PLP

Algebraic Data Types

26 October 2025

Table des matières

1 Algebraic Data Types

En haskell, pour définir des types de données algébriques, on utilise le mot-clé `data`. Un type de données algébrique est une manière de définir un type en termes de ses constructeurs de données, qui peuvent être des valeurs simples ou des combinaisons de plusieurs valeurs.

```
data Color = Red | Green | Blue

rgb Red = (255, 0, 0)
rgb Green = (0, 255, 0)
rgb Blue = (0, 0, 255)
```

> **i Info**
>
> Dans ce genre d'exemple, nous parlons principalement d'`enum` (énumérations), qui sont des types de données algébriques simples avec plusieurs constructeurs sans arguments.

1.1 Types avec paramètres

Les types de données algébriques peuvent également avoir des paramètres, ce qui permet de créer des types plus génériques.

```
data Person = Person Int String Int

> Person 1 "John" 42
```

Ici, `Person` est un type de données algébriques avec trois champs : un entier pour l'ID, une chaîne pour le nom, et un entier pour l'âge.

1.2 Pattern Matching

Le pattern matching est une fonctionnalité puissante en Haskell qui permet de décomposer les valeurs des types de données algébriques en fonction de leurs constructeurs.

```
getName :: Person -> String
getName (Person _ name _) = name

setName :: String -> Person -> Person
setName name (Person id _ age) = Person id name age
```

Dans cet exemple, `getName` extrait le nom d'une personne en utilisant le pattern matching, tandis que `setName` crée une nouvelle instance de `Person` avec un nom mis à jour.

> **💡 Hint**
>
> Si l'on souhaite pouvoir afficher les données d'un type personnalisé, on peut dériver l'instance `Show` automatiquement en ajoutant `deriving (Show)` à la définition du type.
>
> ```
> data Color = Red | Green | Blue deriving (Show)
> ```

1.3 Terminologie

Chaque type algebrique ne peut avoir qu'un seul constructeur de type, mais ce constructeur peut avoir plusieurs formes (constructeurs de données). Cela permet de modéliser des structures de données complexes de manière claire et concise.

Par exemple, le type `Bool` est un type de données algébriques avec deux constructeurs de données : `True` et `False`. Cependant, on peut également lui rajouter des constructeurs de données supplémentaires.

```haskell
data Bool = True | False  -- Has 1+1 = 2 possible values
data TwoBools = TwoBools Bool Bool  -- Has 2*2 = 4 possible values
data Complex = Two Bool Bool | One Bool | None  -- Has 2*2 + 2 + 1 = 7 possible values
```

1.4 Exercices

Définir un type `area` qui permet calculer l'aire de différentes formes géométriques.

```haskell
data Shape = Circle Float | Rectangle Float Float deriving (Show)

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle w h) = w * h
```

2 Type paramétriques

Les types paramétriques permettent de définir des types de données génériques qui peuvent fonctionner avec n'importe quel type de données.

```
data Pair a b = Pair a b
```

Ici, `Pair` est un type de données paramétrique qui prend deux types `a` et `b` comme paramètres. On peut créer des paires de différents types de données en utilisant ce type.

```
intStringPair = Pair 1 "one"
floatBoolPair = Pair 3.14 True
```

2.1 Exercices

On souhaite définir un type `isNull` qui permet de vérifier si une valeur est nulle ou non.

```
data Nullable a = Null | Value a
isNull :: Nullable a -> Bool
isNull Null = True
isNull _ = False
```

3 Alias et nouveaux types

3.1 Type alias

Un alias de type permet de donner un nom alternatif à un type existant, ce qui peut améliorer la lisibilité du code.

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

3.1.1 Alias polymorphiques

Les alias de type peuvent également être polymorphiques, ce qui signifie qu'ils peuvent prendre des paramètres de type.

```
type Pair a b = (a, b)

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

3.2 Nouveaux types

Les nouveaux types permettent de créer des types distincts basés sur des types existants, offrant une meilleure sécurité de type.

```
newtype RGB = RGB (Int, Int, Int)
  deriving (Show)

> RGB (255, 0, 0)
```

Ici, `RGB` est un nouveau type qui encapsule un tuple d'entiers représentant les composantes rouge, verte et bleue d'une couleur.

> **i Info**
>
> `newtype` permet d'offrir une meilleure performance que `data` lorsqu'il n'y a qu'un seul constructeur avec un seul champ, car il n'introduit pas de surcharge supplémentaire au moment de l'exécution.

3.3 Différences type constructeurs et data constructeurs

Les types définis avec `newtype` ont des constructeurs de type et des constructeurs de données distincts. Le constructeur de type est utilisé pour définir le type, tandis que le constructeur de données est utilisé pour créer des valeurs de ce type.

4 Type de données récursifs

Les types de données récursifs sont des types qui se définissent en termes d'eux-mêmes. Ils sont souvent utilisés pour représenter des structures de données comme les listes et les arbres.

- Une liste est soit vide, soit un élément suivi d'une autre liste.

```haskell
data List t = Nil | Cons t (List t)
```

D'autres type recursifs existent comme:

```haskell
chars :: List Char
chars = Cons 'a' (Cons 'b' (Cons 'c' Nil))

bools :: List Bool
bools = Cons True (Cons False Nil)
```

4.1 Built-in lists

Haskell fournit un type de liste intégré qui est défini de manière similaire à notre définition personnalisée. Nous pouvons donc en déduire que les listes en Haskell sont des types de données récursifs.

4.2 Exercices

```haskell
data Matryoshka = Hollow | Nesting Matryoshka

dolls :: Matryoshka -> Int
dolls Hollow = 1
dolls (Nesting m) = 1 + dolls m
```

Ici, `Matryoshka` est un type de données récursif qui représente une poupée russe. La fonction `dolls` calcule le nombre de poupées imbriquées.

5 Maybe et Either

5.1 Gestion des erreurs

En Haskell, les types `Maybe` et `Either` sont utilisés pour gérer les erreurs et les valeurs optionnelles de manière sûre.

Etant donné que Haskell est fortement typé, il se peut que des erreurs surviennent lors de l'exécution d'un programme. Pour gérer ces erreurs, on utilise souvent le type `Maybe`, qui peut représenter soit une valeur valide (`Just value`), soit l'absence de valeur (`Nothing`).

Prenons le cas de la division par zéro :

```
div :: Float -> Float -> Float
duv x 0 = error "Division by zero"
```

5.2 Maybe

Pour éviter l'utilisation de `error`, on peut utiliser le type `Maybe` :

```
safeDiv :: Float -> Float -> Maybe Float
safeDiv x 0 = Nothing
safeDiv x y = Just (x / y)
```

Maybe permet donc d'encapsuler une valeur qui peut être absente, ce qui force le programmeur à gérer explicitement le cas où la valeur n'est pas présente.

5.3 Either

Le type `Either` est une autre façon de gérer les erreurs, en permettant de retourner soit une valeur valide, soit une erreur avec un message. Cela permet d'encapsuler des valeurs de deux types différents.

```
safeDiv :: Int -> Int -> Either String Int
safeDiv x 0 = Left "Division by zero"
safeDiv x y = Right (x `div` y)
```

Ici, `Left` est utilisé pour représenter une erreur avec un message, tandis que `Right` représente une valeur valide.

5.4 Exercices

On aimerait écrire une fonction find qui recherche un élément dans une liste et retourne un `Maybe` de l'élément trouvé.

```
find :: (a -> Bool) -> [a] -> Maybe a

find _ [] = Nothing
find p (x:xs)
  | p x      = Just x
  | otherwise = find p xs
```

6 Records

Les records en Haskell sont une extension des types de données algébriques qui permettent de nommer les champs d'un constructeur de données. Cela améliore la lisibilité et facilite l'accès aux champs.

Une manière de résoudre ce problème est d'utiliser des `accessor functions`.

```
data Config = Config
  String -- User name
  String -- Local host
  String -- Remote host
  Bool -- Is guest?
  Bool -- Is superuser?
  String -- Current directory
  String -- Home directory
  Integer -- Time connected

  getUserName (Config name _ _ _ _ _ _ _) = name
  getLocalHost (Config _ localHost _ _ _ _ _ _) = localHost
  ...
```

Grâce aux records nous pouvons résoudre le problème de l'accès aux champs de manière plus élégante et lisible. Au lieu d'utiliser des fonctions d'accès séparées, nous pouvons directement accéder aux champs par leur nom.

```
data Point = Point { x :: Float, y :: Float } deriving (Show)
a = Point 3 4
b = a { x = 5 }  -- Mise à jour du champ x
> b
Point {x = 5.0, y = 4.0}
```

6.1 Records pattern matching

Le pattern matching avec les records permet d'extraire facilement les champs nommés d'un record.

```
getX :: Point -> Float
getX (Point { x = xCoord }) = xCoord
getY :: Point -> Float
getY (Point { y = yCoord }) = yCoord

> getX a
5.0
> getY a
4.0
```

6.2 Exercices

On souhaite écrire une fonction qui permet de checker si un langage de programmation est fonctionnel ou pas. On aimerait définir un enum `Paradigm` et un record `Language`.

```
data Paradigm = Functional | Imperative | Object

data Language = Language {
   name :: String,
   paradigm :: Paradigm
 }

isFunctional :: Language -> Bool
isFunctional Language { paradigm = Functional } = True
isFunctional _ = False
```

7 Classes de types et instances

Les classes de types en Haskell permettent de définir des interfaces génériques que les types peuvent implémenter. Une classe de types est une collection de fonctions qui peuvent être utilisées avec différents types de données.

7.1 Deriving

Haskell permet de dériver automatiquement des instances de certaines classes de types pour les types de données algébriques en utilisant le mot-clé `deriving`.

```
data Shape = Circle Float | Rectangle Float Float deriving (Show, Eq)
```

Ici, `Shape` dérive automatiquement des instances des classes de types `Show` et `Eq`, ce qui permet d'afficher les formes et de les comparer pour l'égalité.

7.2 Heritage multiple

Un type de données peut dériver des instances de plusieurs classes de types en même temps en les séparant par des virgules dans la clause `deriving`.

7.3 Exercices

Nous devons écrire une classe de type `Comparable` qui offre une méthode `cmp` pour comparer deux valeurs d'un type donné.

```
class Comparable a where
  cmp :: a -> a -> Int

data Bit = Zero | One deriving Show

instance Comparable Bit where
  cmp One Zero = 1
  cmp Zero One = -1
  cmp _ _ = 0
```