Programmation Fonctionnelle
Définitions de Types

Michael Lampis

2023-2024

Les Types

- Rappel : Type = Ensemble de valeurs
- On a vu plusieurs types de base (`Int`,`Char`,`Float`,`..`) et des façons de les combiner pour obtenir des types plus compliqués (`[Int]`, `(Int,Char)`, `[Int]->Char`).
- Haskell nous permet aussi de définir nos propres types à partir de zéro.
  - Indispensable pour définir les types des structures des données plus sophistiquées.
- Mot–clé : `data`

Exemples :

```
data MyBool = MyFalse | MyTrue
data Direction = North | South | East | West
```

Data – Utilisation basique

- On donne une liste de toutes les valeurs possibles, séparées par `|`.
- Cela définit un nouveau type.
- On peut l'utiliser dans une fonction.
- **NB**: le **pattern matching** fonctionne automatiquement.

```
data Direction = North | South | East | West

goingwhere :: Direction -> String
goingwhere North = "You are going North"
goingwhere _ = "You are not going North"

*Main> goingwhere North
"You are going North"
*Main> goingwhere East
"You are not going North"
```

Data

- Souvent on a besoin d'emballer plusieurs valeurs dans le même type (cf. Tuples, `struct` en C)
- Exemple : on peut utiliser un type `Person` qui stocke le nom, prénom et age d'une personne.

```
data Person = Person String String Int

whoareyou :: Person -> String
whoareyou (Person nom prenom _) = "You are "
                                  ++prenom++ " "++nom

*Main> whoareyou (Person "Bond" "James" 52)
"You are James Bond"
```

- **Attn:** `Person` n'est pas un synonyme pour `(String,String,Int)`
- Le compilateur les considère comme des types distincts.

Data – Syntaxe alternative

- Puisque souvent on utilise des types “containers” (comme `Person`), Haskell nous permet d’utiliser une syntaxe alternative et **nommer** les composants du type.

```
data NewPerson = NewPerson { firstname :: String,
                             lastname :: String,
                             age :: Int,
                             height :: Int }
```

Cela donne des fonctions qui correspondent aux noms des champs.

```
*Main> :t firstname
firstname :: NewPerson -> String
*Main> firstname $ NewPerson "James" "Bond" 52 180
"James"
```

NB `NewPerson` est aussi une fonction (constructeur)

```
*Main> :t NewPerson
NewPerson :: String -> String -> Int -> Int -> NewPerson
```

Data – Point

- De la même façon, on peut définir un type pour les points sur le plan bi-dimensionel

```
data Point = Point Integer Integer

moveright :: Point -> Point
moveright (Point x y) = Point (x+1) y

*Main> moveright $ Point 5 4
<interactive>:32:1: error:
 No instance for (Show Point) arising from a use of 'print
 In a stmt of an interactive GHCi command: print it
```

- Pourquoi ça ne marche pas ??
- Pourquoi `Integer` et pas `Float` ?

Show

- L'expression `moveright (Point 5 4)` est correcte et a type `Point`
- L'erreur donnée par ghci indique que le compilateur ne sait pas **comment afficher** un objet de ce type.
- Techniquement, le type `Point` ne fait pas partie de la classe `Show`.

Rappel :

```
class Show a where
..
  show :: a -> String
```

- Chaque type instance de `Show` implémente la fonction `show`
- Grâce à cette fonction, ghci peut nous afficher les types de base

```
Prelude> show 5.2
"5.2"
Prelude> :info Float
..
instance Show Float -- Defined in 'GHC.Float'
```

Rappel – 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
		sauf fonctions	
Show	`show`		affichage
Num	+, −, *	types numériques	opérations de math
Integral	`div,mod`	entiers	
Fractional	/, `recip`	rationnels	

Deriving

- Pour afficher des objets de type `Point` il faut que `Point` soit une instance de `Show`.
- Soit on implémente une fonction `show :: Point->String` (on va le faire plus tard)
- Soit on utilise l'implémentation "par défaut" comme suit :

```
data Point = Point Integer Integer deriving Show

*Main> moveright (Point 3 4)
Point 4 4
```

- `deriving` donne une implémentation standard qui
  - "Affiche" le nom du constructeur
  - Puis applique `show` sur chaque composant de `Point`
  - Concatène les résultats.
- Il faut que les types composants soient instances de `Show`

Deriving

- On peut faire `deriving` de plusieurs classes.

```
data Point = Point Integer Integer deriving (Show,Eq,Ord)

*Main> (Point 4 4) == moveright (Point 3 4)
True
*Main> (Point 4 4) < moveright (moveright (Point 3 4))
False
```

- Pour `Eq` on applique `==` sur chaque composant, retourne `True` si tous égaux.
- Pour `Ord` on applique `<` sur premier composant, si `==` on avance au deuxième, ...
- **Attn**: on ne peut faire `deriving` que pour quelques classes précises.
- `data Point = Point Integer Integer deriving Num` donne erreur !

Read

- Parmi les classes pour lesquelles on peut faire `deriving` on a `Read` qui donne la fonction `read`

```
Prelude> :t read
read :: Read a => String -> a
```

- La fonction `read` nous permet de convertir un `String` en objet de notre classe.

```
Prelude> (read "1.2" )/ 2.0
0.6
```

```
data Point = Point Integer Integer deriving (Show,Read)
```

```
*Main> moveright (read "Point 3 4")
Point 4 4
```

Read – pièges

- Le type de `read` est `Read a => String->a`
- Cela peut poser des problèmes, parce que ce n'est pas toujours possible pour le compilateur de préciser la bonne valeur pour la variable de type `a`.

```
*Main> read "Point 3 4"
*** Exception: Prelude.read: no parse
```

- Source d'ambigüité : l'expression `read "Point 3 4"` a type de retour variable.
- Si on utilise l'expression comme argument, l'ambiguïté est levée grâce à la signature de la fonction.
- Sinon, on peut préciser explicitement le type attendu.

```
*Main> read "Point 3 4" :: Point
Point 3 4
```

Enumerations

- La classe `Enum` couvre les types dont on peut énumérer les éléments.
- Par exemple : `Int, Char, Bool`

```
class  Enum a  where
    succ      :: a -> a
    pred      :: a -> a

*Main> succ 5
6
*Main> succ 'a'
'b'
*Main> succ 1.2
2.2
```

- `succ` donne l'élément suivant
- `pred` donne l'élément précédent

Enumerations

- On peut faire `deriving Enum` pour un type qui ne contient pas d'élément d'un autre type.

```
data Point = Point Integer Integer deriving (Show,Enum)
-- ERROR!

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Enum,Show)
--OK

*Main> [Mon .. Thu]
[Mon,Tue,Wed,Thu]
*Main> [Mon, Wed ..]
[Mon,Wed,Fri,Sun]
```

Point Paramétrique

- Rappel : pourquoi `Point Int Int` et pas `Point Float Float` ?
- On peut déclarer le type `Point` comme un type paramétrique, ce qui permet à l'utilisateur d'utiliser le type de son choix pour le contenu.

```
data Point a = Point a a deriving (Show,Read)
```

- **Attn:** maintenant, `Point` n'est plus un type. C'est une "fonction" qui, étant donné un type `a`, définit un autre type `Point a`.
- Donc, `Point Integer` et `Point Float` sont des types.

```
moveright :: Num a => Point a -> Point a
moveright (Point x y) = Point (x+1) y

*Main> moveright (Point 3 5) -- donne Point 4 5
*Main> moveright (Point 3.2 5) -- donne Point 4.2 5.0
```

- **Attn:** la fonction `moveright` a besoin du contexte `Num a` (pourquoi?)

Call me Maybe

- Exemple de type paramétrique : `Maybe a`
- Représente un objet qui peut avoir type `a` ou peut être vide (souvent pour signifier une erreur)

```
data Maybe a = Nothing | Just a
  deriving (Show, Ord, Eq, Read)
```

```
*Main> :t Just 3
Just 3 :: Num a => Maybe a
*Main> :t Just 'a'
Just 'a' :: Maybe Char
*Main> Nothing
Nothing
*Main> :t Nothing
Nothing :: Maybe a
```

Maybe – Exemples

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv x y = Just (div x y)

*Main> safediv 5 2
Just 2
*Main> safediv 5 0
Nothing

safeadd :: Maybe Int -> Maybe Int -> Maybe Int
safeadd (Just x) (Just y) = Just (x+y)
safeadd _ _ = Nothing

*Main> safeadd (safediv 5 2) (safediv 5 3)
Just 3
*Main> safeadd (safediv 5 2) (safediv 5 0)
Nothing
```

Maybe – Exemples

- Étant donné une liste de `Maybe`, comment garder que les éléments qui ne sont pas `Nothing`?

```
catMaybes :: [Maybe a] -> [a]
catMaybes ms = [ x | Just x <- ms ]
```

Exemple :

```
*Main> map  (safediv 6)  [-3..3]
[Just (-2),Just (-3),Just (-6),Nothing,Just 6,Just 3,Just
*Main> catMaybes  (map  (safediv 6)  [-3..3])
[-2,-3,-6,6,3,2]
```

Maybe – Exemples

- Comment faire `map` d'une fonction sur une liste, et garder que les résultats réussis ?

```
mapMaybe :: (a->Maybe b) -> [a] -> [b]
mapMaybe f xs = catMaybes (map f xs)
-- mapMaybe f = catMaybes . map f

*Main> mapMaybe (safediv 6) [-3..3]
[-2,-3,-6,6,3,2]
```

- `Maybe` est comme une liste dont la taille et 0 ou 1.
- Un peu pénible à utiliser directement...
- Le point commun avec les listes va jouer un rôle intéressant plus tard...

Classes sans deriving ?

- On a vu comment utiliser le mot–clé `deriving` pour signifier qu'un type **hérite** la fonctionnalité par défaut d'une classe.
- Or, peut-être la fonctionnalité par défaut ne nous convient pas.

Exemple :

```
data MyRational = MyRational Int Int deriving (Show,Eq)

*Main> MyRational 3 5
MyRational 3 5
*Main> MyRational 3 5 == MyRational 6 10
False
```

On aurait aimé que :

- `show (MyRational 3 5)` donne, par exemple `"3/5"`
- $3/5 == 6/10$ retourne `True`

Il faut donc réimplémenter les fonctions `show, (==)` pour notre type.

Instance

- Le mot–clé `instance` nous permet de déclarer qu'un type appartient à une classe de types.
- Attn : chaque classe a une collection de fonctions de base qui **doivent** être implémentées pour chaque type de la classe.

Exemple :

```
*Main> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

- Ici le compilateur nous dit que la classe `Eq` a les fonctions `(==), (/=)`.
- Une implémentation **minimale** de cette classe doit contenir soit `(==)` soit `(/=)`.

Instance

- Il faut déclarer `MyRational` comme instance de `Eq, Show`
- . . . et donner au moins des implémentations pour les fonctions obligatoires de deux classes.

```
data MyRational = MyRational Int Int

instance Show MyRational where
  show (MyRational x y) = show x ++ "/" ++ show y

instance Eq MyRational where
  MyRational x1 y1 == MyRational x2 y2 = x1*y2 == x2*y1

*Main> xs = zipWith MyRational [1..5] [2..6]
*Main> show xs
"[1/2,2/3,3/4,4/5,5/6]"
*Main> elem (MyRational 8 10) xs
True
```

Rational

En fait, le type `Rational` existe déjà en Haskell !

- Pour les lignes suivantes il faut importer le module GHC.Real (qui contient le type `Rational`).

```
Prelude> import GHC.Real
Prelude GHC.Real> toRational 0.75
3 % 4
Prelude GHC.Real> (3%4) + (2%3)
17 % 12
Prelude GHC.Real> (3%8) < (1%3)
False
Prelude GHC.Real> zipWith (%) [1..4] [3..6]
[1 % 3,1 % 2,3 % 5,2 % 3]
Prelude GHC.Real> sum $ take 10 $
        zipWith (%) (repeat 1) $ map (2^) [1..]
1023 % 1024
```

Types récursifs

Types récursifs

- Le mot–clé `data` nous permet d'utiliser le type qu'on est en train de définir comme élément constituant de ce type.
- Résultat : types de données récursifs.
- On a déjà vu un exemple : les listes.
- Une liste est
  - Soit la liste vide `[]`
  - Soit `(:) head tail`
- Effectivement, on peut définir notre propre version de listes.

```
data MyList a = Empty | Cons a (MyList a)
```

- Une `MyList` est soit `Empty` (fonction constructeur qui ne prend pas d'argument),
- soit une liste construite avec `Cons`, qui prend un argument de type `a` et une autre liste (du même type).

Listes – Réinventées

- **NB:** Juste pour rigoler

```
data MyList a = Empty | Cons a (MyList a)

instance Show a => Show (MyList a) where
 show Empty = ""
 show (Cons a Empty) = show a
 show (Cons a xs) = show a ++ ", " ++ show xs

mylength :: MyList a -> Int
mylength Empty = 0
mylength (Cons _ xs) = 1+mylength xs

list2mylist :: [a] -> MyList a
list2mylist [] = Empty
list2mylist (x:xs) = Cons x (list2mylist xs)
```

Arithmétique Peano

- **NB:** Pour rigoler (part 2)
- Rappel : arithmétique Peano
  - Il existe un élément 0
  - Il existe une fonction $\text{succ} : \mathbb{N} \to \mathbb{N}$.
  - $\forall x\ 0 \neq \text{succ}(x)$.
  - $\forall x, y\ \text{succ}(x) = \text{succ}(x) \Rightarrow x = y$
- On peut utiliser un nouveau type pour représenter les nombres naturels.

```
data MyInt = Zero | Succ MyInt deriving Eq
```

- Idée : chaque $x \in \mathbb{N}$ est soit égale à 0, soit le successeur d'un autre naturel.
- On peut faire `deriving Eq` (pourquoi c'est cohérent ?)
- On aimerait que `MyInt` soit instance de `Ord,Num,Show`

Arithmétique Peano

```
int2myint :: Integer -> MyInt
int2myint 0 = Zero
int2myint n = Succ $ int2myint $ n-1

myint2int :: MyInt -> Integer
myint2int Zero = 0
myint2int (Succ x) = 1 + myint2int x

instance Show MyInt where
 show = show . myint2int
-- Cheating by using int!
```

- Rappel : il suffit de définir `show` pour faire `MyInt` une instance de la classe `Show`

```
*Main> show (int2myint 6)
"6"
```

Arithmétique Peano

Pour `Ord`?

```
*Main> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```

- La classe `Ord` contient pas mal de fonctions
- Mais, il suffit de définir soit `compare` soit `<=` et les autres fonctions sont obtenues automatiquement.

Ordering ?

Rappel :

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
```

- C'est quoi `Ordering` ?

```
*Main> :info Ordering
data Ordering = LT | EQ | GT    -- Defined in 'GHC.Types'
```

- La fonction `compare` retourne une valeur qui est soit `LT, EQ, GT` (less than, equal, greater than).
- Donc, une autre manière pour implémenter la fonctionnalité de la classe `Ord` est d'implémenter `compare` (à la place de `<=`).

Arithmétique Peano

```
instance Ord MyInt where
 Zero <= _ = True
 (Succ x) <= (Succ y) = x<=y
 _ <= _ = False
```

- **NB**: C'est crucial qu'on a fait `deriving Eq` (sinon, il faudrait définir aussi `(==)`).
- Maintenant on peut utiliser `MyInt` avec n'importe quelle fonction qui prend un `Ord a=>a`.

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (<x) xs)
                   ++ filter (==x) (x:xs)
                   ++ quicksort (filter (>x) xs)
*Main> quicksort (map int2myint [3,1,3,2,4,2,5,1,6])
[1,1,2,2,3,3,4,5,6]
```

Arithmétique Peano

```
*Main> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger,
                                    (negate | (-)) #-}
```

- Addition, Multiplication, Valeur Absolue, `int2myint`
- `signum` retourne $1, 0, -1$ (signe). `negate` ??
- **NB**: Soustraction pas obligatoire !
  - Parce que soustraction est définie comme $x - y := x + (-y)$
  - Ne marche pas pour nous !
  - Mais, on a le droit de donner notre propre définition.

Arithmétique Peano

```
myadd :: MyInt -> MyInt -> MyInt
myadd Zero x = x
myadd (Succ x) y = Succ (myadd x y)

mymult :: MyInt -> MyInt -> MyInt
mymult Zero y = Zero
mymult (Succ x) y = myadd y (mymult x y)

mysub :: MyInt -> MyInt -> MyInt
mysub x Zero = x
mysub Zero _ = Zero
mysub (Succ x) (Succ y) = mysub x y
```

Arithmétique Peano

```
instance Num MyInt where
 (+) = myadd
 (-) = mysub
 (*) = mymult
 abs = id
 signum Zero = Zero
 signum _ = Succ Zero
 fromInteger = int2myint
 negate = id -- !!?
*Main> (int2myint 5)-(int2myint 3)
2
*Main> (int2myint 5)*(int2myint 3)
15
```

Arbres binaires

- Rappel : Un arbre binaire est une structure où chaque nœud a deux enfants et contient une valeur.
- Structure de données de grande importance.
- Définition de type récursif en Haskell

```
data BinTree a = Leaf
               | Branch (BinTree a) a (BinTree a)

data BinTree' a = Leaf'
                | Branch' { left :: BinTree' a,
                            this :: a,
                            right :: BinTree' a}
```

- Définitions équivalentes.
- Deuxième a l'avantage des fonctions `left,this,right`

Arbres binaires

Exemples :

```
list2tree :: [a] -> BinTree a
list2tree [] = Leaf
list2tree (x:xs) = Branch l x r
   where
     l = list2tree $ take (s `div` 2) xs
     r = list2tree $ drop (s `div` 2) xs
     s = length xs

instance Show a => Show (BinTree a) where
 show Leaf = ""
 show (Branch l x r) = show x
                     ++ " ("++show l++")"
                     ++ " ("++show r++")"
*Main> list2tree [1..10]
1 (2 (3 () ()) (4 () (5 () ()))) (6 (7 () (8 () ()))
                                    (9 () (10 () ())))
```

Arbres binaires

Exemples :

```
tree2list :: BinTree a -> [a]
tree2list Leaf = []
tree2list (Branch l x r) = [x] ++
                           tree2list l ++
                           tree2list r

occurs :: Eq a => a -> BinTree a -> Bool
occurs _ Leaf = False
occurs x (Branch l y r) = (x==y)
                          || occurs x l
                          || occurs x r
```

- Que font ces fonctions ?