Programmation fonctionnelle (Haskell)

Wim Vanhoof

2 novembre 2004

1 Introduction

Ce document comprend un résumé des notions de base au dessous de la programmation fonctionnelle en utilisant le langage Haskell. Le texte est basé sur *Introduction to Functional Programming using Haskell* de Richard Bird.

1.1 Le modèle de calcul

L'exécution d'un programme fonctionnel comprend l'évaluation d'une expression. Souvent, il y a un environnement interactif qui permet l'évaluation d'une expression entrée par l'utilisateur (comme c'est le cas avec une machine à calculer). On appelle *session* le cycle continu d'entrer une expression et de la laisser évaluer. Les expressions contiennent généralement des références aux fonctions pré-définies dans une bibliothèque ou définies par le programmeur dans un *script*.

Une fonction est définie par une équation entre deux expressions et elle est accompagnée d'une signature qui représente le *type* de la fonction.

```
square :: Integer -> Integer
square x = x * x

smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
```

L'évaluation d'une expression comprend la simplification de l'expression à sa plus simple forme équivalente. Une expression qui ne peut plus être simplifiée est dite d'être en forme canonique. Le processus de simplification comprend l'usage des définitions comme règles de ré-écriture : si l'expression à simplifier est une instance de la partie gauche d'une définition (= pattern matching), elle simplifie à la partie droite de la règle sous la substitution appropriée. On appelle *réduction* une séquence d'expressions $e_1, \ldots, e_n$ dont chaque expression e_i $(i \geq 2)$ est une simplification de e_{i-1} obtenue en appliquant une seule définition. Une réduction $e_1, \ldots, e_n$ est dite *totale* quand e_n est en forme canonique. En général, on peut construire plusieurs réductions pour une seule expression.

$$
\begin{array}{c}
\text{square } (3\ +\ 4) \\
= \\
\text{square } 7 \\
= \\
7 * 7 \\
= \\
49
\end{array}
\qquad\qquad
\begin{array}{c}
\text{square } (3\ +\ 4) \\
= \\
(3+4) * (3+4) \\
= \\
7 * (3+4) \\
= \\
7 * 7 \\
= \\
49
\end{array}
$$

Une caractéristique-clef de la programmation fonctionnelle est la suivante : si on peut construire des différentes réductions totales pour une seule expression, elles résultent toutes à une même forme canonique. Autrement dit, quelque soit la façon d'évaluer une expression, le résultat est toujours le même. Cependant, il est possible qu'une réduction ne se termine pas.

On appelle *redex* une (sous-)expression que l'on peut évaluer (redex = reducible expression). On appelle *redex intérieur* (= innermost redex) un redex quine contient aucun autre redex. On appelle *redex extérieur* (= outermost redex) un redex qui n'est compris dans aucun autre redex. Les stratégies d'évaluation les plus communes sont :

- la réduction intérieure (= innermost reduction) : à chaque étappe dans l'évaluation on simplifie un redex intérieur.
- la réduction extérieure (= outermost reduction) : à chaque étappe dans l'évaluation on simplifie le redex extérieur.

Ces stratégies d'évaluation ont des caractéristiques différentes. La réduction extérieure garantit de constuire une réduction total qui résulte à la forme canonique de l'expression, si celle-ci existe. En revanche, la réduction intérieure ne garantit pas de trouver la représentation canonique mais quand elle la trouve, elle a parfois besoin de moins d'étappes. Haskell utilise la réduction extérieure en combinaison avec une représentation particulière des expressions qui garantit que chaque sous-expression doublée n'est évaluée qu'une seule fois. On appelle cette stratégie d'évaluation l'évaluation paresseuse (= lazy evaluation). On utilise souvent les mots évaluation, simplification et réduction comme synonymes.

1.2 Les valeurs

Une expression représente toujours une seule valeur mais chaque valeur peut être représentée par plusieurs expressions différentes. Chaque valeur appartient à un type. Haskell comprend quelques types prédéfinis comme **Int**, **Integer**, **Float**, **Double**, **Bool** et **Char** mais le langage permet la construction de nouveaux types. Une façon de créer de nouveaux types est de grouper des types existantes : soient A et B deux types, le type (A,B) représente l'ensemble de pairs d'un élément de A et d'un élément de B. Ainsi, le type A −> B représente l'ensemble de fonctions de A vers B. On observe donc qu'une fonction est considérée comme une valeur.

L'univers des valeurs contient une valeur spéciale : $\perp$. La valeur $\perp$ représente une valeur indéfinie, un exemple étant la valeur $\frac{1}{0}$. Il est clair que la valeur $\perp$ n'est pas toujours calculable par un programme.

Une fonction f du type A−>B associe à chaque valeur de A une valeur unique de B. On dit que f prend un argument en A et retourne un résultat en B. Si x représente un élément de A, fx représente la valeur correspondante en B, c'est à dire le résultat d'appliquer la fonction f à x. On considère deux fonctions f et g égale quand $fx = gx$ pour chaque x (le principe d'existentionalité). On appelle une fonction f *stricte* si $f\perp = \perp$.

On appelle *curifier* l'action de remplacer un argument structuré par une séquence d'arguments plus simples. La version curifiée de la fonction smaller définie ci-dessus est

```
smallerc :: Integer -> (Integer -> Integer)
smallerc x y = if x <= y  then x else y
```

Suite à sa signature, smallerc est une fonction qui prend une valeur entière, x, comme argument et qui retourne une fonction (smallerc x). Cette fonction (smallerc x) est du type **Integer** −> **Integer** ; elle prend un argument y et elle retourne une valeur entière, c'est à dire le minimum de x et y. On observe que l'application de fonction s'associe à gauche. Ainsi, l'expression

smallerc 3 4

est une abbreviation pour

(smallerc 3) 4.

Par conséquent, les parenthèses dans la signature peuvent être éliminées et la signature de smallerc peut être écrit comme

```
smallerc :: Integer -> Integer -> Integer
```

Les avantages principaux de curification sont :

- la curification aide à diminuer le nombre de paranthèses nécessaires dans une expression, et
- les arguments d'une fonction curifiées sont attribués l'un après l'autre. Les résultats intermédiares sont eux-mêmes des fonctions parfois utiles. Considérons la fonction

  ```
  twice :: (Integer -> Integer) -> Integer -> Integer
  twice f x = f (f x)
  ```

  La fonction twice a deux arguments : une fonction **Integer** −> **Integer** et une valeur entière. Quand on applique la fonction sur son premier argument, ce qui reste est une fonction **Integer** −> **Integer**. Ainsi, l'expression twice square représente une fonction qui prend un entier, x, et qui retourne x^4.

Un opérateur est une fonction binaire que l'on écrit entre ses (deux) arguments. On écrit par exemple 3 + 4 et 3 <= 4 au lieu de + 3 4 et <= 3 4. Pourtant, il est parfois utile de considérer un opérateur comme une fonction

régulière. On obtient un tel effet en entourant l'opérateur par des parenthèses. Ainsi, (+) représente une fonction binaire qui prend deux arguments et retourne leur somme. De façon similaire, on peut transformer une fonction binaire en opérateur, en entourant le nom de la fonction par des guillemets inversées. Ainsi, on utilise souvent les fonctions **div** et **mod** comme opérateurs et on écrit 14 `` `div` `` 3 au lieu de **div** 14 3.

On peut également transformer un opérateur en fonction unaire, en incluant un de ses arguments entre les parenthèses. On a par exemple

$(* \ 2)$ ou $(2 \ *)$	la fonction "double"
(> 0)	le test "positif"
$(1/)$	la réciproque
$(/2)$	la fonction "demi"
$(+1)$	la fonction "successeur"

Les opérateurs arithmétiques sont évalués selon leur priorités normaux, c'est à dire la multiplication est supérieure par rapport à l'addition, etc. Il faut se rendre compte qu'appliquer une fonction a la plus grande priorité. Ainsi, l'expression square 3 + 4 est évaluée comme (square 3) + 4 ! Il est évident que l'on peut utiliser des parenthèses pour changer l'ordre dans lequel les opérateurs sont évalués.

Soient f une fonction B -> C et g une fonction A -> B. On peut *composer* f et g en utilisant l'opérateur '.' (dot). La composition $f.g$ représente une fonction A -> C. Ainsi, l'expression (f.g) x est équivalent à f (g x). Reconsidérons la fonction square définie ci-dessus. La fonction square.square est équivalente à une fonction

```
quad :: Integer -> Integer
quad x = square (square x)
```

La composition de fonctions est une opération associative : on a $(f.g).h = f.(g.h)$ pour toutes les fonctions f, g et h de types compatibles.

1.3 La définition de fonctions

Reconsidérons la définition de la fonction smaller. Une définition équivalente est

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y)
    | x <= y  =  x
    | x > y   =  y
```

Cette définition utilise des équations gardées (= guarded equations). Une telle définition comprend plusieurs *clauses*, séparées par une barre verticale. Chaque clause contient une condition et une expression ; les deux parties sont séparées par le symbole "=". Il est évident qu'une fonction peut être récursive :

```
fact :: Integer -> Integer
fact n
    | n == 0  =  1
    | n > 0   = n * fact (n - 1)
```

Il est souvent utile d'introduire une définition locale à une autre définition. A cet effet, on utilise le mot clef "where".

```
f :: (Float,Float) -> Float
f (x,y) = (a + 1) * (b + 2)
          where a = (x + y)/2
                b = (x + y)/3
```

On observe que chaque définition locale est placée sur une autre ligne. On peut les écrire sur une même ligne en les séparant par le symbole ";". Quand on utilise une définition locale en combinaison avec des expressions gardées, elle comprend toutes les expressions gardées :

```
f :: Integer -> Integer -> Integer
f x y
  | x <= 10   =  x + a
  | x > 10    =  x - a
  where a = square (y + 1)
```

1.4 Les types

Haskel est un langage fortement typé (= strongly typed), ce qui implique que l'évaluateur ne veut évaluer que des expressions bien formées. Une expression est bien formée quand on peut déterminer son type en ne considérant que les types des sous-expressions et les signatures des opérations utilisées. Par exemple l'expression

quad (3 + 4)

est bien formée étant données les signatures

```
(+) :: Integer -> Integer -> Integer
quad :: Integer -> Integer
```

et le fait que les constantes 3 et 4 appartiennent au type **Integer**. De même, l'expression quad (quad 3) est bien formée mais quad quad 3 n'est pas bien formée.

Le système de types exploité par Haskell est *polymorphe* et permet l'usage de variables dans les signatures des fonctions. Un exemple est la signature de la fonction (.) :

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Cette signature ne spécifie pas les types concrets des arguments et du résultat. L'occurrence d'une variable dans la signature peut être interprêté comme n'importe quelle type, mais il est clair que chaque occurrence d'une même variable doit être interprêtée par le même type. Ainsi, la signature de ci-dessus spécifie que (.) peut être appellée avec n'importe quelle fonctions unaires comme paramètres autant que le résultat de la fonction donnée comme deuxième argument et la source de la fonction donnée comme premier argument ont le même type (b dans la signature). Elle spécifie également que le résultat de (.) est une fonction

dont la source a le même type que la source du deuxième argument (a) et dont le résultat a le même type que le résultat du premier argument (c).

L'usage de signatures polymorphes permet d'utiliser une même fonction dans des différents contexts. Par exemple la signature polymorphe de (.) permet la construction des expressions bien formées square.square et sqroot.square où

```
square :: Int -> Int
sqroot :: Int -> Float
```

Sans polymorphisme, on devrait typer (.) de plusieurs façons différents :

```
(.) :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
(.) :: (Int -> Float) -> (Int -> Int) -> (Int -> Float)
```

Parfois une signature polymorphe est trop générale. Considérons la fonction (+). Il est clair qu'on veut utiliser (+) dans des différents contexts comme par exemple

```
(+) :: Int -> Int -> Int
(+) :: Integer -> Integer -> Integer
(+) :: Float -> Float -> Float
```

etc. Pourtant la solution polymorphe

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

est trop générale car elle permet de considérer l'addition de deux termes de n'importe quel type comme expression bien formée. Intuitivement, il est clair qu'on veut limiter l'application de (+) aux valeurs numériques. La solution est de limiter les variables dans une signature polymorphe aux types qui appartiennent à une certaine classe de types. Par exemple la signature correcte de (+) est

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

Cette signature limite la variable a dans la signature a −> a −> a aux types qui appartiennent à la classe de types (= typeclass) **Num** qui représente les types numériques. En Haskell ils existent plusieurs classes de types prédéfinies et le langage permet de définir de nouveaux classes.

2 Les types simples

La définition d'un type comprend la description d'un ensemble de valeurs éventuellement infini. Dans ce texte, nous verrons comment les types de base sont définis en Haskell. Il est évident que ces techniques de définition peuvent être utilisées pour introduire de nouveaux types.

2.1 Définition par énumération

Une première méthode pour définir un type dont l'ensemble de valeurs est fini, est d'énumérer explicitement tous les éléments appartenant à ce type. Le

type **Bool** – comprenant les valeurs logiques – et le type **Char** – comprenant les caractères – sont tous les deux défini par l'énumération explicite de leurs éléments.

2.1.1 Le type booléen

Le type booléen est défini en Haskell comme l'ensemble comprenant les valeurs **True** et **False** :

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

On observe l'usage des majuscules dans le nom du type et de ses valeurs. Ensuite, on peut définir des fonctions qui prennent un argument booléen utilisant "pattern matching" comme dans l'exemple suivant :

```
not :: Bool -> Bool
not True = False
not False = True
```

Afin de simplifier une expression de la forme **not** e, les équations sont utilisées comme des règles de ré-écriture. D'abord, e est simplifiée à sa forme canonique. Si ce processus se termine et résulte à **True**, l'évaluateur appliquera la première règle. Si le résultat est **False**, la deuxième règle est utilisée. Si e ne peut pas être réduit à une forme canonique, la valeur de **not** e est indéfinie. Alors **not** $\perp$ = $\perp$ et **not** est donc une fonction stricte. On pourrait dire qu'il y a donc trois valeurs booléennes : **True**, **False** et $\perp$. En fait, chaque type en Haskell inclut une valeur 'anonyme' : $\perp$. Les opérations logiques $\wedge$ et $\vee$ sont définies de façon similaire :

```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool

False && x = False
True && x = x

False || x = x
True || x = True
```

Ces définitions utilisent pattern matching sur l'argument gauche. Pour simplifier une expression de la forme $e_1 \&\& e_2$, l'évaluateur simplifie d'abord e_1 à sa forme canonique (si possible). Si le résultat est **False**, l'évaluateur emploiera la première règle et retourna **False** comme résultat (e_2 n'est donc pas évaluée). Si par contre la valeur de e_1 est **True**, l'évaluateur emploiera la deuxième règle et retourera la valeur de e_2 (ce qui nécessite donc l'évaluation de e_2 à sa forme canonique). On a donc

$$\perp \,\&\&\, \textbf{False} = \perp$$
$$\textbf{False} \,\&\&\, \perp = \textbf{False}$$
$$\textbf{True} \,\&\&\, \perp = \perp$$

Autrement dit, (&&) est stricte dans son argument gauche, mais ne pas dans son argument droite. Une autre définition de (&&) serait

```
False && False = False
False && True = False
True && False = False
True && True = True
```

Cette définition utilise pattern matching sur les deux arguments ; elle est donc stricte dans ses deux arguments.

On peut introduire les opérateurs de comparaison : (==) et (/=). Ces opérateurs seront redéfinis pour chaque type rencontré. Alors il faut mieux introduire une classe de types.

```
class Eq a where
  (==), (/=) :: a -> a-> Bool
```

La classe de types **Eq** comprend donc deux fonctions (parfois appelées méthodes) dont la signature est

```
(==), (/=) :: Eq a => a -> a-> Bool
```

Ensuite on déclare **Bool** une instance de cette classe en fournissant les définitions concrètes des méthodes (==) et (/=) dans le context **Bool**.[1]

```
instance Eq Bool where
  x == y  =  (x && y) || (not x && not y)
  x /= y  =  not (x == y)
```

De façon similaire, on introduit les opérateurs de comparaison (<), (<=), (>=) et (>) qui seront définis pour chaque type d'une classe **Ord** qui ressemble les types dont les éléments sont ordonnés.

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  (x <= y)  =  (x < y) || (x == y)
  (x >= y)  =  (x > y) || (x == y)
  (x > y)   =  not (x <= y)
```

On observe la restriction de la classe **ord** aux types qui appartiennent à la classe **Eq**. Autrement dit, **Ord** est définie comme une sous-classe de eq. La classe **Ord** comprend 4 opérations dont 3 (les opérations (<=), (<=) et (>)) sont définies par la classe elle-même en fonction de la quatrième opération (<) et de l'opération (==) définie dans la classe **Eq**. En déclarant un type instance de **Ord**, il suffit donc de fournir une définition de (>). Dans l'exemple de **Bool** :

```
instance Ord Bool where
  False < False = False
  False < True = True
  True < False = False
  True < True = False
```

Les définitions de (<=), (<=) et (>) sont donc reprises de la classe.

[1] On observe la différence entre les symboles == (comparaison) et = (définition).

2.1.2 Le type des caractères

Le type **Char** est prédéfini en Haskell et comprend les 256 symboles de la table ASCII. Ils incluent les symboles alphanumériques ainsi que quelques caractères de contrôle comme le retour à la ligne etc. Dans leur forme canonique, les caractères sont représentés en entourant le symbole concerné par des guillemets, par exemple 'a', '7', etc. Une espace est représentée par ' '. Les fonctions suivantes sont prédéfinies :

```
ord :: Char -> Int
chr :: Int -> Char
```

Elles établissent la correspondance entre un caractère et sa rangée dans la table ASCII. Les caractères peuvent être comparés et testés sur (in)égalité. On a

```
instance Eq Char where
  (x == y) = (ord x == ord y)

instance Ord Char where
  (x < y) = (ord x < ord y)
```

Dans cette définition, on utilise le fait que **Int** est il-même une instance de **Eq** et de **Ord**.

2.1.3 Autres énumérations

On peut utiliser le technique d'énumération pour introduire de nouveaux types.

```
data Jour = Dim | Lun | Mar | Mer | Jeu | Ven | Sam
```

Cette définition introduit le type Jour comprenant 8 éléments : 7 valeurs qui sont représentées par les constantes Dim, Lun, etc. et la valeur indéfinie $\perp$.

En général on veut que les éléments d'une énumération soient comparables. Afin de déclarer le nouveau type instance de **Eq** et de **Ord**, il est préférable d'établir une correspondance une-à-une entre les constantes d'une énumération et un ensemble d'entiers. A cet effet on introduit une classe de types dont les éléments sont énumérables :

```
class Enum a where
  toEnum   :: a -> Int
  fromEnum :: Int -> a
```

La correspondance entre les constantes d'une énumération d'une part et un ensemble d'entiers d'autre part est établit en déclarant le type d'énumération une instance de **Enum**, ce qui nécessite de fournir des définitions concrètes de **toEnum** et **fromEnum**. Il est clair que ces définitions doivent être telles que **fromEnum** (**toEnum** x) = x pour chaque x mais on ne peut pas exprimer cette exigence en Haskell. On observe que le même principe d'associer une valeur entière unique à chaque élément d'une énumération était présent dans la définition du type **Char**. En effait, on peut déclarer

```
instance Enum Char where
  toEnum   = ord
  fromEnum = chr
```

Supposons la définition suivante pour le type Jour :

```
instance Enum Jour where
  toEnum Dim = 0
  toEnum Lun = 1
  toEnum Mar = 2
  toEnum Mer = 3
  toEnum Jeu = 4
  toEnum Ven = 5
  toEnum Sam = 6
```

Ensuite, on peut déclarer

```
instance Eq Jour where
  (x == y)  =  (toEnum x == toEnum y)

instance Ord Jour where
  (x < y)  =  (toEnum x < toEnum y)
```

Haskel permet, pour n'importe quel type défini par énumération, de dériver automatiquement les définitions qui font le type une instance de **Eq**, de **Ord** et de **Enum**. Il suffit d'introduire une clause "deriving" comme dans l'exemple suivant :

```
data Jour = Dim | Lun | Mar | Mer | Jeu | Ven | Sam
            deriving (Eq, Ord, Enum)
```

2.2 La définition par constructeurs

2.2.1 Les tuples

Le type polymorphe Pair a b représente l'ensemble de toutes les paires qui comprennent une valeur de a et une de b.

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

Ce type est déclaré par une fonction dite *constructeur* MkPair dont la signature

```
MkPair :: a -> b -> Pair a b
```

En général, un constructeur est une fonction ayant les propriétés suivantes :

- elle n'a pas de définition. Son seul effet est de construire une valeur. Dans l'exemple ci-dessus MkPair **True** 'b' est une valeur (du type Pair **Bool Char** en forme canonique.
- elle peut être utilisée dans la partie gauche d'une définition (afin de réaliser pattern matching).

Considérons les fonctions de base

```
fst :: Pair a b -> a
fst MkPair x y = x

snd :: Pair a b -> b
snd MkPair x y = y
```

Haskell permet d'utiliser une syntaxe alternative pour les paires. Le type Pair a b est représenté par (a,b) et une valeur Mkpair e_1, e_2 est représentée par (e_1, e_2). Les fonctions de base deviennent donc

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

On observe l'usage de pattern matching dans la définition de ces fonctions. Afin d'évaluer un expression de la forme **fst** e, l'évaluateur simplifie d'abord e jusqu'à une forme (e_1, e_2) et retourne ensuite la sous-expression e_1. N'oublions pas que le type (a,b) inclut la valeur indéfinie $\perp$. Pourtant, $\perp \neq (\perp, \perp)$. En effet, les constructeurs sont des fonctions non-strictes.

Les éléments d'un type (a,b) peuvent être comparés si les éléments de a et de b peuvent être comparés eux-mêmes :

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (u,v) = (x == u) && (y == v)

instance (Ord a,Ord b) => Ord (a,b) where
  (x,y) < (u,v) = (x < u) || (x == u && y < v)
```

On apelle cette ordre l'ordre lexicographique. Les tuples ne sont pas limités à deux valeurs ; on peut également utiliser les triplets (a,b,c), les 4-tuples (a,b,c,d), etc.

2.2.2 Autres types

En général un type est défini par une énumération de constructeurs.

```
data Either = Left Bool | Right Char
```

Le type **Either** comprend les valeurs construites par les constructeurs **Left** et **Right** dont les signatures

n

On peut généraliser ce type à un type polymorphe

```
data Either a b = Left a | Right b
```

Dans ce cas, les constructeurs sont

```
Left :: a -> Either a b
Right :: b -> Either a b
```

A nouveau, les constructeurs peuvent être utilisés pour pattern matching

```
case :: (a -> c, b -> c) -> Either -> c
case (f,g) (Left x)  =  f x
case (f,g) (Right y) =  g y
```

En supposant que les valeurs de a et de b peuvent être comparées, on peut définir la comparaison des éléments du type **Either** a b :

```
instance (Eq a, Eq b) => Eq (Either a b) where
  Left x == Left y    =  (x == y)
  Left x == Right y   =  False
  Right x == Left y   =  False
  Right x == Right y  =  (x == y)

instance (Ord a, Ord b) => Ord (Either a b) where
  Left x < Left y    =  (x < y)
  Left x < Right y   =  True
  Right x < Left y   =  False
  Right x < Right y  =  (x < y)
```

Ces déclarations sont obtenues de façon automatique par la déclaration

```
data Either a b = Left a | Right b
                  deriving (Eq,Ord)
```

2.3 La définition par équivalence

Il est facile d'introduire un synonyme pour un type

```
type Angle = Float
type Position = (Float,Float)
```

Les synonymes peuvent être polymorphes eux-mêmes :

```
type Pairs a = (a,a)
type Automorphisme a = a -> a
type Flag a = (a,Bool)
```

Un synonyme peut être défini en fonction d'un autre synonyme tant qu'il n'y a pas de circularités dans les définitions. Chaque synonyme d'un type hérite les méthodes qui ont été déclarées dans les instances du type en-dessous. Ces méthodes ne peuvent pas être redéfinies.

Au lieu de créer un synonyme, on peut introduire un nouveau type en emballant un autre type par un constructeur :

```
data Angle = MkAngle Float

instance Eq Angle where
  MkAngle x == MkAngle y  =  normalise x == normalise y
```

```
normalise :: Float -> Float
normalise x
  | x < 0      =  normalise (x + rot)
  | x >= rot   =  normalise (x - rot)
  | otherwise =  x
  where rot = 2*3.14159
```

On observe que les éléments de Angle sont “emballés” (= wrapped) par le constructeur MkAngle. Par conséquant, les actions de construire et d’examiner les valeurs du type Angle nécessitent de emballer et de déballer continuellement ces valeurs. En plus, MkAngle $\perp \neq \perp$ et donc le type Angle n’est pas isomorphe au type **Float**.

Haskell permet pourtant de créer un type isomorphe à un autre type en utilisant la déclaration “newtype” :

```
newtype Angle = MkAngle Float
```

Les constructeurs utilisés dans une déclaration “newtype” sont des constructeurs strictes.

2.4 Les types numériques prédéfinis

Haskell connaît les types numériques suivants :

- **Int** : Les entiers (représentation limitée)
- **Integer** : les entiers (représentation illimitée)
- **Float** : nombres à virgule flottante (à simple précision)
- **Double** : nombres à virgule flottante (à double précision)

Il est évident que le calcul utilisant **Integer** est plus lent que le calcul utilisant **Int**. En revanche, le calcul utilisant **Integer** est exact. Les opérations $(+)$, $(-)$ et $(*)$ sont définies pour chacun de ces types. Les opérations **div** et **mod** sont définies pour les types **Int** et **Integer**. Si $y \neq 0$, x ‘**div**‘ y=x/y, c’est à dire le plus grand entier $\leq x/y$. La valeur x ‘**mod**‘ y est définie par l’équation x = (x ‘**div**‘ y)*y + (x ‘**mod**‘ y). Pour y positive, on a 0 $\leq$x ‘**mod**‘ y$< y$; pour y négative, on a $y <$x ‘**mod**‘ y≤ 0. L’opération $(/)$ est définie pour les types **Float** et **Double**.

3 Les types récursifs

3.1 Les nombres naturels

On pourrait introduire les nombres naturels par la définition récursive suivante

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

On a donc

$$\text{Zero} \in \text{Nat et Succ n} \in \text{Nat si n} \in \text{Nat}$$

c’est à dire Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero)), etc. sont tous des éléments de Nat. Le constructeur Succ (dite “successeur”) a la signature

```
Succ :: Nat -> Nat
```

On pourrait définir l'addition par pattern matching :

```
(+) :: Nat -> Nat -> Nat
m + Zero   = m
m + Succ n = Succ (m + n)
```

Etant donné l'addition, on pourrait d'abord définir la multiplication et ensuite l'exposant :

```
(*) :: Nat -> Nat -> Nat
m * Zero   = Zero
m * Succ n = (m * n) + m

(^) :: Nat -> Nat -> Nat
m ^ Zero   = Succ Zero
m ^ Succ n = (m ^ n) * m
```

On peut introduire les opérateurs de comparaison en déclarant Nat une instance de **Eq** et de **Ord** :

```
instance Eq Nat where
  Zero == Zero     =  True
  Zero == Succ n   = False
  Succ m == Zero   = False
  Succ m == Succ n =  (m == n)

instance Ord Nat where
  Zero < Zero     = False
  Zero < Succ n   = True
  Succ m < Zero   = False
  Succ m < Succ n =  (m < n)
```

Alternativement, il suffit de déclarer

```
data Nat = Zero | Succ Nat
           deriving (Eq,Ord)
```

Il reste la définition de la soustraction :

```
(-) :: Nat -> Nat -> Nat
m - Zero         =  m
Succ m - Succ n  =  m - n
```

Cette définition utilise pattern matching sur les deux arguments. La fonction est partielle. On a par exemple (Zero − Succ Zero) = ⊥.

N'oublions pas que ⊥ ∈Nat et, par conséquant, Succ ⊥, Succ (Succ⊥), etc. appartiennent toutes à Nat. Elles représentent différentes valeurs indéfinies. Nat contient encore une autre valeur différente de toutes les autres, c'est à dire la valeur "infinie" que l'on pourrait définir comme

```
infini :: Nat
infini = Succ infini
```

En résumé, on observe donc que les valeurs de Nat sont partagées en 3 classes :
- les nombres finis ; càd. ceux qui correspondent aux nombres naturels ;
- les nombres partiels : $\perp$, Succ$\perp$, Succ (Succ$\perp$),etc. ; et
- les nombres infinis dont il n'y a qu'un seul.

Comme nous le verrons plus tard, cette classification est typique pour tous les types récursifs. La plupart des fonctions définies sur le type Nat ressemble le schéma suivant :

```
f :: Nat -> a
f Zero     = c
f (Succ n) = h (f n)
```

où a représente un type quelconque, c un élément de a et h :: a −> a. On observe qu'un appel f e remplace Zero par c et Succ par h dans e. On peut encore généraliser ce schéma en introduisant h et c comme paramètres et on obtient la fonction

```
foldn :: (a -> a) -> a -> Nat -> a
foldn h c Zero     =  c
foldn h c (Succ n) =  h (foldn h c n)
```

Les fonctions (+), (∗) et (ˆ) peuvent toutes être définies en fonction de foldn :

```
m + n  =  foldn Succ m n
m * n  =  foldn (+m) Zero n
m ^ n  =  foldn (*m) (Succ Zero) n
```

On peut utiliser foldn avec d'autres types que Nat.

```
fact :: Nat -> Nat
fact = snd . foldn f (Zero, Succ Zero)
       where f (m, n) = (Succ m, Succ m * n)

fib :: Nat -> Nat
fib = fst . foldn g (Zero, Succ Zero)
      where g (m, n) = (n, m + n)
```

L'usage de foldn permet de définir de façon concise un nombre de définitions récursives. En plus, les propriétés de foldn peuvent être utilisées afin de prouver des propriétés d'une instance particulière.

3.2 Les listes

Le type **List** a est défini comme

```
data List a = Nil | Cons a (List a)
```

Autrement dit, une liste est soit la liste vide, représentée par Nil, soit une liste crée en ajoutant un élément (de a) à la tête d'une autre liste (de type **List** a). Tous les éléments d'une liste sont donc toujours d'un même type.

En Haskell, on utilise les abbreviations suivantes. Le type **List** a est abrégé comme [a], la constante Nil est écrit comme [] et le constructeur Cons est représenté par l'opérateur (:). De plus, (:) s'associe à droite et on a donc

$$\begin{gathered}\text{Cons 1 (Cons 2 (Cons 3 Nil))}\\ =\\ 1:(2:(3:[]))\\ =\\ 1:2:3:[]\end{gathered}$$

Ce dernier terme est à son tour abrégé par [1,2,3] . Comme toujours, on emploie pattern matching pour définir une fonction sur les listes.

```
instance (Eq a) => Eq [a] where
  [] == []         = True
  [] == (y:ys)     =  False
  (x:xs) == []     = False
  (x:xs) == (y:ys) =  (x == y) && (xs == ys)
```

Le test si une liste est vide est réalisé par

```
null :: [a] -> Bool
null []     = True
null (x:xs) = False
```

L'enchaînement de deux listes :

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

On a par exemple [1,2,3] ++ [4,5] = [1,2,3,4,5] et [1,2] ++ [] ++ [1] = [1,2,1]. On observe que $\bot$++ ys = $\bot$ mais xs ++$\bot$ $\neq$ $\bot$. L'enchaînement est une opération associative.
L'opération "flatten" transforme une liste de listes dans une seule liste en enchaînant les éléments, par exemple flatten [[1,2],[],[3,2,1]] = [1,2,3,2,1] .

```
flatten :: [[a]] -> [a]
flatten []       = []
flatten (xs:xss) = xs ++ flatten xss
```

Une liste est enversée par la fonction **reverse**. Ainsi, **reverse** [1,2,3,4] = [4,3,2,1] .

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Une version alternative utilisant un accumulateur :

```
reverse :: [a] -> [a]
reverse xs = revacc xs []

revacc :: [a] -> [a] -> [a]
revacc [] as     = as
revacc (x:xs) as = revacc xs (x:as)
```

La longeur d'une liste est déterminée par la fonction suivante

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Les listes finies sont les seules à avoir une longeur déterminée. Les listes partielles comme $\bot$, x:$\bot$, x:y:$\bot$, etc. on une longeur indéfinie. Les fonctions suivantes prennent respectivement le premier élément et le reste d'une liste.

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Similairement, on peut définir deux fonctions qui prennent respectivement le dernier élément et toute la liste sauf ce dernier élément :

```
last :: [a] -> a
last = head . reverse

init :: [a] -> [a]
init = reverse . tail . reverse
```

Avec cette définition, on a **init** xs = $\bot$ pour toute liste xs partielle ou infinie. Une définition supérieure serait

```
init (x:xs) = if null xs then [] else x:init xs
```

avec laquelle **init** xs = xs pour toute liste xs infinie. La fonction **take** n xs prend les n premiers éléments d'une liste xs ; la fonction **drop** n xs prend tous les éléments sauf les n premiers.

```
take :: Int -> [a] -> [a]
take 0 xs         = []
take (n + 1) []   = []
take (n + 1) (x:xs) = x:take n xs

drop :: Int -> [a] -> [a]
drop 0 xs         = xs
drop (n + 1) []   = []
drop (n + 1) (x:xs) = drop n xs
```

On observe l'usage de pattern matching sur les nombres entiers. En Haskell, pendant le pattern matching les clauses sont considérées dans leur ordre naturel dans le programme et les arguments d'une clause particulière sont considérés de gauche à droite. La première clause dont tous les arguments sont "matched" est prise par l'évaluateur. Par conséquant, **take** $0\bot$ = [] et **take**$\bot$ [] =$\bot$. Concluons par l'opération "indice" (= index) qui sélectionne un élément particulier d'une liste :

```
(!!) :: [a] -> Int -> a
(x:xs)!!0        = x
(x:xs)!!(n + 1) = xs!!n
```

3.3 Le type String

Le type **String** est un synonyme pour une liste de caractères :

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

Pourtant il existe une syntaxe spéciale pour représenter des valeurs type string : les caractères concernés sont mis en séquence et entourés par des doubles guillemets. Ainsi, on a par exemple "Hello World" =

['H','e','l','l','o',' ','W','o','r','l','d'].

4 Travailler avec des listes

L'opération **map** applique une fonction donnée à tous les éléments d'une liste. Par exemple

```
map square [9,3] = [81,9]
map (<3) [1,2,3] = [True,True,False]
map nextLetter "HAL" = "IBM" where nextLetter x = chr ((ord x) + 1)
```

La définition de **map** est

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x:map f xs
```

Comme un deuxième exemple de l'usage de map, on considère calculer la somme des 100 premiers entiers carrés, ce qui est réalisé par l'expression

sum (**map** square (upto 1 100))

avec les définitions accompagnantes

```
sum :: (Num a) => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```
upto :: (Integral a) => a -> a -> [a]
upto m n = if m > n then [] else m:upto (m + 1) n
```

Haskell connaît l'abbréviation [m..n] = upto m n. Ainsi, [m..] représente la liste infinie des entiers à partir de m.

Une deuxième fonction utile est **filter**, qui prend une fonction booléenne p ainsi qu'une liste xs et qui retourne une liste comprenante les éléments de xs qui satisfont p. Par exemple, **filter** pair [1,2,4,5,32] = [2,4,32].

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs
```

Parfois, les opérations réalisées par **map** et **filter** peuvent être décrit en utilisant une notation particulière (appellée "list comprehension" en anglais) ; par exemple

$$[x*x \mid x <- [1..5],\ \textbf{odd}\ x] = [1,9,25]$$

De façon formelle, cette notation prend la forme [e|Q] où e est une expression et Q est une séquence (éventuellement vide) de générateurs et gardiens, séparés par des virgules. Un générateur prend la forme x<− xs où x est une variable ou un tuple de variables, et xs une expression équivalente à une liste. Un gardien est une expression dont la valeur est booléenne. On définit

$$\begin{array}{lcl} [e \mid x<-xs,\ Q] & = & \text{flatten}\ (\textbf{map}\ f\ xs) \\ & & \textbf{where}\ f\ x = [e \mid Q] \\ [e \mid p,\ Q] & = & \textbf{if}\ p\ \textbf{then}\ [e \mid Q]\ \textbf{else}\ [] \end{array}$$

Quelques exemples additionnels :

$$\begin{array}{lcl} [(a,b) \mid a<-[1..3],\ b<-[1..2]] & = & [(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)] \\ [(i,j) \mid i<-[1..4],\ j<-[i+1..4]] & = & [(1,2),(1,3),(2,3),(2,4),(3,4)] \end{array}$$

On est libre d'interposer des gardiens :

$$[(i,j) \mid i<-[1..4],\ \textbf{even}\ i,\ j<-[i+1..4],\ \textbf{odd}\ j] = [(2,3)]$$

L'exemple suivant calcule tous les triplets (x,y,z) tant que $x^2 + y^2 = z^2$.

```
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z)|(x,y,z)<-triples n, pyth (x,y,z)]

triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x<-[1..n], y<-[1..n], z<-[1..n]]

pyth :: (Int,Int,Int) -> Bool
pyth (x,y,z) = (x*x + y*y == z*z)
```

La fonction **zip** prend 2 lists et retourne une liste comprenant les paires d'éléments correspondants. On a

zip [0..4] "hello" = [(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]

Quand les listes ont une longueur différente, la longueur du résultat est le minimum des longueurs concernées :

zip [0,1] "hello" = [(0,'h'),(1,'e')]

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys         = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Le pattern matching est fait d'abord sur le premier argument. On a donc **zip** $\perp$ [] = $\perp$ et **zip** [] $\perp$ = []. Considérons par exemple le produit scalaire d'un vecteur x et d'un vecteur y, défini comme sp x y = $\sum x_i * y_i$.

```
sp :: (Num a) => [a] -> [a] -> a
sp xs ys = sum (map times (zip xs ys))
           where times (x,y) = x * y
```

Pour déterminer si une séquence $[x_0, \ldots, x_{n-1}]$ est non-décroissante (càd. $x_k \leq x_{k+1}, \forall k : 0 \leq k \leq n-2$), on pourrait utiliser la fonction

```
nondec :: (Ord a) => [a] -> Bool
nondec xs = and (map leq (zip xs (tail xs)))
            where leq (x,y) = (x <= y)

and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

On observe que ces deux exemples comprennent une version décurifiée d'un opérateur bien connue. En faite, on a

times = **uncurry** (*)
leq = **uncurry** (<=)

où

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

L'inverse de la fonction **zip** est la fonction **unzip** qui prend une liste de paires et retourne une paire de deux listes.

```
unzip :: [(a,b)] -> ([a],[b])
unzip = pair (map fst , map snd)
```

On a **unzip** ([(1,**True**),(2,**True**),(3,**False**)] = ([1,2,3],[**True**,**True**,**False**]). La fonction **foldr** est similaire à la fonction foldn mais travaille sur des listes :

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

La fonction **foldr** associe les éléments d'une liste de droite à gauche :

$$\textbf{foldr}\ f\ e\ [x_1, x_2, x_3, x_4] = f(x_1, f(x_2, f(x_3, f(x_4, e)))).$$

Presque toutes les fonctions que nous avons définies sur les listes peuvent être reformulées en fonction de **foldr**.

```
flatten :: [[a]] -> [a]
flatten = foldr (++) []

reverse :: [a] -> [a]
reverse = foldr snoc []
          where snoc x xs = xs ++ [x]

length :: [a] -> Int
length = foldr uneplus 0
         where uneplus x n = 1 + n

sum :: Num a => [a] -> a
sum = foldr (+) 0

and :: [Bool] -> Bool
and = foldr (&&) True

map :: (a -> b) -> [a] -> [b]
map f = foldr (cons.f) []
        where cons x xs = x:xs

unzip :: [(a,b)] -> ([a],[b])
unzip = foldr conss ([],[])
        where conss (x,y) (xs,ys) = (x:xs,y:ys)
```

De façon similaire, on peut définir une fonction **foldl** qui associe les éléments de gauche à droite.

$$\textbf{foldl}\ f\ e\ [x_1, x_2, x_3, x_4] = f(f(f(f(e, x_1), x_2), x_3), x_4).$$

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Avec cette définition on peut par exemple réaliser une version efficace de **reverse** :

```
reverse :: [a] -> [a]
reverse = foldl rcons []
          where rcons xs x = x:xs
```

L'opération **scanl** applique une opération **foldl** à chaque segment initial d'une liste. Ainsi, **scanl** (+) 0 calcule la liste des sommes accumulées d'une liste de nombres et **scanl** (∗) 1 [1.. n] calcule la liste des n premiers factoriels. La définition de **scanl** comprend une fonction auxiliaire, qui retourne la liste de tous les segments initials d'une liste, par exemple

$$\textbf{inits}[x_0, x_1, x_2] = [[], [x_0], [x_0, x_1], [x_0, x_1, x_2]].$$

On a

```
inits :: [a] -> [[a]]
inits [] = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
```

ou alternativement

```
inits = foldr f [[]] where f x xss = []:map (x:) xss
```

L'opération **scanl** elle-même est définie comme

```
scanl :: (a -> b -> b) -> b -> [a] -> [b]
scanl f e = map (foldl f e).inits
```

L'opération duale de **scanl** est **scanr** définie par

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e = map (foldr f e).tails
```

où la fonction tails retourne les segments finals d'une liste, par exemple

$$\textbf{tails}[x_0, x_1, x_2] = [[x_0, x_1, x_2], [x_1, x_2], [x_2], []].$$

On peut définir **tails** par

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails (x:xs) = (x:xs):tails xs
```

5 Les types de données abstraits

La définition d'un *type (de données) concret* comprend la description des valeurs qui appartiennent à ce type. En revanche, la définition d'un *type de données abstrait* (dite TDA) comprend la description des opérations susceptibles aux valeurs qui appartiennent à ce type sans spécifier la représentation de ces valeurs-mêmemes. En particulier, la définition d'un TDA comprend :

– une signature : la liste des opérations et leurs types

– une spécification algébrique : une liste d'axiômes sur les résultats des opérations spécifiées.

Considérons comme exemple la spécification d'un type Queue a pour représenter une queue d'éléments de type a. L'intention d'une telle structure de données est d'insérer un élément à l'arrière de la queue et de retirer un élément à l'entête de la queue. La signature de ce type est

```
empty :: Queue a
join :: a -> Queue a -> Queue a
front :: Queue a -> a
back :: Queue a -> Queue a
isEmpty :: Queue q -> Bool
```

Les axiômes spécifient le comportement des opérations sans spécifier leur implémentation :

$$\begin{gathered}
\text{isEmpty empty} = \textbf{True} \\
\text{isEmpty } (\textbf{join } x\ q) = \textbf{False} \\
\text{front } (\textbf{join } x\ \text{empty}) = x \\
\text{front } (\textbf{join } x\ (\textbf{join } y\ xq)) = \text{front } (\textbf{join } y\ xq) \\
\text{back } (\textbf{join } x\ \text{empty}) = \text{empty} \\
\text{back } (\textbf{join } x\ (\textbf{join } y\ xq)) = \textbf{join } x\ (\text{back } (\textbf{join } y\ xq)) \\
\text{isEmpty}(\textbf{join } x\bot) = \text{isEmpty}\bot = \bot
\end{gathered}$$

Afin d'implémenter le TDA, il suffit de choisir une représentation des valeurs, d'implémenter les fonctions spécifiées et de prouver que les axiômes sont satisfaits. Une première possibilitée est de représenter une queue par une liste finie :

```
joinc :: a -> [a] -> [a]
joinc x xs = xs ++ [x]

emptyc :: [a]
emptyc = []

isEmptyc :: [a] -> Bool
isEmptyc xs = null xs

frontc :: [a] -> a
frontc (x:xs) = x

backc :: [a] -> [a]
backc (x:xs) = xs
```

Prouver cette implémentation correcte est triviale à cause de la correspondance une-à-une entre une queue et sa représentation concrète (la liste finie). Cette correspondance est mise en évidence par les fonctions suivantes :

```
abstr :: [a] -> Queue a
abstr = foldr join empty . reverse
```

```
reprn :: Queue a -> [a]
reprn empty = []
reprn (join x xq) = reprn xq ++ [x]
```

Une solution alternative est de représenter une queue xq par une paire de listes (xs,ys) tant que les éléments de xq comprennent les éléments de la liste xs++ **reverse** ys. En plus, nous imposons sur la représentation l'invariante

```
valid :: ([a],[a]) -> Bool
valid (xs,ys) = not (null xs) || null ys
```

On pourrait définir la fonction abstr comme

```
abstr :: ([a],[a]) -> Queue a
abstr (xs,ys) = (foldr join empty . reverse) (xs ++ reverse ys)
```

On observe que cette fonction n'est pas inversive : il n'y a pas de correspondance une-à-une entre une que et la représentation par paire et on ne peut donc pas définir reprn. L'implémentation

```
emptyc = ([],[])
isEmptyc (xs,ys) = null xs
joinc x (ys,zs) = mkValid (ys,x:zs)
frontc (x:xs,ys) = x
backc (x:xs,ys) = mkValid (xs,ys)

mkValid :: ([a],[a]) -> ([a],[a])
mkValid (xs,ys) = if null xs then (reverse ys,[])
                  else (xs,ys)
```

Les opérations d'un TDA peuvent être organisées dans une module :

```
module Queue (Queue, empty, isEmpty, join, front, back) where
  newtype Queue a = MkQ ([a],[a])

  empty :: Queue a
  empty = MkQ ([],[])

  isEmpty :: Queue a -> Bool
  isEmpty (MkQ (xs,ys)) = null xs

  join :: a -> Queue a -> Queue a
  join x (MkQ (ys,zs)) = mkValid (ys,x:zs)

  front :: Queue a -> a
  front (MkQ (xs,ys)) = x

  back :: Queue a -> Queue a
  back (MkQ (x:xs,ys)) = mkValid (xs,ys)
```

```
mkValid :: ([a],[a]) -> Queue a
mkValid (xs,ys) = if null xs then MkQ (reverse ys, [])
                  else MkQ (xs,ys)
```

L'entête de la module comprend le nom de la module (avec majuscule) suivi par la liste des opérations exportées par la module. On observe l'introduction d'un nouveau constructeur (MkQ) non-exportée. Ainsi on garantit que la réalisation du TDA est complètement cachée par la module.