Chapitre 4 : Types, classes et modules

Carole Porrier
carole.porrier@univ-paris13.fr

Département d'informatique
IUT de Villetaneuse

4 avril 2024

Paradigmes de développement universels

Table des matières

Table des matières

Types énumératifs

- Nous avons déjà vu les **types énumératifs**:

```
data Bool = False | True
data Jour = Dim | Lun | Mar | Mer | Jeu | Ven | Sam
data Mois = Jan | Fev | Mrs | Avr | Mai | Jun |
            Jui | Aou | Sep | Oct | Nov | Dec
```

- Il est possible de créer des **types algébriques**:

```
data Vecteur3D = Vecteur3D Float Float Float
    deriving (Show)
data Point = Point2D Float Float |
             Point3D Float Float Float deriving (Show)
data Triangle = Triangle Point Point Point
    deriving (Show)
```

```
*Main> Triangle (Point2D 1 1) (Point2D 3 5) (Point2D (-1)
    4)
Triangle (Point2D 1.0 1.0) (Point2D 3.0 5.0) (Point2D
    (-1.0) 4.0)
```

Types algébriques

```
data Vecteur3D = Vecteur3D Float Float Float
    deriving (Show)
data Point = Point2D Float Float |
             Point3D Float Float Float deriving (Show)
data Triangle = Triangle Point Point Point
    deriving (Show)
```

- `Vecteur3D`, `Point` et `Triangle` sont des **types algébriques**
- `Vecteur3D`, `Point2D`, `Point3D` et `Triangle` sont des **constructeurs**.
- On peut réutiliser le **même nom** pour les **constructeurs** et pour leur **type**.

Enregistrements (1/2)

- Souvent, on doit fournir des **fonctions d'accès** au contenu;

```
data Etudiant = Etudiant String String String Int
    deriving (Show)

prenom :: Etudiant -> String
prenom (Etudiant p _ _ _) = p

nom :: Etudiant -> String
nom (Etudiant _ n _ _) = n

codePermanent :: Etudiant -> String
codePermanent (Etudiant _ _ c _) = c

age :: Etudiant -> Int
age (Etudiant _ _ _ a) = a
```

- Une alternative consiste à utiliser les **enregistrements**.

Enregistrements (2/2)

```
data Etudiant = Etudiant {
    prenom         :: String,
    nom            :: String,
    codePermanent  :: String,
    age            :: Int
} deriving (Show)

*Main> let e = Etudiant "Bob" "LeMineur" "LEMB12345678" 35
*Main> e
Etudiant {prenom = "Bob", nom = "LeMineur", codePermanent = "
    LEMB12345678", age = 35}
*Main> prenom e
"Bob"
*Main> nom e
"LeMineur"
*Main> codePermanent e
"LEMB12345678"
*Main> age e
35
```

Constructeurs de types

- Les types **énumératifs**, **algébriques** et **enregistrements** sont présents dans **la plupart des langages**;
- En Java et en C++, il y a la notion de *template* (types **génériques**):
  - En **Java**: `TreeSet<E>`, `HashMap<K,V>`, etc.
  - En **C++**: `std::vector<T,Alloc>`, `std::map<Key,T,Compare,Alloc>`, etc.
- On aimerait voir `TreeSet` et `vector` comme des **fonctions**:

$$\begin{array}{rcccc} \texttt{TreeSet} & : & \text{Type} & \to & \text{Type} \\ & & \texttt{E} & \mapsto & \texttt{TreeSet<E>} \end{array}$$

$$\begin{array}{rcccc} \texttt{vector} & : & \text{Type} \times \text{Type} & \to & \text{Type} \\ & & (\texttt{T},\texttt{Alloc}) & \mapsto & \texttt{vector<T,Alloc>} \end{array}$$

Types paramétrés

- En Haskell, ce type de construction est **facile**;
- Il suffit d'utiliser une **variable de type**;
- Ainsi que le mot réservé `data`:

```
-- Equivalent d'avoir des objets "null" en Java/C++
data Maybe a = Nothing | Just a
-- Union de deux types
data Either a b = Left a | Right b
-- Vecteurs de n'importe quoi
data Vector3D a = Vector3D a a a

-- Somme
somme :: Num a => Vector3D a -> Vector3D a -> Vector3D a
somme (Vector3D x1 y1 z1) (Vector3D x2 y2 z2)
    = Vector3D (x1 + x2) (y1 + y2) (z1 + z2)
-- Produit scalaire
produitScalaire :: Num a => Vector3D a -> Vector3D a -> a
produitScalaire (Vector3D x1 y1 z1) (Vector3D x2 y2 z2)
    = x1 * x2 + y1 * y2 + z1 * z2
```

Le type `Maybe a`

- En Haskell, le type `Maybe a` est très pratique;
- Disponible dans le module Data.Maybe;
- Utile lorsque le **domaine** de la fonction n'est pas **partout défini**;

```
Prelude> import Data.Maybe
Prelude> import Data.List
-- Recherche selon un critère
Prelude> :t find
find :: (a -> Bool) -> [a] -> Maybe a
Prelude> find odd [100,103..200]
Just 103
Prelude> find odd [100,104..200]
Nothing
Prelude> :t fromJust -- Permet d'extraire une valeur
fromJust :: Maybe a -> a
Prelude> fromJust $ find odd [100,103..200]
103
```

Le type `Map k v`

- Un autre type utile est `Map k v`;
- Permet de représenter un **tableau associatif** (clé/valeur);
- Disponible dans le module Data.Map.

```
Prelude Data.Map> import Data.Map
Prelude Data.Map> let map = fromList [(0,"fraise"), (1, "pomme")
    , (2, "banane")]
Prelude Data.Map> map
fromList [(0,"fraise"),(1,"pomme"),(2,"banane")]
Prelude Data.Map> keys map
[0,1,2]
Prelude Data.Map> Data.Map.lookup 1 map
Just "pomme"
Prelude Data.Map> Data.Map.lookup 5 map
Nothing
```

Table des matières

Types récursifs

- On peut définir un type de donné par rapport à **lui-même**;
- C'est pratique en particulier pour les structures **arborescentes**:

```
-- Un arbre binaire
data ArbBin a = ArbBinVide |
    NoeudArbBin (ArbBin a) a (ArbBin a)

-- Un arbre binaire dans lequel on distingue les feuilles
data ArbBin2 a = ArbBinVide2 | FeuilleArbBin2 a |
    NoeudArbBin2 (ArbBin a) a (ArbBin a)

-- Un arbre avec un nombre quelconque d'enfants
data Arbre a = ArbreVide | NoeudArbre [Arbre a]
```

Exemple: un système de fichiers

```haskell
-- Un système de fichiers simple
data Fichier = Fichier {
    nomFichier  :: String,
    taille      :: Int,
    permissions :: String
}
data Repertoire a = Repertoire String [a] [Repertoire a]
type SystemeFichier = Repertoire Fichier

systeme =
    Repertoire "~"
      [Fichier ".vimrc"     128 "rw-------",
       Fichier ".gitconfig" 32  "rw-------"]
      [Repertoire ".ssh"
        [Fichier "id_rsa.pub" 25 "rw-------"]
        [],
       Repertoire ".git"
        [Fichier "config" 38 "rw-------"]
        []
      ]
```

Types récursifs et filtrage

- On peut facilement définir une fonction qui utilise un **type récursif**;
- Souvent, on utilise le **filtrage** (*pattern matching*):
  - On traite d'abord le ou les **cas de base**;
  - Puis on traite le **cas général**.
- Évidemment, toutes les autres **constructions** sont aussi disponibles: fonctions d'ordre supérieur, `where`, `let...in`, etc.

Arbre binaire de recherche (1/2)

```haskell
-- Un arbre binaire de recherche
data ABR a = ABRVide | Noeud a          -- Contenu du noeud
                            (ABR a) -- Sous-arbre gauche
                            (ABR a) -- Sous-arbre droit
    deriving (Show,Eq)

-- L'insertion est tres facile
inserer :: Ord a => a -> ABR a -> ABR a
inserer x ABRVide = Noeud x ABRVide ABRVide
inserer x (Noeud y g d)
    | x == y = Noeud x g d
    | x < y  = Noeud y (inserer x g) d
    | x > y  = Noeud y g (inserer x d)

-- Verifier l'appartenance
appartient :: Ord a => a -> ABR a -> Bool
appartient _ ABRVide = False
appartient x (Noeud y g d)
    | x == y = True
    | x < y  = appartient x g
    | x > y  = appartient x d
```

Arbre binaire de recherche (2/2)

- On peut insérer en bloc grâce à la fonction **foldr**:

```
*Main> foldr inserer ABRVide [5,2,4,1,3,6]
Noeud 6 (Noeud 3 (Noeud 1 ABRVide (Noeud 2 ABRVide ABRVide)
    ) (Noeud 4 ABRVide (Noeud 5
 ABRVide ABRVide))) ABRVide
```

- Il s'agit d'un motif **très fréquent**, qui devrait être préféré à:

```
-- Insertions multiples (serait mieux avec foldr)
insererPlusieurs :: Ord a => [a] -> ABR a -> ABR a
insererPlusieurs [] = id
insererPlusieurs (x:xs) = insererPlusieurs xs . inserer x
```

- L'**ordre** des paramètres dans la **signature** indique s'il faut utiliser **foldr** ou **foldl**.

foldr ou foldl?

```
inserer :: Ord a => a -> ABR a -> ABR a
inserer x ABRVide = Noeud x ABRVide ABRVide
inserer x (Noeud y g d)
    | x == y = Noeud x g d
    | x < y  = Noeud y (inserer x g) d
    | x > y  = Noeud y g (inserer x d)

inserer' :: Ord a => ABR a -> a -> ABR a
inserer' ABRVide x = Noeud x ABRVide ABRVide
inserer' (Noeud y g d) x
    | x == y = Noeud x g d
    | x < y  = Noeud y (inserer' g x) d
    | x > y  = Noeud y g (inserer' d x)
```

```
*Main> foldr inserer ABRVide [5,2,4,1,3,6]
Noeud 6 (Noeud 3 (Noeud 1 ABRVide (Noeud 2 ABRVide ABRVide)) (
    Noeud 4 ABRVide (Noeud 5 ABRVide ABRVide))) ABRVide
*Main> foldl inserer' ABRVide [5,2,4,1,3,6]
Noeud 5 (Noeud 2 (Noeud 1 ABRVide ABRVide) (Noeud 4 (Noeud 3
    ABRVide ABRVide) ABRVide)) (Noeud 6 ABRVide ABRVide)
```

Table des matières

Rappel

- En Haskell, une **classe** est une **contrainte** de type;

```
Prelude> :m Data.List
Prelude Data.List> :t sort
sort :: Ord a => [a] -> [a]
Prelude Data.List> :t elem
elem :: Eq a => a -> [a] -> Bool
```

- On peut **trier** une liste si ses éléments sont **ordonnés**;
- On peut **vérifier l'appartenance** à une liste si ses éléments sont **"égalisables"**.
- Une classe peut **hériter** d'une autre classe:

```
class Eq a => Ord a where
    ...
```

Hiérarchie

```
Eq -> Ord -> Real -> Integral;
Eq -> Num -> Real -> Realfrac;
Show -> Num -> Fractional -> Floating -> Realfloat;
Enum -> Integral;
Fractional -> Realfrac -> Realfloat;
Monad -> Monadplus;
```

Définition minimale complète (1/2)

- La classe Eq est définie comme suit:

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

- Il est donc **suffisant** de définir seulement `(==)` ou seulement `(/=)`.
- On peut aussi définir **toutes les opérations** si c'est plus efficace;
- On dit alors que la **définition minimale complète** d'une instance de `Eq` est

```
(==) | (/=)
```

Définition minimale complète (2/2)

- La classe Ord est **à peu près** définie comme:

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    x `compare` y | x == y     = EQ
                  | x <= y     = LT
                  | otherwise = GT
    (<) :: a -> a -> Bool
    x < y = x `compare` y == LT
    (<=) :: a -> a -> Bool
    x <= y = x `compare` y /= GT
    (>) :: a -> a -> Bool
    x > y = x `compare` y == GT
    (>=) :: a -> a -> Bool
    x >= y = x `compare` y /= LT
```

- Sa **définition minimale complète** est

```
compare | (<=)
```

- Il est recommandé pour des raisons d'**efficacité** de définir `compare`.

Définir une instance de classe (`instance`)

- On souhaite définir un ordre sur les **points** du plan;
- Plus le point est **proche** de l'origine, plus il est **"petit"**;
- Si deux points sont à la **même distance** de l'origine, on prend celui dont l'angle est le **plus petit**.
- Noter que `Point` doit être une **instance** de `Eq`, puisque `Ord` en **hérite**.

```
data Point = Point Float Float deriving (Eq)

instance Ord Point where
    (Point x y) `compare` (Point x' y')
        | r == r'   = a `compare` a'
        | r < r'    = LT
        | otherwise = GT
        where r  = x * x + y * y
              r' = x' * x' + y' * y'
              a  = atan2 y x
              a' = atan2 y' x'
```

Définir une classe

- **Syntaxe:**

```
class <nom classe> <variable de type> where
    <fonction1> <signature>
    <fonction2> <signature>
    ...
```

- Par exemple, la classe des types **mesurables**:

```
class Mesurable a where -- Choses qui ont une longueur
    longueur :: a -> Int

instance Mesurable Int where
    longueur _ = 1

instance Mesurable Char where
    longueur _ = 1

instance Mesurable [a] where
    longueur = length

instance Mesurable (Maybe a) where
    longueur Nothing = 0
    longueur (Just _) = 1
```

En pratique

- Dans le cours, il sera assez **rare** que vous ayez à définir une classe (mais ça peut arriver, en particulier dans l'examen!);
- Souvent, les classes **prédéfinies** dans Haskell sont **suffisantes**;
- Par contre, il est très fréquent de définir une **instance** d'une classe déjà existante;
- Il est donc important d'être **à l'aise** avec le mécanisme.
- Si vous voulez aller plus loin, les **classes** Functor, Monoid et Foldable Applicative et Monad sont **fondamentales**.

Table des matières

Module Haskell

- Un **module** est simplement un **fichier**, par exemple **Graph.hs**;
- Il commence par une lettre **majuscule**;
- L'extension est **.hs**;
- Il contient
  - des **types**,
  - des **classes** et
  - des **fonctions**.
- Ceux-ci **collaborent** ensemble pour offrir différents services.

Programme Haskell

- Un **programme** est un **ensemble de modules**;
- Un de ces modules, le point d'entrée, doit s'appeler **Main.hs**;
- Les **modules** devraient vérifier les deux critères suivants:
  - **Cohésion forte**: Chaque **type**, **classe** et **fonction** a sa raison d'être **au sein** d'un module;
  - **Couplage faible**: Chaque **module** est le plus possible **indépendant** des autres modules.

Charger un module dans l'interpréteur

- Pour **charger** un module Haskell dans GHCi (par exemple **Graph.hs**), il suffit d'entrer la **commande**:

```
Prelude> :load Graph.hs
```

ou simplement

```
Prelude> :l Graph.hs
```

- Si un ou plusieurs modules **déjà chargés** ont été **modifiés**, on peut les mettre à jour avec

```
Prelude> :reload
```

ou simplement

```
Prelude> :r
```

Charger un module standard ou installé

- On peut aussi charger un module **standard** ou **installé** par Cabal;
- **Première** façon de faire:

```
Prelude> import Data.List
```

- **Deuxième** façon de faire:

```
Prelude> :m + Data.List
Prelude> :m + Data.List Data.Set Data.Char Data.Map
```

- La deuxième façon de faire n'est pas possible dans un **module Haskell**, seulement dans l'**interpréteur**.

Espace de nom

- Lorsqu'on **charge** un module, tous les types, classes et fonctions qu'il définit sont **disponibles** dans l'espace **global**;
- S'il y a **deux fonctions** qui ont le même nom, il y aura une **ambiguïté** quand on tentera de les utiliser:

```
Prelude> :m + Data.Set
Prelude Data.Set> :t map

<interactive>:1:1:
    Ambiguous occurrence 'map'
    It could refer to either 'Data.Set.map',
                             imported from 'Data.Set'
                             (and originally defined in 'containers-0.5.5.1:Data.Set.
                                  Base')
                         or 'Prelude.map',
                             imported from 'Prelude' (and originally defined in 'GHC.
                                  Base')
Prelude Data.Set>
```

Éviter les conflits de noms

Plusieurs façons d'éviter les **conflits de noms**.

- N'importer que le **type**, la **classe** ou la **fonction** dont on a besoin:

```
import Data.List (intersperse,sort)
import Graph (Graph,edges)
```

- Ou **cacher** les fonctions problématiques, si on n'en a pas besoin:

```
import Data.Set hiding (map)
```

- Si on a besoin de **deux fonctions** portant le même nom, il reste à **préfixer** par le nom du module:

```
Prelude> :m + Data.Set
Prelude> Data.Set.map (+3) (fromList [1,4,2,3,1,3])
fromList [4,5,6,7]
```

Synonymes

- Il est parfois long de **préfixer** les types, fonctions ou classes d'un module avec son **nom**:

```
Prelude Data.Map> (Data.Map.!) (Data.Map.fromList [(0,"
    alpha"),(1,"beta")]) 1
"beta"
```

- Il suffit alors d'importer le module en lui attribuant un **synonyme**:

```
Prelude Data.Map> import qualified Data.Map as Map
Prelude Data.Map Map> (Map.!) (Map.fromList [(0,"alpha")
    ,(1,"beta")]) 1
"beta"
```

- Cette façon de faire fonctionne aussi dans un **module**, pas seulement dans l'**interpréteur**.

Contexte

- Supposez que nous ayons un fichier **Module.hs** qui compile;
- Nous souhaitons en faire un **module Haskell**;
- Il est important de **déclarer** les types, classes et fonctions qu'on souhaite **exporter** (rendre **publics**);
- Il est aussi important de **documenter**
  - l'**en-tête** du module;
  - chaque type, classe et fonction qui sont **exportés**.
- Il est aussi **fortement encouragé** de mettre des **exemples** (doctest);
- Toute fonction **publique** devrait avoir sa signature **déclarée**.

Documentation d'en-tête de module

Format **reconnu** par **Haddock**:

```
{-|
Module      : Graph
Description : A module for playing with simple graphs
Copyright   : (c) Carole Porrier
License     : GPL-3
Maintainer  : carole.porrier@univ-paris13.fr
Stability   : experimental

This module provides functionalities for playing with simple
graphs. The data structure behing @Graph@ is a strict map,
associating each vertex with its neighbors. Notice that the
implementation is not optimal: Its main objective is to
explore the basic operations on lists and maps in Haskell.
 -}
```

Documentation de type/classe/fonction

- La première ligne seulement doit commencer par `-- |`;
- Toutes les autres commencent par `--` (n'oubliez pas l'espace);
- Optionnellement, on peut mettre des **exemples**, qui peuvent être testés automatiquement par Doctest.

```
-- | Adds a list of edges to a graph. This is equivalent
-- to calling 'addEdge' for every edge in the list.
--
-- >>> numVertices $ addEdges [(1,2),(3,4)] emptyGraph
-- 4
-- >>> hasEdge 4 3 $ addEdges [(1,2),(3,4)] emptyGraph
-- True
addEdges :: Ord v => [(v,v)] -> Graph v -> Graph v
```

Exportation

```
module Graph (
    -- * Type and basic constructor
    Graph, emptyGraph,
    -- * Constructor of famous graph families
    completeGraph, cycleGraph, wheelGraph,
    completeBipartiteGraph, petersenGraph,
    -- * Basic queries
    -- | The usual queries can be performed on a graph.
    numVertices, numEdges, hasVertex, vertices,
    neighbors, hasEdge, edges,
    -- * Basic insertions
    -- | Currently, only insertions are allowed
    -- (i.e. no deletions).
    addVertex, addEdge, addEdges,
    -- * Output
    -- | One might output the graph to a GraphViz string
    -- according to the @dot@ format.
    toGraphvizString,
    ) where
```

Makefile

```
HS_FILES = Main.hs Graph.hs Configuration.hs ISTree.hs
DOC_DIR = doc

.PHONY: clean doc docdir exec main test

main: $(HS_FILES)
	ghc -o main Main.hs

exec: main
	./main

doc: $(HS_FILES) docdir
	haddock -o $(DOC_DIR) --html $(HS_FILES)

docdir:
	mkdir -p $(DOC_DIR)

test: clean
	doctest $(HS_FILES)

clean:
	rm -f *.o *.hi
	rm -f main
	rm -rf doc
```