Principes de Programmation
Fiche de Révisions:
Types Fonctionnels et Algébriques

25 janvier 2019

1 Types Fonctionnels

1.1 Différencier termes et types

En Haskell, la syntaxe des termes et celle des types sont différentes, et un même symbole/mot peut avoir un sens complètement différent au milieu d'un terme et au milieu d'un type. Il faut donc rapidement distinguer les termes des types selon les endroits où ils sont écrit. C'est pour ca que l'on vous conseil vivement d'utiliser un IDE avec de la coloration syntaxique prévu pour haskell (les types et les termes y ont des couleurs différentes).

1.2 Comme Type d'une Fonction

Exemple 1.
Le programme suivant (existant dans Haskell) prend la négation d'une valeur booléenne :

```
not :: Boolean -> Boolean
not x  = if x then False else True
```

Remarquez deux choses :

- Le test du `if` est la variable `x` elle même. En fait, en Haskell (et dans la plupart des langages typés), un `if_then_else` ne branche pas en fonction d'un test, mais en fonction de la valeur d'un Booléen ! Ici ce booléen est directement donné en entrée.
- Le type de `not` est `(Boolean -> Boolean)`, signalant que ce programme prend un unique argument qui doit être un Booléen et retourne un Booléen. Comme on va le voir, il ne s'agit pas juste d'une notation pour indiquer le type d'entrée et le type de sortie, mais vraiment du type de la fonction `not` !

1.3 Et Avec Plus d'un Argument ?

Exemple 2.
Le programme suivant (existant dans Haskell) retourne la valeur maximal entre deux entiers :

```
max :: Int -> Int -> Int
max x y = if (x >= y) then x else y
```

Il y a de nouveau deux remarques à faire :

- le type donné à `max` est cette fois `(Int -> Int -> Int)`, cela veut dire que `max` attend deux arguments de type `Int` et retourne un autre `Int`. Mais cela va plus loin, en effet par convention le type flèche est "associatif à droite", cela veut dire [1] que `(Int -> Int -> Int) = (Int -> (Int -> Int))`. De là on peut aussi dire que `max` prend un argument de type `Int` et revoit le type fonctionnel `(Int -> Int)`. Ces deux visions sont parfaitement cohérentes puisque `max` prend ses arguments l'un après l'autre, il faut donc savoir ce qu'il se passe si on ne donne qu'un argument. En l'occurrence, on pourra voir au cours des TPs que cela peut être très utile.
- Le même programme peut aussi prendre le maximum entre deux `Float` ou entre deux `String`, etc... En fait, son vrais type n'est pas `(Int -> Int -> Int)` mais un sur-type plus général que l'on décrira plus tard.

Exemple 3.
Le programme suivant (existant dans Haskell) compare deux entiers :

```
(<=) :: Int -> Int -> Boolean
```

Il y a cette fois pas mal de remarques à faire :

- Il ne s'agit pas d'une fonction, mais d'un opérateur... Ce qui n'a aucune importance en Haskell. En effet, un opérateur n'est qu'une fonction à deux arguments écrit en notation infixe. Ainsi on peut écrire `2<=3`, mais aussi `(<=) 2 3` ; c'est le même programme. Tout opérateur (comme `<=`) est une fonction lorsque écrit entre parenthèse (comme `(<=)`), et toute fonction (comme `max`) est un opérateur lorsque écrit entre *quotes* (comme `` `max` ``).
- On n'a pas écrit le code correspondant à cet opérateur. C'est parce qu'il est *hardcodé* sur les entier, c'est une brique de base que l'on ne peux pas coder sois même...
- Comme pour `max`, le même opérateur peut être utiliser sur plusieurs types. De fait son type le plus général n'est pas celui-ci. Par contre, son implémentation dépendra du type sur lequel on l'utilise et est choisie à la compilation. Voire le cours sur les *type-classes*.
- Remarquez que l'on renvoi un booléen, on peut donc bien écrire `if (2<=3) then ....`

1. C'est en fait une définition (sinon on ne saurait pas sur quoi porte la flèche).

1.4 Comme Type d'Ordre Supérieur

Exemple 4.
Le programme suivant (existant dans Haskell) applique une fonction à chaque élément d'une liste :

```
map :: (Int -> String) -> [Int] -> [String]
map f []    = []
map f (x:l) = (f x) : (map f l)
```

Voici quelques remarques :

- Il y a deux implémentations ! une pour la liste vide et l'autre pour une liste non vide (avec un élément de tête `x` devant une queue de liste `l`). C'est un cas de *pattern-matching*, que l'on va développer dans la prochaine section.
- On utilise les notations suivantes pour les listes : `[]` est la liste vide. `(:)` est la concaténation, c'est un opérateur comme les autres. `[Int]` est le type d'une liste d'entiers. `[3]` est une liste avec un seul élément qui est l'entier `3`.
- Le type est bizarre : il prend deux arguments : Le premier, de type `(Int -> String)`, est une fonction. Le second, de type `[Int]`, est une liste d'entier. Le tout retourne alors une liste de `String`.
  C'est tout à fait légal, en Haskell comme dans les autres langages fonctionnel, de donner une fonction en argument une fonction (qui après tout n'est qu'un bout de programme, donc une donnée comme les autres). On utilise alors le type fonctionnel en argument et on a un type flèche dans un type flèche.
- Attention, ne pas confondre `((Int -> String) -> [Int] -> [String])` et `(Int -> String -> [Int] -> [String])` ; le premier a deux arguments dont une fonction, le second à trois arguments... Par contre, ce type est bien équivalent à `((Int -> String) -> ([Int] -> [String]))`, ce qui veut dire que l'on peut voire `map` comme un programme qui transforme une fonction de `Int` vers `String` en une autre fonction sur les listes correspondantes, de `[Int]` vers `[String]`. De fait, avec l'habitude, cette interprétation est beaucoup plus naturelle, en particulier lorsque l'on comprend le concept de foncteurs (voire le cours associé).

2 Types Algébriques et Pattern Matching

2.1 Enumération

Exemple 5.
Le type suivant (existant dans Haskell) est l'implémentation apparente [2] des Booléens :

2. En interne, ils sont implémentés plus efficacement, mais ils peuvent être manipulés par l'utilisateur comme s'ils étaient implémentés ainsi ; on parle alors d'abstraction.

```
data Boolean = True | False
```

Ceci veut dire qu'un booléen n'a que deux valeurs possibles : les constantes `True` et `False`. Ainsi, `True` et `False` sont tout deux des Booléens admissibles ; et inversement, si l'on a un Booléen, on peut toujours regarder si c'est True ou si c'est False.

Attention, cela ne veut pas dire que les seuls programmes de types `Boolean` sont `True` et `False`. Par exemple, `(2 <= 3)` est bien de type `Boolean` et s'il s'évalue bien en `True`, il faut faire un calcule ; même pire, un programme peut boucler et être un `Boolean` (voire TD1)...

Exemple 6.
Le programme suivant (existant dans Haskell) est une autre [3] implémentation possible de la négation :

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

Ici, on utilise ce que l'on appel le *pattern matching*. L'argument de la fonction étant un Booléen, on sait qu'il ne peut être évalué que en `True` ou en `False`. On écrit donc deux implémentations différentes, l'une où l'argument s'avère être `True` et l'autre où il s'avère être `False`.

Exemple 7.
Le programme suivant (qui n'existe pas en Haskell) est une implémentation possible de la factorielle :

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

Quelques remarques :

- On utilise encore un *pattern matching*, cette fois sur les entiers. En effet, les entiers sont vu par Haskell comme une énumération infinie de chacune de leurs valeurs.
- In n'y a pas un code pour chaque constante (heureusement puisqu'il y en a une infinité). Ce n'est pas grave, Haskell va d'abord tester la première, et si ça ne marche pas, il testera la seconde qui ne précise pas de valeur spécifique pour `n`. On sait juste que `n` ne peux pas valoir `0` car le premier programme est alors appliqué avant.

2.2 Type Produit

Exemple 8.
Le type suivant (qui n'existe pas en Haskell) est le type d'un points dans un plan en coordonnées Cartésiennes :

3. mais équivalente

```
data Point = Vect Float Float
```

Cela veut dire que les valeurs que peuvent prendre un point sont de la forme `(Vect x y)` où `x` et `y` sont des flottants. Ainsi, `(Vect 3 (2.24))` ou `(Vect (-15.3) (52.887))` sont des points ; et inversement, si l'on a un Booléen, on peut toujours récupérer ses coordonnées gauche et droite. `Vect` est appelé *constructeur* du type algébrique `Point`, il est considéré comme une fonction de type `Vect :: Float -> Float -> Pont`.

Dans la pratique, lorsque l'on a qu'un seul constructeur pour un type algébrique (on verra que l'on peut en avoir plus), une bonne pratique consiste à lui donner le même nom que le type définit. De cette façon, on a besoin de se souvenir que d'un seul nom, par contre il ne faut bien comprendre que le type et le constructeur sont des entités séparées avec le même nom (le compilateur arrive a faire la différence). Ainsi, un meilleur code pour `Point` est le suivant :

```
data Point = Point Float Float
```

Exemple 9.
Le programme suivant (qui n'existe pas en Haskell) prend la distance à l'origine :

```
norme :: Point -> Float
norme (Point x y) = sqrt (x^2 + y^2)
```

Ici, on utilise encore le *pattern matching*. L'argument de la fonction étant un point, on sait qu'il ne peut être évalué que en `(Point x y)`. On peut donc récupérer l'abscisse et l'ordonnée ainsi.

2.3 Mixer Les Deux

Exemple 10.
Le type suivant est une simplification du type `Maybe` existant vraiment en Haskell, il s'agit d'un entier qui peut avoir été trouvé ou rester inconnu (sorte d'erreur) :

```
data MaybeInt = Just Int | Nothing
```

Cela veut dire que les valeurs que peuvent prendre un `MaybeInt` sont ou bien de la forme `(Just n)` où `n` est un entier, ou bien la constante `Nothing`. Ainsi, `(Just 0)` `(Just (-53))` ou `Nothing` sont de type `MaybeInt` ; et inversement, si l'on a un `MaybeInt`, on peut toujours savoir si c'est une erreur ou si on a bien un entier, de plus, dans le second cas, on peut récupérer cet entier.

Exemple 11.
Le programme suivant (qui n'existe pas en Haskell) fait une division sur un `MaybeInt` :

```
divise :: MaybeInt -> Int -> MaybeInt
divise Nothing  _ = Nothing
divise _        0 = Nothing
divise (Just m) n = Just (m `div` n)
```

Quelques remarques :

- Il y a des *underscores* `_`. Cela indique une variable qui n'est pas utilisé. Ça peut faciliter la tâche du compilateur en lui disant qu'il n'a pas à regarder cette variable ; cela va surtout aider le lecteur habitué qui va pouvoir lire rapidement le code sans avoir a se souvenir du nom d'une variable pas utilisée.
- On a trois cas ! En effet, on fait ici deux *pattern matching*, l'un sur les entiers, l'autre sur le type algébrique `MaybeInt`. On dit que l'on retourne `Nothing` dès que le premier argument est un `Nothing` ou que le second est un `0` ; ainsi on n'arrive au troisième cas lorsque le premier argument est un `Something` et que le second est différent de `0`.
- On utilise la division d'entier `(div :: Int->Int->Int)` comme un opérateur infixe ; c'est pour ça qu'il y a des *quotes* autour.

2.4 Types Algébriques Récursifs

Exemple 12.
Le type suivant est une simplification du type `[Int]` d'entier des liste d'entiers :

```
data ListInt =  Int : ListInt  |  []
```

Quelques remarques :

- (hors cours) On utilise un opérateur `(:)` comme constructeur. On peut en fait utiliser n'importe quel symbole d'opérateur commencent par " `:` " comme constructeur (par exemple `:?` ou `:%%`). Par contre l'utilisation d'un symbole `[]` comme constante est normalement interdit ; les listes sont une exception de la bibliothèque standard.
- `ListInt` est définit à partir de lui-même. Ce n'est pas un soucis, au contraire, c'est ainsi que l'on peut défini des structures de données de taille non bornée voire infinies.
- On pourrait vouloir des listes d'autre chose que des entiers sans changer la syntaxe... C'est ce que l'on fait à la prochaine section.

Exemple 13.
Le programme suivant (qui n'existe pas en Haskell) somme les éléments d'une lise deux par deux :

```
sumByPaires :: ListInt -> ListInt
sumByPaires []      = []
sumByPaires [x]     = [x]
sumByPaires (x:y:l) = (x+y) : (sumByPaires l)
```

Quelques remarques :

- la notation `[x]` est en fait du sucre syntaxique pour `(x:[])`.
- la notation `(x:y:l)` veut dire `(x:(y:l))`, en effet, le cons `(:)` associe à droite.

— Il y a encours 3 cas ! En effet, on fait ici un double *pattern matching* : on évacue le cas de la liste vide pour ouvrir une fois le contenu de la liste, puis on refait un *pattern matching* en évacuant le cas du singleton pour ouvrir une seconde fois la liste.

Ce genre de code est très concis et pratique. Attention tout de même à traiter tous les cas. Il arrive qu'il y ai un cas que l'on ne veuille pas traiter car on n'est pas sensé l'avoir lorsque l'on arrive dans cet endroit du code ; dans ce cas là utilisez une erreur (comme `undefined`) plutôt que de ne pas terminer le *pattern matching*.

3 Polymorphisme

Dans de nombreux exemples au dessus, on a fait la remarque qu'un code ou un type algébrique pourrait être généralisé en incluant d'autres types. C'est le polymorphisme.

3.1 Types Paramétrés

Exemple 14. Voici le vrais `Maybe` :

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

Le `a`, ici, est une *variable de type*. Il représente n'importe quel type Haskell. Ce qui est important c'est que c'est le même partout. Ainsi, lorsque l'on écrit `Just t`, si le terme `t` à le type `t :: a`, on aura `Just x :: Maybe a`. Celui-ci est donc paramétrique en une variable de type.

(Hors cours) On sait que `Maybe Int` est un type. Mais qu'en est-il de `Maybe` tout seul ? si vous tapez `:kind Maybe` suivante dans `ghci`, celui-ci vous dira que `Maybe :: Type->Type` (ou que `Maybe :: *->*` selon les versions). Cela veut dire que `Maybe` est paramétré, il a besoin d'un type en argument pour devenir un type, et `Type->Type` est sa sorte, qui est un "type de types". En fait, il est possible de définir des objets de sorte aussi complexe que l'on veut, comme `StateT :: Type -> (Type -> Type) -> Type -> Type`

Exemple 15. Voici un type possible pour les arbres binaires :

```
data Arbre a =  Noeud (Arbre a) a (Arbre a)
              | Feuille
```

Cela signifie qu'un `t :: Arbre Int` est un arbre ne contenant que des entier, ou qu'un `t :: Arbre (Maybe Bool)` est un arbre ne contenant que des `Maybe Bool`, c'est à dire que des feulles et des noeud de la forme [4] `Noeud arbreG Nothing arbreD` ou bien `Noeud arbreG (Just x) arbreD`.

Remarquez aussi que on ne peut pas avoir deux valeurs de types différents dans un même arbre.

4. Attention, en vérité, certains noeuds peuvent aussi ne pas être encore évalués (on est paresseux !).

Exemple 16.
Les couples de valeurs, en haskell, seraient définit ainsi :

```
data (,) a b =  a,b
```

Quelques remarques :

- On peut introduire plusieurs paramètre. Cela veut dire que la sorte du couple est `(,) :: Type -> Type -> Type`.
- On verra souvent écrit `(a,b)` à la place de `((,) a b)`. Le premier est du sucre syntaxique pour le second.
- (hors cours) utiliser la virgule comme constructeur est sensé être interdit (cela ne commence pas par `:`), c'est encore une exception de la librairie standard.
- On remarquera quel le type paramétré `(,)` et le constructeur `(,)` utilisent la même notation, c'est standard lorsque l'on a qu'un seul constructeur.

Exemple 17 (Les listes, une syntaxe exceptionelle).
Voici l'implémentation apparente [5] des Listes en Haskell :

```
data [] a =  a : ([] a)  |  []
```

Il est bizare, et ce pour plusieurs raisons, d'abord car vous ne pourez pas écrire ca directement en Haskel car les liste sont une exception à la syntaxe, ensuite car on utilise le même symbole deux fois :

- Remarquez que `[]` est utilisé à la fois comme nom du type paramétré et comme un des constructeurs. Ce n'est pas interdit et même recommendé lorsqu'il n'y a qu'un constructeur (voir la section sur les wrapers), mais dans ce cas là ce n'est pas recomandé.
- Il n'est normalement pas possible d'écrire un nom de types paramétré comme `[]` avec des symboles [6].
- De même, on ne devrait pas pouvoir avoir de symbole `[]` comme constructeur pour la liste vide...
- On verra souvent écrit `[Int]` à la place de `([] Int)`. Le premier est du sucre syntaxique pour le second.

Le type des liste est si exceptionnel pour des raisons historique et de compatibilité avec d'autres langages.

3.2 Fonctions Polymorphes

Un type polymorphe est un type contenant une ou plusieurs variables de types que l'on peut instancier comme on veut.

Exemple 18.
On se souvient du code de map :

```
map f []     = []
map f (x:l) = (f x) : (map f l)
```

5. En interne, elles sont implémentées plus efficacement, mais elles peuvent être manipulées par l'utilisateur comme si elles étaient implémentées ainsi ; encore de l'abstraction.
6. En fait, on peut mais il faut utiliser une extension de syntaxe comme `XTypeOperators`.

Si vous tapez `:t map` dans *ghci*, il vous donnera le type suivant :

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

`map` peut donc s'appliquer à n'importe quelle fonction et n'importe quelle liste tant que le type d'entrée de la fonction est le même que le type du contenu de la liste (c'est à dire du `a`). Dit autrement, `map` transforme n'importe quelle fonction en une fonction sur des listes du type correspondant.

3.3 Type Le Plus Général

Attention, un même programme peut avoir plusieurs types ; par exemple, `map` a le type `(a -> b) -> [a] -> [b]` mais aussi le type `(Int -> Float) -> [Int] -> [Float]` ou le type `(a -> a) -> [a] -> [a]`... Par contre, il n'y a qu'un seul *type le plus général*, c'est à dire le plus polymorphe. Lorsque l'on demande le type d'une fonction, par défaut, on demande le type le plus général (du moins si vous voulez le maximum de points).

Le *type le plus général* donne énormément d'information sur ce que fait une fonction. En fait, il se trouve que l'on recherche souvent la fonction la plus simple/naturelle qui aurait exactement ce type. Par exemple, la seule fonction qui ai comme type le plus général `a->a` est l'identité.

Exemple 19.
Voici le code de `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []  = []
filter p (x:l) | p x       = x : (filter f l)
               | otherwise = filter f l
```

De part son type, on sait qu'elle prend une fonction `a->Bool` qui peut accepter ou refuser des termes de types `a`, ainsi qu'un second argument de type `[a]` qui est une liste d'éléments de ce même type `a`, le résultat est aussi de type `[a]`. On pourait obtenir un type semblable en retournant le second argument, mais ce ne serait pas le plus général (pourquoi le premier argument serait demandé de type `a->Bool`. Il faut donc utiliser le premier argument pour modifier le seconde ; le plus naturel lorque l'on a une fonction d'acceptation et une liste est alors de suprimer tous les éléments refusés de la liste ; c'est ce que fait `filter`.

Pour ceux qui ne l'avaient pas encore vu, les barres verticales `|` servent à mattre des gardes. Ce sont des conditions à ajouter au pattern-matching qui permettent d'éviter d'écrire des `if..then.else...` Ainsi les deuxième et troisième lignes se lisent "si `x:l` est non vide et si `p x` s'évalue en `True`, alors on renvoit `x` suivit du reste de la liste filtrée (le filtre garde donc le `x`), si `x:l` est non vide mais que `p x` s'évalue en `False`, on renvoit le reste de la liste filtrée sans `x` (qui a été enlevé par le filtre)."

3.4 Conventions de Casses

La casse (majuscule ou minuscule) de la première lettre des mots écrit dans votre code Haskell est extrêmement importante ; le compileur s'en sert pour comprendre quel genre d'objet il s'agit.

Dans un type :

— Un mot (ou une lettre) commencent par une minuscule est forcément une variable de type (de n'importe quelle sorte, mais le plus souvent un type).

— Un mot (ou une lettre) commencent par une majuscule est forcément un type déjà définit (par vous ou une librairie)

— (hors cours) Un symbole (autre que `=` et `::`) est un opérateur de type (comme `(->)`, `(,)` ou `[]`).

Dans une fonction :

— Un mot (ou une lettre) commencent par une minuscule est forcément une variable de terme (de n'importe quel type) ou une fonction.

— Un mot (ou une lettre) commencent par une majuscule est forcément un constructeur d'un type algébrique (ou synonyme de type ou *wraper*, voire la section suivante).

— (hors cours) Un symbole ne commencent pas pas `:` (autre que `==`) est un opérateur (comme `(<=)`, `(<>)` ou `(++)`).

— (hors cours) Un symbole commencent par `:` (autre que `::`) est un constructeur (comme `:`).

4 Utilisation des Type Classes

On se souvient du code de `max` :

```
max x y = if (x >= y) then x else y
```

Il est clair que ce code a le type

```
max \verb+Int->Int->Int+
```

mais vous pouvez vérifier qu'il a aussi les types `Float->Float->Float` et `String->String->String` en tapant du *ghci*

```
Prelude> max 0.1 0.2
0.2
Prelude> max "foo" "bar"
"foo"
```

ou en essayant directement :

```
{haskell}
Prelude> :t max :: Float->Float->Float
max :: Float->Float->Float    :: Float -> Float -> Float
Prelude> :t max :: String->String->String
max :: String->String->String :: String -> String -> String
```

On pourrait alors penser qu'il est de type `a->a->a`, c'est à dire qu'il prend n'importe quel arguments, pouvu qu'ils aient le même types. Malheureusement, ce n'est pas le cas, car il n'a pas de type `(Int->Int)->(Int->Int)->(Int->Int)`, comme on peut le vérifier sur ghci :

```
Prelude> max (+1) (*2)
<interactive>:11:1:
    No instance for (Ord (a0 -> a0)) arising from a use of ‘max’
  .... bla bla bla ...
```

en effet il est difficile de dire laquelle de deux fonctions sur les entiers est la plus grande...

En fait `max` est te type `a->a->a` pour tout `a` qui dispose d'un ordre, c'est à dire tel que `(<=): a->a->Bool` est bien définit. Ce sont les `a` qui instancient la *type-class* `Ord` (voire cours sur les *type-classes*). On note donc :

```
max :: (Ord a) => a -> a -> a
```

La double flèche `=>` n'indique pas un type fonctionnel, mais une contrainte sur les variables de type utilisées (en l'occurrence `a`). Il faut lire "pour tout type `a` de la classe `Ord`, la fonction `max` est de type `a->a->a`.

In autre exemple important est la fonction `show` de type :

```
show :: (Show a) => a -> String
```

Elle va imprimer (dans un sting) n'importe quel type pourvus que celui-ci soit de la classe `Show`. C'est la fonction qu'utilise ghci pour vous montrer les résultats. Notez que tous les types ne sont pas de cette classe, c'est pour ca que ghci ne veut jamais afficher des fonctions. Pire, l'implémentation de `show` étant différente selon le type entré, si ghci n'arrive pas à déterminer précisément un type non-polymorphique pour ce que vous lui demandez d'afficher, il vous renvera potentiellement une erreur.

5 Types Synonymes et Wrappers (hors cours)

Il y a deux autres syntaxes que `data ..` pour introduir un nouveau type, ce sont les syntaxes des types synonymes et des wrappers :

```
type NouveauSynonyme   = ...
newtype NouveauWrapper = ...
```

Tous deux ont la même fonction : fournir un autre nom à un type qui existe déjà. Mais ils ont une différence : le type synonyme va unifier les deux types avant le typage tandit que le wrapper va les unnifier après.

Exemple 20. Si j'écrit

```
type Clef = Int
```

alors les mots `Clef` et `Int` désignent la même chose exactement, on peut utiliser l'un ou l'autre, ca n'a pas d'importance. Par exemple, on aurra `3::Clef` ou bien `(+) :: Clef -> Int -> Int`. Le seul intéret de faire ca est de rendre les types écrits plus lisibles.

Exemple 21. Si j'écrit

```
newtype Clef = Clef Int
```

alors les mots `Clef` et `Int` sont complètement différents du point de vue du typeur. Par contre, leur implémentation à l'éxécution sera la même. Cela a plusieurs avantage lorsque l'on fait du Haskell avancé, le premier est de de fournir plusieurs implémentation d'une même classe par un même type ; le second est d'ajouter des contraintes par rapport au type wrappé afin de détecter les erreurs d'utilisations.