Principes de Programmation
Fiche de Révisions:
Type-Classes et Égalités de Programmes

22 juin 2017

1 Type-Classes

Moralement, une *type-class*, ou "classe de types", est un ensemble de types (ou de types paramétrés) qui ont un comportement similaires. Ils peuvent donc être interchangeable dans un programme qui n'utiliserait que ce comportement.

Exemple 1.
- `Eq` : la classe des types dont les éléments disposent d'une relation d'équivalence (ou une égalité).
- `Ord` : la classe des types dont les éléments sont ordonnés.
- `Read` : la classe des types dont on peut afficher les éléments (en faire des `String`).
- `Show` : la classe des types dont on peut parser les éléments (les créer à partir d'une `String`).
- `Enum` : la classe des types dont les éléments peuvent être énumérer. [1]
- `Bounded` : La classe des types disposant d'un minimum et un maximum.
- `Num` : La classe des types se comportant comme des entiers relatifs.
- `Fractional` : La classe des types se comportant comme des nombre rationnels.
- `Functor` : La classe des types paramétrés se comportant comme des structures de données ou des "conteneurs".
- `Monad` : La classe des types paramétrés se comportant comme des environnements de travail.
- etc...

Rappelons la définition formelle :

Définition 1 (Type-Class).
Une *type-class* est annoncé par un code de la forme :

```
class MaClasse c where
  fun1 :: t1
```

1. càd. décompter les un après les autres.

```
  fun2 :: t2
  ...
-- eq1
-- eq2
-- ...
```

Une type class est la donnée :

- D'une sorte ; dans ce cours, on considérera le plus souvent la sorte `Type` des types (tels que `Int` ou `(Float->Boolean)`), ainsi que la sorte `(Type->Type)` des types paramétrés (tels que `[]` ou `Arbre`). Cette sorte est appelée la sorte de la *type-class*
- D'une variable, ici `c`, de la sorte correspondante ; cela veut dire que si la sorte est `Type` alors `c` est un type, et si la sorte est `(Type->Type)`, alors `(c Int)` est un type, mais aussi `(c (Float->Boolean))`...
- De noms de fonctions (ou programmes), ici `fun1, fun2...`, qui doivent être implémentez (voir définition suivante),
- Chaque fonction est accompagné d'un type, ici `t1, t2...`, ce type est donné explicitement à partir :
  - de types connus (comme `Int` ou `Float`),
  - variables de types (comme `a, b...`) représentant n'importe quel type,
  - de la variable donnée pour la *type-class* (ici `c`),
  - composés par des opérateurs de types (comme `->` ou `(\_,\_)`).
- Une *type-class* est souvent accompagnée (explicitement ou implicitement) d'équations entre ces fonction, celles-ci sont discutées dans la prochaine section.

Avancé : Il est possible de préciser une relation de sous-classe. En effet, on peut demander à ce que les instances de `MaClasse` soient toutes des instances de `MaSurClasse`, dans ce cas on indique :

```
class MaSurClasse c => MaClasse c where
  fun1 :: t1
  fun2 :: t2
  ...
-- eq1
-- eq2
-- ...
```

De plus, il est généralement nécessaire de n'implémenter qu'un sous-ensemble des fonctions d'une classe, les autres sont implémentées automatiquement (grâce aux équations). On précisera donc souvent les fonctions "requises", c'est à dire celles que l'on doit implémenter.

Exemple 2.
La classe des monoides, de sorte `Type` :

```
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

```
--  mempty <> x         =   x
--  x <> mempty         =   x
--  (x <> y) <> z       =   x <> (y <> z)
```

La classe des types égalisables, de sorte `Type` :

```
class Eq e where
  ==     :: e -> e -> Bool
  /=     :: e -> e -> Bool
-- requis: == | /=
--  x /= y                 =  not (x == y)
-- (x == x)                =  True
-- (x == y)                =  (y == x)
-- (x/=y)||(y/=z)||(x==z)  =  True
```

Remarquez que les deux opérateurs sont indiqués comme requis, mais qu'ils sont séparés par un *pipe* "`|`". Cela veut dire qu'il suffit de définir l'un ou l'autre

Remarquez aussi la forme étrange que prennent les dernières équations (qui ne sont que des notations pour la réflexivité, la symétrie et la transitivité). On en rediscutera.

Exemple 3.
La classe des monades, de sorte `(Type->Type)` est définie comme ça dans Haskell : [2]

```
class Foncteur m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a-> m b) -> m b
   (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
-- requis: return, (>>=)
--  m >> k                   =  m >>= \_ -> k
--  fmap f x                 =  x >>= return . f
--  return a >>= k           =  k a
--  m >>= return             =  m
--  m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Remarquez qu'il y a plus de fonctions et d'équations que dans le cours sur les monades. C'est parce que ce cours se focalise sur une définition minimal, qui n'est pas celle de Haskell, mais qui est suffisante pour la comprendre et l'utiliser. De façon général, nous introduisons souvent des type-classes qui sont plus simple que celles de Haskell. Elles sont généralement équivalentes (à quelques subtilités près) et plus facile à comprendre, mais il faut parfois revenir à la définition original en cas de bug...

Remarque 1.
Le concept de *type-class* est un concept propre à Haskell, mais des concepts très similaires se retrouve dans d'autres langages de programmations. Par exemple :

2. Monade est en fait une sous classe de Applicative, elle même sous classe de Functor, mais comme on ne voit pas les applicatifs dans ce cours, je passe cette subtilité.

– Les interfaces et classes abstraites de Haskell sont très proche dans le concept, malgré quelques subtiles différences. [3]
– Les modules de Ocaml peuvent remplacer les types classes, mais il sont plus lourd à gérer. [4]

Définition 2 (Instance).
Étant donné une *type-class* `MaClasse` et un élément `MonInstance` de la même sorte, on instancie `MaClasse` par `MonInstance` avec un code de la forme :

```
instance MaClasse MonInstance where
  fun1 = code1
  fun2 = code2
  ...
```

où `code1, code2...` sont les programmes implémentant les fonctions `fun1, fun2...`.

Attention, il y a des critères à vérifier :
– Un critère requis par le compilateur : les codes des fonctions implémentées doivent avoir le bon type !
– Un critère vérifié sur papier (ou dans les commentaires) : ces implémentations doivent vérifier les équations de programmes fournies avec la *type-class*.

Avancé : Il est possible d'instancier plein de types à la fois en utilisant des types paramétrés. Pour ça on utilise la notation :

```
instance ClasseA a, ClasseB b => MaClasse MonTypeParam a b c where
  fun1 = code1
  fun2 = code2
  ...
```

qui dit que pour n'importe quel type [5] `a` de la classe `ClasseA`, n'importe quel type `b` de la classe `ClasseB` et n'importe quel type `c`, le type `(MonTypeParam a b c)` est une instance de la classe `MaClasse`.

Exemple 4.
Pour tout type `a`, les listes d'éléments de type `a` forment un monoide :

```
class Monoid [] a where
  mempty = []
  g <> d == g ++ d
```

Attention il faut encore vérifier les équations (voire TD4).

Remarque 2. (Hors programme) Il est possible (mais généralement pas conseiller, de demander la définition d'un type qui serait paramétrique en `a`. C'est utile pour des constructions avancées (comme un type de clé d'accès qui dépend du type de la donné), mais c'est un peu dangereux et devrait s'accompagner de plusieurs vérifications.

3. détallées dans une version future.
4. Ils remplacent aussi les modules de Haskell...
5. Lorsque l'on dit "type" ici, vous pouvez le remplacer par "élément de la sorte correspondante" dans les rares cas (pas ou peu abordés en cours) où cela aurait un sens.

2 Équations de Programmes

Le fait qu'un programme "se comporte bien" peut avoir plusieurs sens :
- L'un d'eux est de définir une sémantique relationnelle ou ensembliste puis vérifier que le programme correspond bien.
- Une autre est de faire passer des tests (de préférence nombreux et générés aléatoirement) et de vérifier le résultat.
- Encore une autre méthode consiste à abstraire le programme comme un automate, un réseau de Pétri ou une autre structure sur laquelle on sait prouver des propriétés automatiquement.
- Il existe encore d'autres méthodes très puissantes comme la possibilité d'introduire des preuves complètes directement dans le type du programme, celles-ci sont vérifiées à la compilation.
- Mais dans notre cas, nous nous intéressons à un cas de "cohérence interne", prouvant qu'un programme est une implémentation correcte d'un concept mathématique, ou d'une type-class. On peut parler de sémantique équationelle ou catégorique.

Le problème fondamental est de savoir si un algorithme (ou un programme) prévu dans un autre cadre d'application peut s'étendre à l'utilisation que l'on veut en faire.

L'algorithme (ou le programme correspondant) est alors prouvé correcte relativement à certaines conditions sur son entrée. De même, la structure (type + fonctions associées) sur laquelle on l'utilise doit alors vérifier ces mêmes axiomes. On a coupé le travail en deux, obtenant plus de modularité dans l'utilisation de l'algorithme.

Quelle forment vont alors prendre ces "axiomes" ? Généralement, ceux-ci seront ramener à des *équations de programme*. Une "équation de programme" est la donnée de deux programmes de même type que l'on prétend être équivalent, au sens où ils se comportent de la même façon dans le même cadre. Ou, plus formellement :

Définition 3.
Étant donné un ensemble de variables typées `x::T1,...,z::Tn`, une équation de programme est la donnée de deux programmes $p1$ et $p2$ de même type et utilisant ces variables. Ces variables sont appelés variables libres de l'équation.

Cette équation est vérifiée si on peut transformer `p1` en `p2` à l'aide d'équations plus simples et prouvées. Voici une liste des équations autorisées :
- `p1 p2 p2 = (p1 p2) p3` car lorsque l'on a deux applications à la suite, on considère toujours qu'il y a des parenthèse invisible à gauche. De même on peut toujours ajouter ou supprimer des parenthèses qui ne servent à rien (mais faites attention).
- `(\x -> p1) p2  = p1'` où `p1'` est le programme `p1` sauf que toutes les occurrences syntaxiques de `x` ont été remplacées par `p2`. C'est la β-équivalence. [6] Par exemple on a l'égalité `(\n -> n+n) 2 = 2+2`.

6. c'est comme la β-réduction du cours de L2, sauf que l'on peut aller dans les deux sens.

- Si on a la définition `f x1 .. xn = pr` alors on a l'égalité `p = p'` où `p'` est le programme `r` dans lequel une occurrence de `f` a été remplacée par `(\x1 -> ... \xn -> pr)`. C'est la définition de `f`.
- `(\x -> p x) = p` à condition que `p` ai un type fonctionnel et que `x` n'apparaissent pas dans son code. C'est l'`\eta`-équivalence.
- `(\x -> p) = (\y -> p')` si `y` n'apparaît pas dans `p` et que `p'` est le programme `p` où toutes les occurrences syntaxiques de `x` ont été remplacées par `y`. C'est l'α-équivalence.
- Toute équation supposée au départ. Par exemple si ont suppose que `a` instancie la classe `Eq` et que `p :: a`, alors on a l'équation `(p==p) = True`.
- Toute équation sur des structures connues. Par exemple on peut immédiatement résoudre `2+2 = 4`.
- Toute équation prouvée en cours ou TD (à moins que l'on demande justement de prouver celle-ci en examen...).
- Toute contextualisation d'une de ces équation. Par exemple, on a `5 * ((\n -> n+n) 2)  =  5 * (2+2)`.
- Toute inversion de ces équations.

Il est aussi admit de prendre des raccourcis de quelques règles si la réduction est clair...

3 Équations pour les Types Fonctionnels et Algébriques

Lorsque l'on veut prouver des équations sur un type fonctionnel `(a -> b)`, c'est à dire lorsque l'on veut prouver que `p1 = p2` avec `p1` et `p2` de type `a->b`, il faut généralement prouver que pour tout `x::a`, on a l'équation `(p1 x) = (p2 x)`. Lorsque l'on veut prouver des équations qui utilisent des types fonctionnelles, c'est à dire avec une variable libre `(f :: a->b)`, et bien rien ne change vraiment, on applique les même règles.

Lorsque l'on veut prouver des équations sur les types algébriques (comme `Boolean`, `[a]` ou encore `Automate`), tout se passe de la même façon. Par contre, lorsque prouver des équations qui utilisent des types algébriques, il faut bien souvent ouvrir le type. Il s'agit en fait de faire comme avec le pattern matching : une étude de cas.

En effet, si on veut prouver que `Boolean` est une instance ce `Eq`, il faut prouver que `(x==x)=True` pour n'importe quel booléen `x`. On fait donc deux cas, le cas où `x=True` et le cas où `x=False`.

Reste à savoir ce que l'on fait lorsque l'on a des entiers naturels ou lorsque l'on dispose de types algébriques récursifs. Dans ce cas, il faut généralement faire une récursion. Par exemple, pour prouver que pour toute liste `l::[a]` on a `(l==l) = True`, il faut faire une récursion, prouver que pour `l=[]` on a bien `([]==[]) = True` puis prouver que si `(l'==l') = True`, alors pour tout `(x::a)` on a `(x::l' == x::l') = True`.

Faite attention : l'utilisation de la récursion oubli généralement des cas.

En particulier, sur les entiers négatifs ou sur les structures infinies (on a vu qu'il était possible de définir des listes infinies en Haskell), ces équations seront probablement fausses. Par exemple, si on lance dans `ghci` le teste `[0..]==[0..]`, et bien le programme va boucler. De tels comportements sur des valeurs extrêmes sont admissible (c'est la fameuse différence entre la théorie et la pratique), mais il faudrait au moins noter ce cas dangereux dans les commentaires. Ainsi, si vous utilisez une récursion indiquez toujours que vous ne garantissez rien sur la valeurs négatives ou infinies...

4 Type-Classes connues

Remarque 3.
Il est possible de dériver certaines *type-classes* automatiquement.

On peut dériver les types suivants sur un type algébrique `MonTypeAlg` :

- `Eq`, `Ord`, `Show` et `Read` dans les cas où `MonTypeAlg` n'est définit qu'à partir de types les implémentant et à partir de lui même (définition récursive).
- (hors programme) `Enum` et `Bounded` lorsque `MonTypeAlg` ne comporte que des constantes.
- (hors programme) avec l'extension de langage `GeneralizedNewtypeDeriving`, un *wraper* (introduit avec `newtype`) autour d'un type `MonType` peut dériver n'importe quelle classe implémenté par `MonType`.

Les équations sont alors automatiquement vérifiées, excepté qu'une récurrence est utilisé dans le cas de types algébriques récursifs, il a a donc des cas bouclant sur les structures infinies.

4.1 Eq

Il s'agit de la classe des types disposant d'une relation d'équivalence, *i.e.*, dont on peut différencier/identifier les membres.

```
class Eq e where
  ==      :: e -> e -> Bool
  /=      :: e -> e -> Bool
-- requis: == | /=
--   x /= y                 =  not (x == y)
-- (x == x)                 =  True
-- (x == y)                 =  (y == x)
-- (x/=y)||(y/=z)||(x==z)   =  True
```

4.1.1 Dérivation :

Il est possible de dériver `Eq` sur n'importe quelle type algébrique qui n'est définit que à partir de types instanciant `Eq`. La dérivation automatique de l'égalité `(x==y)` va alors vérifier si les constructeurs utilisés pour `x` et `y` sont les mêmes (à l'aide d'un pattern matching), puis s'il y a des arguments à ce même constructeurs, on vérifiera s'ils sont les mêmes.

Exemple 5. Un type fonctinnel comme `(Int->Int)` n'est pas (de base) une instance de `Eq`.

Par contre, les listes sont toutes des instances de `Eq`. L'égalité résultante est la suivante :

```
[]    == []     = True
(x:k) == (y:l)  = (x==l) && (k==l)
k     == l      = False
```

Le troisième cas n'arrive que lorsque aucun des deux patern-matching précédent n'a fonctionné. C'est à dire lorsque l'on a une liste vide en face d'une liste non-vide. Remarquez aussi l'appel `(x==y)` à l'égalité sur les éléments de la liste (qui sera implémentée différemment. Par contre, l'appel `(k==l)` est bien un appel récursif sur la sous-liste, il se peut donc l'égalité boucle sur des listes infinies...

4.2 Ord

Il s'agit de la classe des types disposant d'une relation d'ordre totale, *i.e.*, dont on peut comparer les membres.

4.2.1 Définition simplifiée et idéalisée

```
class Eq e => Ord e where
  <       :: e -> e -> Bool
  <=      :: e -> e -> Bool
  >       :: e -> e -> Bool
  <=      :: e -> e -> Bool
-- requis: < | <= | > | >=
--   x >= y                =  y <= x
--   x >  y                =  not (x <= y)
--   x <  y                =  not (x >= y)
--   x == y                =  (x >= y) && (y >= x)
-- (x >= y) || (y >= x)    =  True
-- (x<y)||(y<z)||(x>=z)    =  True
```

(hors programme) Remarquez, en particulier, la condition `(Eq e => Ord e)` qui fait de `Ord` une sous classe de `Eq`. Cela veut dire que tout implémentation de `Ord` doit être une imlémentation de `Eq`. Malheureusement, le "doit être" n'a pas un sens bien définit, de temps en temps il est demandé d'instancier la sur-classe avant et de temps en temps, celle-ci est instanciée automatiquement en fonction de la sous-classe. Dans le cas de `Eq`, elle devrait être instanciée automatiquement à partir de `Ord`, mais cela n'est malheureusement pas implémenté encore...

4.2.2 Définition réelle (hors programme)

```
data Ordering = LT | EQ | GT
```

```haskell
class Eq e => Ord e where
  compare :: a -> a -> Ordering
  <       :: e -> e -> Bool
  <=      :: e -> e -> Bool
  >       :: e -> e -> Bool
  <=      :: e -> e -> Bool
  max     :: e -> e -> e
  min     :: e -> e -> e
-- requis: (==,compare) | (==,<=)
```

Plusieurs remarques à faire :

- Il y a beaucoup d'opérateurs de définit. Par exemple `(compare x y)` rend une des trois constantes `LT`, `EQ` ou `GT` selon que `x` soit plus petit, égal (au sens de `==`) ou supérieur à `y`. Cela ne semble pas très naturelle de les définir ici, plutôt que de définir séparément un `max::(Ord a)=>a->a->a`, mais cela permet de donner des implémentations plus efficaces dans le cas de structures bas niveau (entiers, flotants, etc...).
- Il faut forcément définir `==` et (`compare` ou `<=`), en fait il n'est pas possible de juste définir `<` par exemple. Il s'agit d'un défaut d'implémentation qui sera probablement corrigé dans une version ultérieur de Haskell. Les langages de programmations, pour des raisons historiques, sont plein de ce genre de "bug de conception".
- On n'écrit pas ici toutes les égalités requises car elles sont trop nombreuses, on fait simplement confiance à l'utilisateur pour savoir ce qu'est un ordre... C'est une pratique courant mais dangereuse, par exemple, il est difficile de se rendre compte que l'on impose en fait que l'ordre soit totale, sans quoi on casse des algos correcte comme l'insertion dans un arbre de recherche.
- Si vous regardez la doc officielle, vous verrez que l'implémentation de `==` n'est pas considérée comme requise, mais si vous essayez d'implémenter `Ord` sans avoir implémenter `Eq`, vous aurez un message d'erreur...

4.2.3 Dérivation :

Comme pour `Eq`, il est possible de dériver `Ord` sur n'importe quelle type algébrique qui n'est définit que à partir de types instanciant `Ord`. La dérivation automatique de la comparaison `compare` va alors comparer les constructeurs utilisés pour `x` et `y` (selon l'ordre dans lequel ils ont été définit), et s'ils sont les mêmes et qu'il y a des arguments on récurse sur les arguments.

Exemple 6. Par exemple, sur le type Maybe :

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

La dérivation de `Ord` créera la fonction suivante :

```haskell
instance Ord a => Ord Maybe a where
  (Just x) <= (Just y)   =  x <= y
  (Just _) <= Nothing    = False
```

```
  Nothing  <= (Just _)    = True
  Nothing  <= Nothing     = True
```

4.3 Show

C'est la classe des types dont les éléments peuvent être affichés sous forme de `String`. Sauf exception, une fonction d'affichage en *Haskell* passera par un appel à `show`.

4.3.1 Définition simplifiée et idéalisée

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

Il n'y a pas d'égalité à vérifier ici, il suffit d'implémenter une fonction d'affichage...

4.3.2 Définition réelle (hors programme)

La définition réelle passe par la définition d'une autre méthode (non requise) appelée `ShowPrec`. Mais l'utilisation de cette méthode est complexe et son utilité très discutable. [7]

```
class Show a where
  show     ::  a   -> String
  showList :: [a] -> String -> String
-- requis: show
```

On ajoute ici une fonction `showList` qui affiche une liste d'éléments de types `a` essentiellement (généralement précédée de `[`, suivit de `]` et séparé par des virgules) puis affiche le second argument... Cette fonction est généralement moins intéressante que `show` que l'on peut appliquer directement sur une liste. En effet, il existe l'instance `(instance Show a => Show [a])` ce qui veut dire que l'on peut afficher une liste de `a` avec `show` quoi qu'il arrive. En fait `showList` ne sert que dans les rare cas où l'on voudrait afficher la liste autrement (car `a` est un caractère par exemple)...

4.3.3 Dérivation :

Comme pour les classes précédentes, il est possible de dériver `Ord` sur n'importe quelle type algébrique qui n'est définit que à partir de types instanciant `Ord`. La dérivation automatique de `show` va alors écrire le code correspondant à la valeur en question.

Exemple 7. Par exemple, sur le type Maybe :

7. Je n'ai pas les détail sur sa présence, mais il s'agit probablement d'une *feature* obsolète.

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

La dérivation automatique créera la fonction suivante :

```haskell
instance Show a => Show Maybe a where
  show (Just a) = ``Just ``++ (show a)
  show Nothing  = Nothing
```

4.4 Read

C'est la classe des types dont les éléments peuvent être parsés depuis un `String`.

4.4.1 Définition simplifiée et idéalisée

```haskell
class Show a where
  read    :: String -> a
-- requis: show
```

4.4.2 Remarque

La fonction `read` est généralement l'inverse de la fonction `show`. Mais elle est utilisée de moins en moins souvent (ou sur un ensemble de types restreint), au point qu'elle est presque obsolète. Mais elle permet de faire de jolis exercices...

Maintenant, on préférera utiliser des versions plus complexes, telles que les persistants qui permettent une interaction avec des bases de données, un fichier *XML* ou une requête *html*.

4.5 Num

C'est la classe des types dont les éléments sont des extensions/spécialisations des entiers. Moralement, elle correspond à la classe des implémentations possible des entiers. Mathématiquement, il s'agit plus ou moins de la classe des annaux.

4.5.1 Définition simplifiée et idéalisée

```haskell
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  fromInteger   :: Integer -> a
-- requis:  + , (- | negate) , * , fromInteger
-- negate x = (fromInteger 0) - x
-- x - y    = x + (negate y)
-- equations d'anneau:
-- (x + y) + z           = x + (y + z)
--  x + y                = y + x
-- (x * y) * z           = x * (y * z)
--  x * (y + z)          = x * y + x * z
```

```
-- (y + z) * x           = y * x + z * x
--  a + (fromInteger 0)  = a
-- a + (negate a)        = (fromInteger 0)
-- a * (fromInteger 1)   = a
-- (fromInteger 1) * a   = a
-- equations de morphisme d'anneau:
-- fromInteger (m + n)    = fromInteger m + fromInteger n
-- fromInteger (m * n)    = fromInteger m * fromInteger n
-- fromInteger (negate n) = negate (fromInteger m)
```

Cela fait beaucoup (trop) d'équations. C'est pour ça que l'on n'invente pas "comme ça" un nouveau `Num`, il s'agit d'objets connus et acceptés (nombre complexes, matrices, flotants, etc...).

4.5.2 Définition réelle (hors programme)

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs           :: a -> a
  signum        :: a -> a
  fromInteger   :: Integer -> a
-- requis:  + , (- | negate) , * , fromInteger
-- negate x = (fromInteger 0) - x
-- x - y    = x + (negate y)
-- equations d'anneau:
-- (x + y) + z           = x + (y + z)
--  x + y                = y + x
-- (x * y) * z           = x * (y * z)
--  x * (y + z)          = x * y + x * z
-- (y + z) * x           = y * x + z * x
--  a + (fromInteger 0)  = a
-- a + (negate a)        = (fromInteger 0)
-- a * (fromInteger 1)   = a
-- (fromInteger 1) * a   = a
-- equations de morphisme d'anneau:
-- fromInteger (m + n)    = fromInteger m + fromInteger n
-- fromInteger (m * n)    = fromInteger m * fromInteger n
-- fromInteger (negate n) = negate (fromInteger m)
-- equations pour abs et signum:
-- abs (negate x)         = abs x
-- abs ((signum x)*x)     = x
-- (signum 0)             = 0
-- (signum 1)             = 1
-- signum (negate x)      = negate (signum x)
-- signum (x*y)           = (signum x)*(signum y)
```

Quelques remarques :

- Les opérations `abs` et `signum` ne sont pas très commune mathématiquement, il est donc difficile d'être sure qu'un objet complexe les vérifie bien.
- En réalité, la plupart de ces équations sont fausse pour `Int` et `Float`. Plus exactement, elles sont fausse dans les cas extrême : en cas de débordement ou d'erreur d'approximation. Cela veut dire que les algorithmes peuvent buger dans de tels cas. C'est pour ça que l'on introduit des bibliothèques de calculs de précision...

4.6 Enum (à la limite du programme)

4.7 Bounded (à la limite du programme)

4.8 Monoid (à la limite du programme)

4.9 Functor

4.10 Applicative (hors programme)

4.11 Monad

4.12 StateMonad, IOMonad, etc... (à la limite du programme)

4.13 Foldable (à la limite du programme)

4.14 Traversable (hors programme)

4.15 MonadTransformers (hors programme)