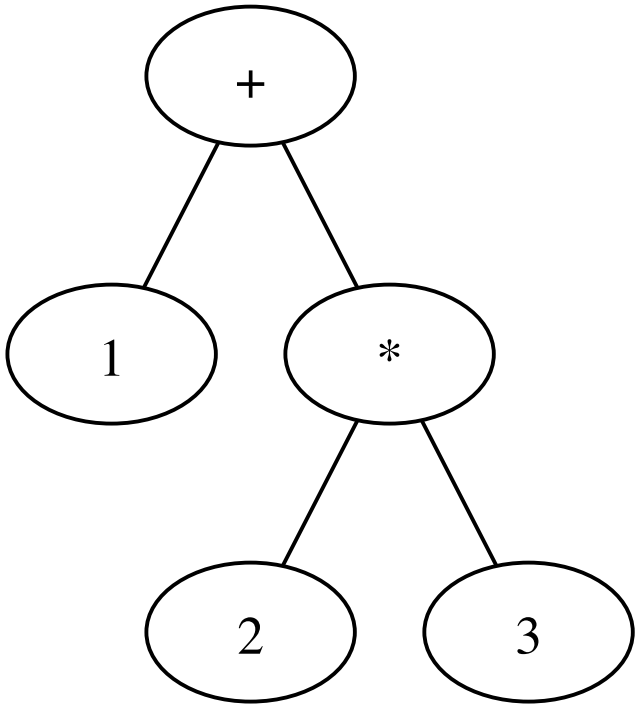
Expressions arithmétiques

On considère une forme elementaire d'expressions, construites des entiers en utilisant l'addition et la multiplication.

Par exemple :

$1 + 2 * 3$

ou, en forme arborescente :



On définit le type des expressions – par récursion – par :

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

L'expression du transparent précédent est representée par :

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

En utilisant le filtrage par motifs et la récursion, on définit des fonctions pour manipuler ces expressions.

Par exemple :

```
taille :: Expr -> Int
taille (Val n) = 1
taille (Add x y) = 1 + taille x + taille y
taille (Mul x y) = 1 + taille x + taille y

eval :: Expr -> Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

- Les trois constructeurs ont les types suivants :

```
Val :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

- On peut définir un schéma récursif analogue à `foldr` pour les listes :

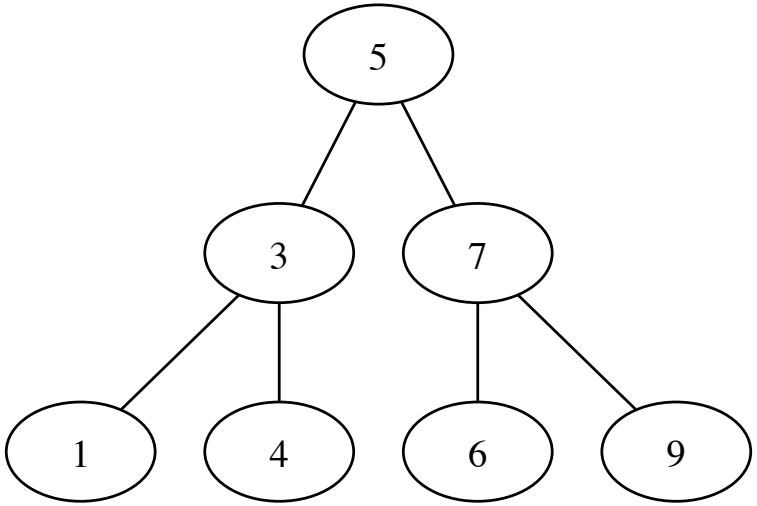
```
foldExpr :: (Int -> a) ->
            (a -> a -> a) ->
            (a -> a -> a) -> (Expr -> a)
foldExpr f g h (Val n) = f n
foldExpr f g h (Add x y) = g (foldExpr f g h x)
                             (foldExpr f g h y)
foldExpr f g h (Mul x y) = h (foldExpr f g h x)
                             (foldExpr f g h y)
```

- Plusieurs fonctions sur les expressions peuvent se définir en utilisant `foldExpr`. Par exemple :

```
eval = foldExpr id (+) (*)
taille = foldExpr (\_ -> 1) (+) (+)
```

Arbres (de recherche) binaires

C'est souvent utile de stocker les données dans des arbres binaires :



Spécification d'arbre binaire :

- ou bien c'est une feuille etiquetée par un entier,
- sinon, c'est un noeud, avec
  - un fils à la gauche (un autre arbre binaire),
  - un entier comme étiquette,
  - un fils à la droite (un autre arbre binaire).

Le mécanisme de déclaration `data` permet d'implémenter tout de suite cette spécification :

```
data ArbreBinaire = Feuille Int
   | Noeud ArbreBinaire Int ArbreBinaire
```

L'arbre du transparent précédent se représente alors comme suit :

```
Noeud (Noeud (Feuille 1) 3 (Feuille 4))
      5
      (Noeud (Feuille 6) 7 (Feuille 9))
```

On définit une fonction qui décide si un entier a une occurrence dans un tel arbre :

```
occurs :: Int -> ArbreBinaire -> Bool
occurs m (Feuille n) = m==n
occurs m (Noeud l n r) = m==n
        || occurs m l
        || occurs m r
```

Mais . . . dans le pire de cas – si l'entier n'a pas une telle occurrence – cette fonction parcours l'arbre entier.

Considérez la fonction `flatten` qui transforme un arbre dans un liste, en lisant les étiquettes de l'arbre selon un parcours en profondeur gauche :

```
flatten :: ArbreBinaire -> [Int]
flatten (Feuille n) = [n]
flatten (Noeud l n r) =
     flatten l ++ [n] ++ flatten r
```

Un arbre binaire est un *arbre de recherche binaire* si la liste retournée par `flatten` est triée.

Par exemple, l'arbre de notre example est un arbre de recherche binaire, car la liste retournée est `[1,3,4,5,6,7,9]`.

Les arbres de recherche binaires ont la propriété importante que, quand on souhaite chercher un valeur, on peut toujours décider dans quel sous-arbre chercher :

```
occurs m (Feuille n) = m==n
occurs m (Noeud l n r)
  | m==n = True
  | m<n = occurs m l
  | m>n = occurs m r
```

Cette nouvelle définition est bien plus efficient, car elle parcours seulement une branche de l'arbre.

Si l'arbre est *balancé*, alors une branche a au plus $\log_2(n)$ éléments, où n est la taille de l'arbre.

Plan

Les records

Les records :

```
data Person = P {
  name :: String,
  surname :: String,
  birthYear :: Int
  }

eMacron :: Person
eMacron = P {
  name="Macron",
  surname="Emmanuel",
  birthYear=1977
  }
```

Les records comme tuplets (ou produits)

```
type Person' = (String,String,Int)

eMacron' :: Person'
eMacron' = ("Macron","Emmanuel",1977)

fst3 (x,_,_) = x
snd3 (_,y,_) = y
thrd3 (_,_,z) = z
```

Les records comme tuplets (ou produits) II

Les fields des records se comportent comme des projections :

```
*Main> fst3 eMacron'
"Macron"
*Main> :t fst3 eMacron'
fst3 eMacron' :: String
*Main> :t fst3
fst3 :: (t, t1, t2) -> t

*Main> name eMacron
"Macron"
*Main> :t name eMacron
name eMacron :: String
*Main> :t name
name :: Person -> String
```

En jargon, les fields sont des destructeurs.

Place à l'imagination

```
data Entity = Android {id :: Int}
            | Human {names::[String],
                     surname::String }
            | Colony [Entities]
```

Plan

Déclaration d'une classe (`class`)

On trouve, dans `Prelude`, la déclaration suivante :

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

La classe `Eq` a deux methodes :

- *l'égalité : `==`*
- *l'inegalité : `/=`*

L'inagalité y est aussi définie – de façon stantard – en fonction de l'égalité.

Instancier une classe (`instance`)

`Bool` est un membre de la classe `Eq`.
On trouve, dans `Prelude`, la déclaration suivante :

```
instance Eq Bool where
    False == False = True
    True == True = True
    _ == _ = False
```

Le type `Bool` *appartient à la classe* `Eq` *:*

- *on donne la définition de l'égalité,*
- *on donne pas la définition de l'inegalité,*
  – on utilisera alors la définition standard.

Extension de classes

La classe `Ord` est une extension de la classe `Eq` :

```
class Eq a => Ord a where
    (<),(<=),(>),(>=) :: a -> a -> Bool
    min, max :: a -> a -> a
```

En plus de `==` et `/=`, `Ord` a six autres methodes :
`<`, `<=`, `>`, `>=`, `min`, `max`

Remarques :

- Seulement un nouveau type – déclaré via `data` – peut être une instance d'une classe.
- Dans la déclaration de `Ord`, on trouve les définitions suivantes de `min` et `max` :

```
class Eq a => Ord a where

  ...

    min x y
                | x <= y = x
                | otherwise = y

    max x y
                | x <= y = y
                | otherwise = x
```

Instanciation automatique (`deriving`)

Nous pouvons instancier un nouveau type à des classes prédéfinie, via le mot clés `deriving` :

```
data Reponse =  Non | Inconnu | Oui
                deriving (Eq,Ord,Show,Read)
```

Par exemple :

```
*Main> Non < Inconnu
True
```

Définir un module

Par exemple :

```
module Complexe (
                  Complexe,
                  re,
                  im,
                  construct,
                  origin,
                  modulus,
                  angle
                )
    where

newtype Complexe = Paire (Float,Float)
    deriving (Eq)

...
```

Plan

Exercices I

1. Utilisez la récursion et la fonction `add` pour définir une fonction qui multiplie le nombre naturels.
2. Définissez une fonction fold adaptée aux expressions, et donnez quelques exemple d'utilisation.
3. Précisez ce que veut dire qu'un arbre binaire est balancé. Définissez une fonction qui décide si un arbre binaire est balancé.
4. Definissez le type de `Complexe` nombre complexes, et faites de lui une instance de la classe `Num`.
5. Definissez le type `Image` des images, et instanciez ce type à la classe `Show`.

Exercices II

6. Voici une possible déclaration de la classe `Set` :

```
class Set s where
    appartient :: Eq a => a -> s a -> Bool
    ajouter :: Eq a => a -> s a -> s a
    enlever :: Eq a => a -> s a -> s a
    pick_one :: Eq a => s a -> a
    vide :: s a
    est_vide :: s a -> Bool
    union :: Eq a => s a -> s a -> s a
    intersection :: Eq a => s a -> s a -> s a
    difference :: Eq a => s a -> s a -> s a
```

Proposez deux types – un basé sur les listes, l'autre sur les arbres binaires – et instanciez cette classe à ces types.