Présentation des classes Haskell

Jean-Luc JOULIN

Version 1
18 septembre 2022

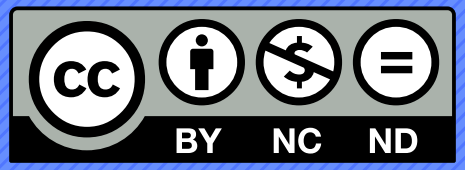
License

Cette présentation est diffusée suivant les termes de la license Creative Common :

BY **Attribution**. Cette présentation peut être librement utilisée, à condition de l'attribuer à l'auteur en citant son nom.

NC **Pas d'utilisation Commerciale**. Aucune utilisation commerciale n'est permise.

ND **Pas de Modification**. Aucune œuvre dérivée basée sur cette présentation n'est autorisée.



Coloration syntaxique

Cette présentation utilise une coloration syntaxique afin de faciliter la lecture du code présenté :

- Mots clef du langage.
- Les fonctions, les opérateurs.
- Les types, les constructeurs.
- Les valeurs numériques.
- Les chaînes de caractères.
- Les parenthèses, les crochets, les commentaires.

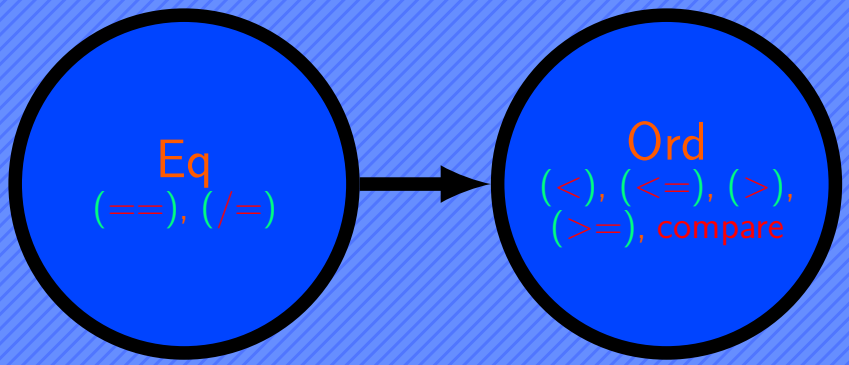
Sommaire

Classes standards

Eq	Classe supportant les tests d'égalité.
Ord	Classe contenant les tests de comparaison.
Show	Classe pouvant être représentée par une chaîne de caractères.
Read	Classe pouvant convertir une chaîne de caractère en type.
Enum	Classe contenant des énumérations.
Num	Classe contenant les types numériques.
Floating	Classe contenant les nombres à virgule flottante.
Integral	Classe contenant les nombres entiers.

Les classes d'égalité et d'ordre

Organisation des classes d'égalité et d'ordre



La classe Eq

- Fonctions pour tester l'égalité entre des éléments.
- Définit les opérateurs $(==)$ et $(/=)$.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition :
  - Soit à $(==)$.
  - Soit à $(/=)$.

La classe Eq (implémentation)

L'implémentation de la classe Eq est la suivante :

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

    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

La définition minimum pour créer une instance de la classe Eq est soit `==`, soit `/=`.

La classe Ord

- Fonctions pour tester les relations d'ordre entre des éléments.
- Définit les opérateurs (<), (<=), (>), (>=).
- Définit les fonctions compare, min et max.
- Utilise le Type Ordering pour donner la relation d'ordre entre deux éléments.
- Peut être étendue à tous les types apprtenant à la classe Eq avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition :
  - Soit à compare.
  - Soit à (<=).

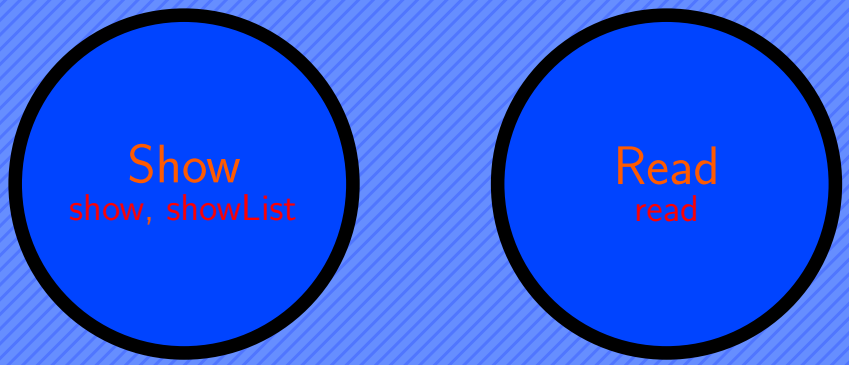
La classe Ord (Implémentation)

L'implémentation de la classe Ord est la suivante :

```
data Ordering = LT | EQ | GT

class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }


    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

Les classes de conversion des chaînes de caractères

Organisation des classes de conversion des String



La classe Show

- Fonctions pour convertir un élément d'un type particulier en chaîne de caractères.
- Définit les fonctions :
  - show Permet de convertir un élément en String.
  - showList Permet de convertir une liste d'éléments en String.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec un instance, il est nécessaire de donner une définition :
  - Soit à show.
  - Soit à showsPrec.

La classe Show (Implémentation)

```haskell
class  Show a  where
    -- | Convert a value to a readable 'String'.
    --
    showsPrec :: Int    -- ^ the operator precedence of the enclosing
                        -- context (a number from @0@ to @11@).
                        -- Function application has precedence @10@.
              -> a      -- ^ the value to be converted to a 'String'
              -> ShowS

    -- | A specialised variant of 'showsPrec', using precedence context
    -- zero, and returning an ordinary 'String'.
    show      :: a   -> String

    -- | The method 'showList' is provided to allow the programmer to
    -- give a specialised way of showing lists of values.
    -- For example, this is used by the predefined 'Show' instance of
    -- the 'Char' type, where values of type 'String' should be shown
    -- in double quotes, rather than between square brackets.
    showList  :: [a] -> ShowS

    showsPrec _ x s = show x ++ s
    show x          = shows x ""
    showList ls   s = showList__ shows ls s
```

La classe Read

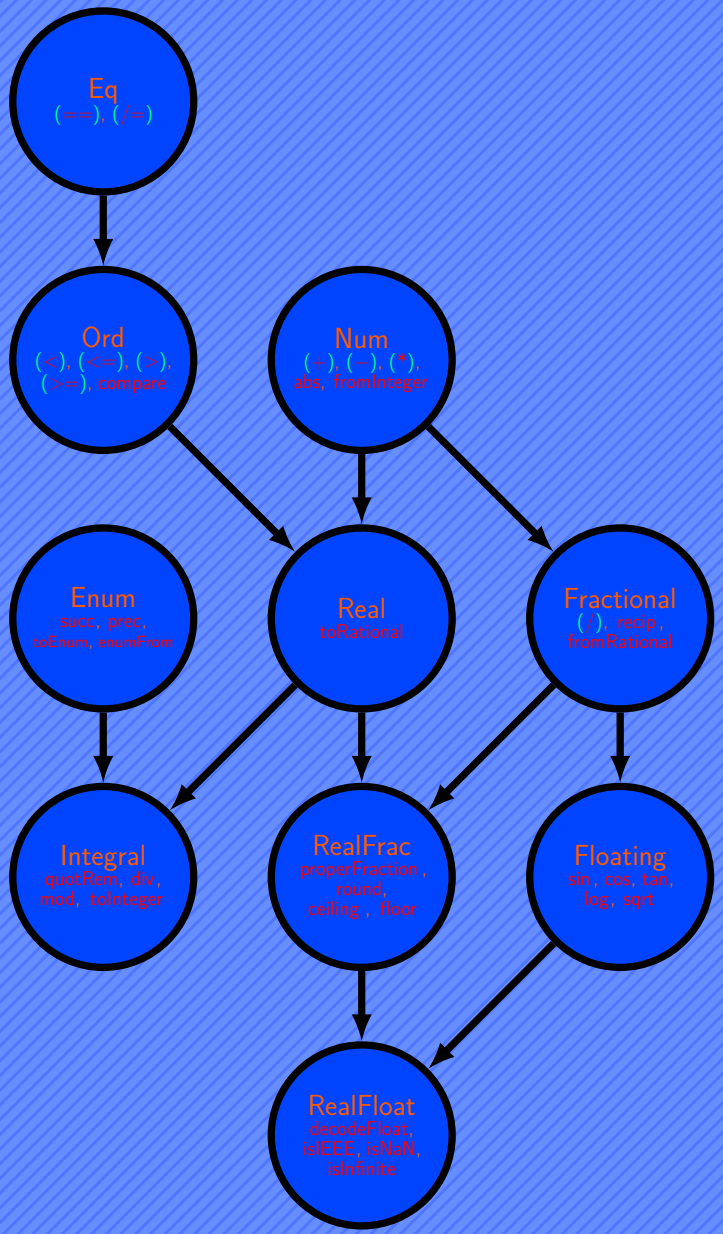
- Fonctions pour convertir une chaîne de caractère String en un élément d'un type spécifique.
- Définit les fonctions readsPrec, readPrec, readList et readListPrec.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition :
  - Soit à readsPrec.
  - Soit à readPrec.

La classe Read (Implémentation)

```haskell
class Read a where
  -- | attempts to parse a value from the front of the string, returning
  -- a list of (parsed value, remaining string) pairs.  If there is no
  -- successful parse, the returned list is empty.
  readsPrec    :: Int    -- ^ the operator precedence of the enclosing
                         -- context (a number from @0@ to @11@).
                         -- Function application has precedence @10@.
               -> ReadS a

  -- | The method 'readList' is provided to allow the programmer to
  -- give a specialised way of parsing lists of values.
  -- For example, this is used by the predefined 'Read' instance of
  -- the 'Char' type, where values of type 'String' should be are
  -- expected to use double quotes, rather than square brackets.
  readList     :: ReadS [a]

  -- | Proposed replacement for 'readsPrec' using new-style parsers (GHC only).
  readPrec     :: ReadPrec a

  -- | Proposed replacement for 'readList' using new-style parsers (GHC only).
  -- The default definition uses 'readList'.  Instances that define 'readPrec'
  -- should also define 'readListPrec' as 'readListPrecDefault'.
  readListPrec :: ReadPrec [a]

  -- default definitions
  readsPrec    = readPrec_to_S readPrec
  readList     = readPrec_to_S (list readPrec) 0
  readPrec     = readS_to_Prec readsPrec
  readListPrec = readS_to_Prec (\_ -> readList)
```

Les classes numériques

Organisation des classes Haskell numériques



La classe Num

- Fonctions pour effectuer des opérations de base sur les nombres.
- Définit les opérateurs standards sur les nombres : (+), (−), (*).
- Définit les fonctions negate, abs, signum, fromInteger.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à (+), (*), abs, signum, fromInteger et :
  - Soit à (−).
  - Soit à negate.

Pour l'opérateur de division (/), voir la classe Fractional.

La classe Num (Implémentation)

```haskell
class  Num a  where
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either @-1@ (negative), @0@ (zero)
    -- or @1@ (positive).
    signum              :: a -> a
    -- | Conversion from an 'Integer'.
    -- An integer literal represents the application of the function
    -- 'fromInteger' to the appropriate value of type 'Integer',
    -- so such literals have type @('Num' a) => a@.
    fromInteger         :: Integer -> a

    x - y               = x + negate y
    negate x            = 0 - x
```

La classe Real

- Permet de convertir les nombres réels en nombre rationnels.
- Définit la fonction toRational.
- Peut être étendue à tous les types appartenant aux classe Num et Ord avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à toRational.

La classe Real (Implémentation)

```haskell
class  (Num a, Ord a) => Real a  where
    -- | the rational equivalent of its real argument with full precision
    toRational          ::  a -> Rational
```

La classe Fractional

- Permet de convertir les nombres réels en nombre rationnels.
- Définit l'opérateur (/).
- Définit les fonctions recip et fromRational.
- Peut être étendue à tous les types appartenant à la classe classe Num avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à fromRational et :
  - Soit à (/).
  - Soit à recip.

La classe Fractional (Implémentation)

```haskell
class  (Num a) => Fractional a  where
    -- | Fractional division.
    (/)                 :: a -> a -> a
    -- | Reciprocal fraction.
    recip               :: a -> a
    -- | Conversion from a 'Rational' (that is @'Ratio' 'Integer'@).
    -- A floating literal stands for an application of 'fromRational'
    -- to a value of type 'Rational', so such literals have type
    -- @('Fractional' a) => a@.
    fromRational        :: Rational -> a

    recip x             =  1 / x
    x / y               = x * recip y
```

La classe RealFrac

- Permet d'extraire les composants de fractions.
- Définit les fonctions properFraction, truncate, round, ceiling , floor .
- Peut être étendue à tous les types appartenant aux classes Real et Fractional avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition properFraction.

La classe RealFrac (Implémentation 1)

```
class  (Real a, Fractional a) => RealFrac a  where
    -- | The function 'properFraction' takes a real fractional number @x@
    -- and returns a pair @(n,f)@ such that @x = n+f@, and:
    --
    -- * @n@ is an integral number with the same sign as @x@; and
    --
    -- * @f@ is a fraction with the same type and sign as @x@,
    --   and with absolute value less than @1@.
    --
    -- The default definitions of the 'ceiling', 'floor', 'truncate'
    -- and 'round' functions are in terms of 'properFraction'.
    properFraction      :: (Integral b) => a -> (b,a)
    -- | @'truncate' x@ returns the integer nearest @x@ between zero and @x@
    truncate            :: (Integral b) => a -> b
    -- | @'round' x@ returns the nearest integer to @x@;
    --   the even integer if @x@ is equidistant between two integers
    round               :: (Integral b) => a -> b
    -- | @'ceiling' x@ returns the least integer not less than @x@
    ceiling             :: (Integral b) => a -> b
    -- | @'floor' x@ returns the greatest integer not greater than @x@
    floor               :: (Integral b) => a -> b
```

La classe RealFrac (Implémentation 2)

```
truncate x          =  m  where (m,_) = properFraction x

round x             =  let (n,r) = properFraction x
                           m     = if r < 0 then n - 1 else n + 1
                       in case signum (abs r - 0.5) of
                            -1 -> n
                            0  -> if even n then n else m
                            1  -> m
                            _  -> errorWithoutStackTrace "round default defn: Bad
                                  value"

ceiling x           =  if r > 0 then n + 1 else n
                       where (n,r) = properFraction x

floor x             =  if r < 0 then n - 1 else n
                       where (n,r) = properFraction x
```

La classe RealFloat

- Apporte des fonctions pour travailler avec les nombres à virgule flottantes.
- Définit les fonctions decodeFloat, encodeFloat, isNaN, isIEEE, isInfinite , isDenormalized, isNegativeZero, floatRadix, floatDigits , floatRange.
- Peut être étendue à tous les types appartenant aux classe RealFrac et Floating avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à decodeFloat, encodeFloat, isNaN, isIEEE, isInfinite , isDenormalized, isNegativeZero, floatRadix, floatDigits , floatRange.

La classe RealFloat (Implémentation 1)

```haskell
-- | Efficient, machine-independent access to the components of a
-- floating-point number.
class  (RealFrac a, Floating a) => RealFloat a  where
    -- | a constant function, returning the radix of the representation
    -- (often @2@)
    floatRadix          :: a -> Integer
    -- | a constant function, returning the number of digits of
    -- 'floatRadix' in the significand
    floatDigits         :: a -> Int
    -- | a constant function, returning the lowest and highest values
    -- the exponent may assume
    floatRange          :: a -> (Int,Int)
    -- | The function 'decodeFloat' applied to a real floating-point
    -- number returns the significand expressed as an 'Integer' and an
    -- appropriately scaled exponent (an 'Int').  If @'decodeFloat' x@
    -- yields @(m,n)@, then @x@ is equal in value to @m*b^^n@, where @b@
    -- is the floating-point radix, and furthermore, either @m@ and @n@
    -- are both zero or else @b^(d-1) <= 'abs' m < b^d@, where @d@ is
    -- the value of @'floatDigits' x@.
```

La classe RealFloat (Implémentation 2)

```
decodeFloat          :: a -> (Integer,Int)
-- | 'encodeFloat' performs the inverse of 'decodeFloat' in the
-- sense that for finite @x@ with the exception of @-0.0@,
-- @'Prelude.uncurry' 'encodeFloat' ('decodeFloat' x) = x@.
-- @'encodeFloat' m n@ is one of the two closest representable
-- floating-point numbers to @m*b^^n@ (or @&#177;Infinity@ if overflow
-- occurs); usually the closer, but if @m@ contains too many bits,
-- the result may be rounded in the wrong direction.
encodeFloat          :: Integer -> Int -> a
-- | 'exponent' corresponds to the second component of 'decodeFloat'.
-- @'exponent' 0 = 0@ and for finite nonzero @x@,
-- @'exponent' x = snd ('decodeFloat' x) + 'floatDigits' x@.
-- If @x@ is a finite floating-point number, it is equal in value to
-- @'significand' x * b ^^ 'exponent' x@, where @b@ is the
-- floating-point radix.
-- The behaviour is unspecified on infinite or @NaN@ values.
exponent             :: a -> Int
-- | The first component of 'decodeFloat', scaled to lie in the open
-- interval (@-1@,@1@), either @0.0@ or of absolute value @>= 1\/b@,
-- where @b@ is the floating-point radix.
-- The behaviour is unspecified on infinite or @NaN@ values.
significand          :: a -> a
-- | multiplies a floating-point number by an integer power of the radix
scaleFloat           :: Int -> a -> a
-- | 'True' if the argument is an IEEE \"not-a-number\" (NaN) value
isNaN                :: a -> Bool
-- | 'True' if the argument is an IEEE infinity or negative infinity
isInfinite           :: a -> Bool
```

La classe RealFloat (Implémentation 3)

```
    isDenormalized      :: a -> Bool
    -- | 'True' if the argument is an IEEE negative zero
    isNegativeZero      :: a -> Bool
    -- | 'True' if the argument is an IEEE floating point number
    isIEEE              :: a -> Bool
    -- | a version of arctangent taking two real floating-point arguments.
    -- For real floating @x@ and @y@, @'atan2' y x@ computes the angle
    -- (from the positive x-axis) of the vector from the origin to the
    -- point @(x,y)@.  @'atan2' y x@ returns a value in the range [@-pi@,
    -- @pi@].  It follows the Common Lisp semantics for the origin when
    -- signed zeroes are supported.  @'atan2' y 1@, with @y@ in a type
    -- that is 'RealFloat', should return the same value as @'atan' y@.
    -- A default definition of 'atan2' is provided, but implementors
    -- can provide a more accurate implementation.
    atan2               :: a -> a -> a
```

La classe RealFloat (Implémentation 4)

```
exponent x           =  if m == 0 then 0 else n + floatDigits x
                        where (m,n) = decodeFloat x

significand x        =  encodeFloat m (negate (floatDigits x))
                        where (m,_) = decodeFloat x

scaleFloat 0 x       =  x
scaleFloat k x
  | isFix            =  x
  | otherwise        =  encodeFloat m (n + clamp b k)
                        where (m,n) = decodeFloat x
                              (l,h) = floatRange x
                              d     = floatDigits x
                              b     = h - l + 4*d
                              isFix = x == 0 || isNaN x || isInfinite x


atan2 y x
  | x > 0                =  atan (y/x)
  | x == 0 && y > 0      =  pi/2
  | x <  0 && y > 0      =  pi + atan (y/x)
  |(x <= 0 && y < 0)                 ||
   (x <  0 && isNegativeZero y) ||
   (isNegativeZero x && isNegativeZero y)
                         = - atan2 (- y) x
  | y == 0 && (x < 0 || isNegativeZero x)
                          =  pi    -- must be after the previous test on zero y
  | x==0 && y==0          =  y     -- must be after the other double zero tests
  | otherwise             =  x + y -- x or y is a NaN, return a NaN (via +)
```

La classe Integral

- Permet d'effectuer des opérations sur les entiers.
- Définit les fonctions quot, rem, div, mod, quotRem, divMod, toInteger.
- Peut être étendue à tous les types appartenant aux classe Real et Enum avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à quotRem et à toInteger :

La classe Integral (Implémentation)

```haskell
class  (Real a, Enum a) => Integral a  where
    -- | integer division truncated toward zero
    quot                :: a -> a -> a
    -- | integer remainder, satisfying
    --
    -- > (x `quot` y)*y + (x `rem` y) == x
    rem                 :: a -> a -> a
    -- | integer division truncated toward negative infinity
    div                 :: a -> a -> a
    -- | integer modulus, satisfying
    --
    -- > (x `div` y)*y + (x `mod` y) == x
    mod                 :: a -> a -> a
    -- | simultaneous 'quot' and 'rem'
    quotRem             :: a -> a -> (a,a)
    -- | simultaneous 'div' and 'mod'
    divMod              :: a -> a -> (a,a)
    -- | conversion to 'Integer'
    toInteger           :: a -> Integer

    n `quot` d          =  q  where (q,_) = quotRem n d
    n `rem` d           =  r  where (_,r) = quotRem n d
    n `div` d           =  q  where (q,_) = divMod n d
    n `mod` d           =  r  where (_,r) = divMod n d

    divMod n d          =  if signum r == negate (signum d) then (q-1, r+d) else qr
                           where qr@(q,r) = quotRem n d
```

La classe Floating

- Permet d'effectuer des opérations sur les nombres à virgule flottantes.
- Définit les fonctions :
  - Nombre pi.
  - Exponentielles exp, (**).
  - Logarithmiques log, logBase.
  - Racines sqrt.
  - Trigonométriques sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.
- Peut être étendue à tous les types appartenant à la classe Fractional avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh, asinh, acosh et à atanh.

La classe Floating (Implémentation 1)

```haskell
class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

    x ** y              =  exp (log x * y)
    logBase x y         =  log y / log x
    sqrt x              =  x ** 0.5
    tan  x              =  sin  x / cos  x
    tanh x              =  sinh x / cosh x
```

La classe Floating (Implémentation 2)

```haskell
    -- | @'log1p' x@ computes @'log' (1 + x)@, but provides more precise
    -- results for small (absolute) values of @x@ if possible.
    log1p                :: a -> a

    -- | @'expm1' x@ computes @'exp' x - 1@, but provides more precise
    -- results for small (absolute) values of @x@ if possible.
    expm1                :: a -> a

    -- | @'log1pexp' x@ computes @'log' (1 + 'exp' x)@, but provides more
    -- precise results if possible.
    log1pexp             :: a -> a

    -- | @'log1mexp' x@ computes @'log' (1 - 'exp' x)@, but provides more
    -- precise results if possible.
    log1mexp             :: a -> a

    log1p x = log (1 + x)
    expm1 x = exp x - 1
    log1pexp x = log1p (exp x)
    log1mexp x = log1p (negate (exp x))
```

La classe Enum

- Permet d'effectuer des opérations sur les nombres à virgule flottantes.
- Définit les fonctions succ, prec, toEnum, fromEnum, enumFrom, enumFromThen, enumFromTo et enumFromThenTo.
- Peut être étendue à tous les types appartenant à la classe Fractional avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à toEnum et à fromEnum.

La classe Enum (Implémentation 1)

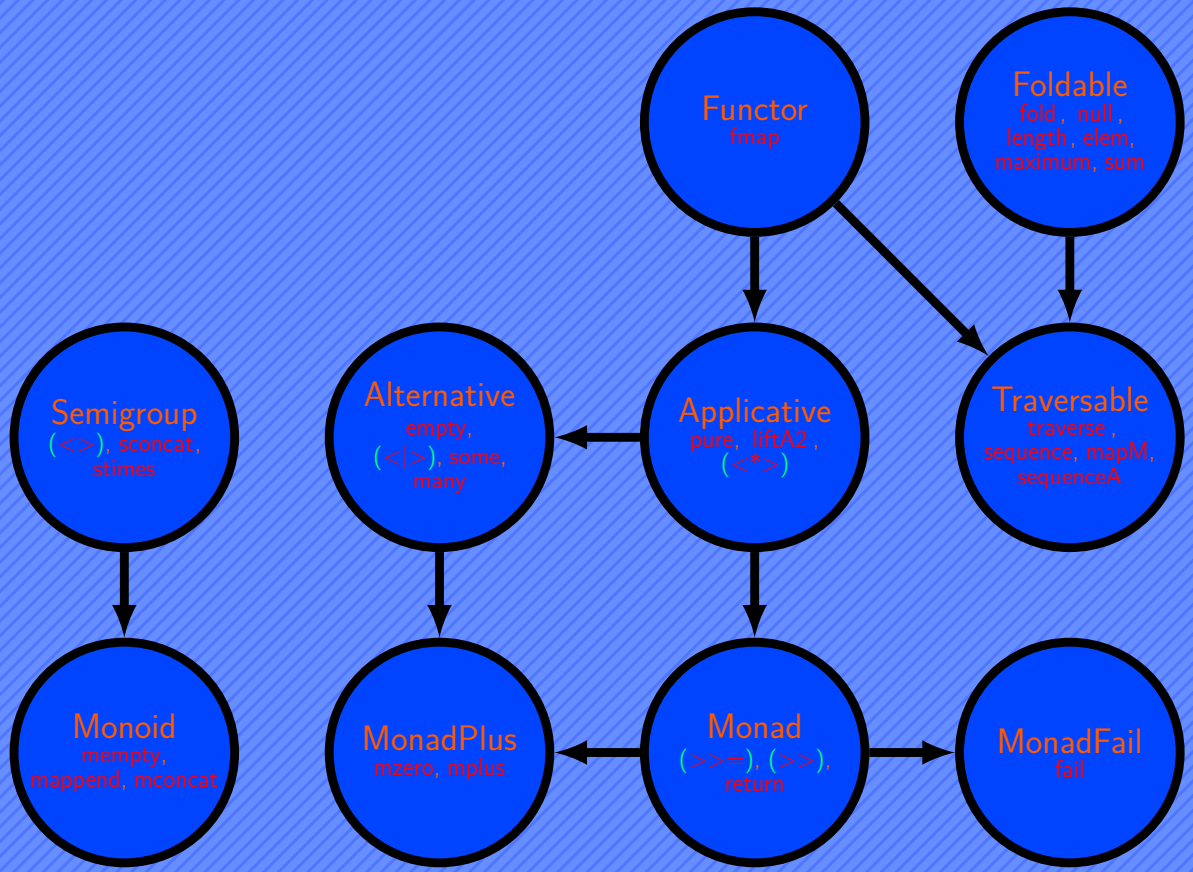
```
class  Enum a   where
    -- | the successor of a value.  For numeric types, 'succ' adds 1.
    succ                :: a -> a

    -- | the predecessor of a value.  For numeric types, 'pred' subtracts 1.
    pred                :: a -> a

    -- | Convert from an 'Int'.
    toEnum              :: Int -> a

    -- | Convert to an 'Int'.
    -- It is implementation-dependent what 'fromEnum' returns when
    -- applied to a value that is too large to fit in an 'Int'.
    fromEnum            :: a -> Int

    -- | Used in Haskell's translation of @[n..]@ with @[n..] = enumFrom n@,
    --   a possible implementation being @enumFrom n = n : enumFrom (succ n)@.
    --   For example:
    enumFrom            :: a -> [a]

    -- | Used in Haskell's translation of @[n,n'..]@
    --   with @[n,n'..] = enumFromThen n n'@, a possible implementation being
    --   @enumFromThen n n' = n : n' : worker (f x) (f x n')@,
    enumFromThen        :: a -> a -> [a]
```

La classe Enum (Implémentation 2)

```haskell
    -- | Used in Haskell's translation of @[n..m]@ with
    --   @[n..m] = enumFromTo n m@, a possible implementation being
    enumFromTo          :: a -> a -> [a]

    -- | Used in Haskell's translation of @[n,n'..m]@ with
    --   @[n,n'..m] = enumFromThenTo n n' m@, a possible implementation
    --   being @enumFromThenTo n n' m = worker (f x) (c x) n m@,
    --   @x = fromEnum n' - fromEnum n@, @c x = bool (>=) (<=) (x > 0)@
    --   @f n y
    --      | n > 0 = f (n - 1) (succ y)
    --      | n < 0 = f (n + 1) (pred y)
    --      | otherwise = y@ and
    --   @worker s c v m
    --      | c v m = v : worker s c (s v) m
    --      | otherwise = []@
    --   For example:
    --
    --     * @enumFromThenTo 4 2 -6 :: [Integer] = [4,2,0,-2,-4,-6]@
    --     * @enumFromThenTo 6 8 2 :: [Int] = []@
    enumFromThenTo      :: a -> a -> a -> [a]

    succ                    = toEnum . (+ 1)  . fromEnum
    pred                    = toEnum . (subtract 1) . fromEnum
    enumFrom x              = map toEnum [fromEnum x ..]
    enumFromThen x y        = map toEnum [fromEnum x, fromEnum y ..]
    enumFromTo x y          = map toEnum [fromEnum x .. fromEnum y]
    enumFromThenTo x1 x2 y = map toEnum [fromEnum x1, fromEnum x2 .. fromEnum y]
```

Les classes monadiques

Organisation des classes monadiques



Les Foncteurs

- Un foncteur peut être vu comme un "conteneur" d'un certain type et la capacité d'appliquer une fonction à chaque élément du conteneur.
- Par conteneur on peut penser à :
  - Une liste.
  - Un arbre.
  - ...
- Un foncteur peut aussi être vu comme un "contexte de calcul".

La classe Functor

- Permet de "mapper" un type a vers un type b via une fonction (a → b) .
- Définit les fonctions fmap et (<$).
- Peut être étendue à tous les types avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à fmap.

La classe Functor (Implémentation)

```haskell
class  Functor f  where
    fmap        :: (a -> b) -> f a -> f b

    -- | Replace all locations in the input with the same value.
    -- The default definition is @'fmap' . 'const'@, but this may be
    -- overridden with a more efficient version.
    (<$)        :: a -> f b -> f a
    (<$)        =  fmap . const
```

Les Foncteurs applicatifs

- Type particulier de foncteur.
- Possède plus de fonctions qu'un foncteur mais moins qu'une monade.
- Permet de générer du code plus générale que les monades (plus de foncteurs applicatifs que de monades).
- Permet d'utiliser la programmation fonctionnelle plutôt que impérative et séquentielle (monades).

La classe Applicative

- Permet de contenir une valeur "pure",d'effectuer des sequences de calculs et de combiner les résultats.
- Définit les fonctions pure, (<*>), liftA2 , (*>) et (<*).
- Peut être étendue à tous les types appartenant à la classe Functor avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à pure et :
  - soit à (<*>).
  - soit à liftA2 .

La classe Applicative (Implémentation)

```haskell
class Functor f => Applicative f where
    -- | Lift a value.
    pure :: a -> f a

    -- | Sequential application.
    --
    -- A few functors support an implementation of '<*>' that is more
    -- efficient than the default one.
    (<*>) :: f (a -> b) -> f a -> f b
    (<*>) = liftA2 id

    -- | Lift a binary function to actions.
    --
    -- Some functors support an implementation of 'liftA2' that is more
    -- efficient than the default one. In particular, if 'fmap' is an
    -- expensive operation, it is likely better to use 'liftA2' than to
    -- 'fmap' over the structure and then use '<*>'.
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
    liftA2 f x = (<*>) (fmap f x)

    -- | Sequence actions, discarding the value of the first argument.
    (*>) :: f a -> f b -> f b
    a1 *> a2 = (id <$ a1) <*> a2

    -- | Sequence actions, discarding the value of the second argument.
    (<*) :: f a -> f b -> f a
    (<*) = liftA2 const
```

Les monades

- Une monade peut être vu comme une "boite" pouvant contenir un ou plusieurs objets de même nature.
- Possibilité de placer des éléments dans la "boite" et de les ressortir ensuite.
- Une monade autorise la programmation séquentielle.

La classe Monad

- Permet d'effectuer des actions de manière séquentielle et de retourner un résultat.
- Définit les fonctions (>>=), (>>) et return.
- Peut être étendue à tous les types appartenant à la classe Applicative avec les instances.
- Pour être étendue, il est nécessaire de donner au moins une définition à (>>=).

La classe Monad (Implémentation)

```haskell
class Applicative m => Monad m where
    -- | Sequentially compose two actions, passing any value produced
    -- by the first as an argument to the second.
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b

    -- | Sequentially compose two actions, discarding any value produced
    -- by the first, like sequencing operators (such as the semicolon)
    -- in imperative languages.
    (>>)        :: forall a b. m a -> m b -> m b
    m >> k = m >>= \_ -> k -- See Note [Recursive bindings for Applicative/Monad]

    -- | Inject a value into the monadic type.
    return      :: a -> m a
    return      = pure
```

La classe MonadPlus

- Permet d'utiliser des monades avec des choix.
- Définit les fonctions mzero et mplus.
- Peut être étendue à tous les types appartenant aux classes Monad et Alternative avec les instances.
- Pour être étendue, il n'est pas nécessaire de donner une définition à l'une ou l'autre des fonctions.

La classe MonadPlus (Implémentation)

```haskell
class (Alternative m, Monad m) => MonadPlus m where
   -- | The identity of 'mplus'.  It should also satisfy the equations
   --
   -- > mzero >>= f  =  mzero
   -- > v >> mzero   =  mzero
   --
   mzero :: m a
   mzero = empty

   -- | An associative operation.
   --
   mplus :: m a -> m a -> m a
   mplus = (<|>)
```

La classe MonadFail

- Permet d'apporter la fonction d'échec à des monades.
- Définit la fonction `fail` .
- Peut être étendue à tous les types appartenant à la classe Monad avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à `fail` .

La classe MonadFail (Implémentation)

```haskell
class Monad m => MonadFail m where
    fail :: String -> m a
```

Les Monoïdes

En mathématiques, un monoïde est une structure algébrique utilisée en algèbre générale, définie comme un ensemble muni :

- d'une loi de composition interne associative.
- d'un élément neutre.

Permettent de répondre à une problématique récurrente en programmation qui est de combiner deux éléments du même type (composition) et de renvoyer comme résultat un élément du même type : $(a \rightarrow a \rightarrow a)$

La classe Monoid

- Permet d'apporter une fonction de composition.
- Définit la fonction mempty, mappend et mconcat.
- Peut être étendue à tous les types appartenant à la classe Semigroup avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à mempty.

La classe Monoid (Implémentation)

```haskell
class Semigroup a => Monoid a where
        -- | Identity of 'mappend'
        mempty  :: a

        -- | An associative operation
        mappend :: a -> a -> a
        mappend = (<>)

        -- | Fold a list using the monoid.
        --
        -- For most types, the default definition for 'mconcat' will be
        -- used, but the function is included in the class definition so
        -- that an optimized version can be provided for specific types.
        mconcat :: [a] -> a
        mconcat = foldr mappend mempty
```

Les Semi-Groupes

En mathématiques, un semi-groupe est une structure algébrique constituée d'un ensemble muni d'une loi de composition interne associative.

La classe Semigroup

- Classe associée a des types pouvant être composés.
- Définit les fonctions (<>), sconcat et stimes.
- Peut être étendue à tous les types avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à (<>).

La classe Semigroup (Implémentation)

```haskell
class Semigroup a where
        -- | An associative operation.
        (<>) :: a -> a -> a

        -- | Reduce a non-empty list with '<>'
        --
        -- The default definition should be sufficient, but this can be
        -- overridden for efficiency.
        --
        sconcat :: NonEmpty a -> a
        sconcat (a :| as) = go a as where
          go b (c : cs) = b <> go c cs
          go b []     = b

        -- | Repeat a value @n@ times.
        --
        -- Given that this works on a 'Semigroup' it is allowed to fail if
        -- you request 0 or fewer repetitions, and the default definition
        -- will do so.
        stimes :: Integral b => b -> a -> a
        stimes = stimesDefault
```

La classe Alternative

- Classe associée a des structures pouvant supporter des alternatives.
- Définit les fonctions empty, (<|>), some et many.
- Peut être étendue à tous les types de la classe Applicative avec les instances.
- Pour être étendue, il est nécessaire de donner une définition à empty et à (<|>).

La classe Alternative (Implémentation)

```haskell
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a
    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a

    -- | One or more.
    some :: f a -> f [a]
    some v = some_v
      where
        many_v = some_v <|> pure []
        some_v = liftA2 (:) v many_v

    -- | Zero or more.
    many :: f a -> f [a]
    many v = many_v
      where
        many_v = some_v <|> pure []
        some_v = liftA2 (:) v many_v
```

La classe Traversable

- Classe associée a des structures de données pouvant être parcourus de gauche à droite.
- Définit les fonctions traverse, sequenceA, mapM et sequence.
- Peut être étendue à tous les types appartenant aux classes Functor et Foldable avec les instances.
- Pour être étendue, il est nécessaire de donner une définition :
  - Soit à traverse.
  - Soit à sequenceA.

La classe Traversable (Implémentation)

```haskell
class (Functor t, Foldable t) => Traversable t where
    -- | Map each element of a structure to an action, evaluate these actions
    -- from left to right, and collect the results. For a version that ignores
    -- the results see 'Data.Foldable.traverse_'.
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    -- | Evaluate each action in the structure from left to right, and
    -- collect the results. For a version that ignores the results
    -- see 'Data.Foldable.sequenceA_'.
    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    -- | Map each element of a structure to a monadic action, evaluate
    -- these actions from left to right, and collect the results. For
    -- a version that ignores the results see 'Data.Foldable.mapM_'.
    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    -- | Evaluate each monadic action in the structure from left to
    -- right, and collect the results. For a version that ignores the
    -- results see 'Data.Foldable.sequence_'.
    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

La classe Foldable

- Classe associée a des structures de données pouvant être "plié" ("réduite").
- Définit les fonctions de pliage foldMap, fold, foldr, foldl ...
- Définit des fonctions de tests null, length, elem.
- Définit des fonctions de conversion en liste toList.
- Définit des fonctions de "pliage" spécifique maximum, minimum, sum et product.
- Peut être étendue à tous les types avec les instances.
- Pour être étendue, il est nécessaire de donner une définition :
  - Soit à foldMap.
  - Soit à foldr.

La classe Foldable (Implémentation 1)

```haskell
class Foldable t where
  -- | Combine the elements of a structure using a monoid.
  fold :: Monoid m => t m -> m
  fold = foldMap id

  -- | Map each element of the structure to a monoid, and combine the results.
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap f = foldr (mappend . f) mempty

  -- | A variant of 'foldMap' that is strict in the accumulator.
  -- @since 4.13.0.0
  foldMap' :: Monoid m => (a -> m) -> t a -> m
  foldMap' f = foldl' (\ acc a -> acc <> f a) mempty

  -- | Right-associative fold of a structure.
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo #. f) t) z

  -- | Right-associative fold of a structure, but with strict application of the
      operator.
  -- @since 4.6.0.0
  foldr' :: (a -> b -> b) -> b -> t a -> b
  foldr' f z0 xs = foldl f' id xs z0
    where f' k x z = k $! f x z
```

La classe Foldable (Implémentation 2)

```haskell
-- | Left-associative fold of a structure.
foldl :: (b -> a -> b) -> b -> t a -> b
foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z

-- | A variant of 'foldr' that has no base case,
-- and thus may only be applied to non-empty structures.
foldr1 :: (a -> a -> a) -> t a -> a
foldr1 f xs = fromMaybe (errorWithoutStackTrace "foldr1: empty structure") (foldr
     mf Nothing xs)
  where
    mf x m = Just (case m of
                     Nothing -> x
                     Just y  -> f x y)

-- | A variant of 'foldl' that has no base case,
-- and thus may only be applied to non-empty structures.
foldl1 :: (a -> a -> a) -> t a -> a
foldl1 f xs = fromMaybe (errorWithoutStackTrace "foldl1: empty structure")
                (foldl mf Nothing xs)
  where
    mf m y = Just (case m of
                     Nothing -> y
                     Just x  -> f x y)
```

La classe Foldable (Implémentation 3)

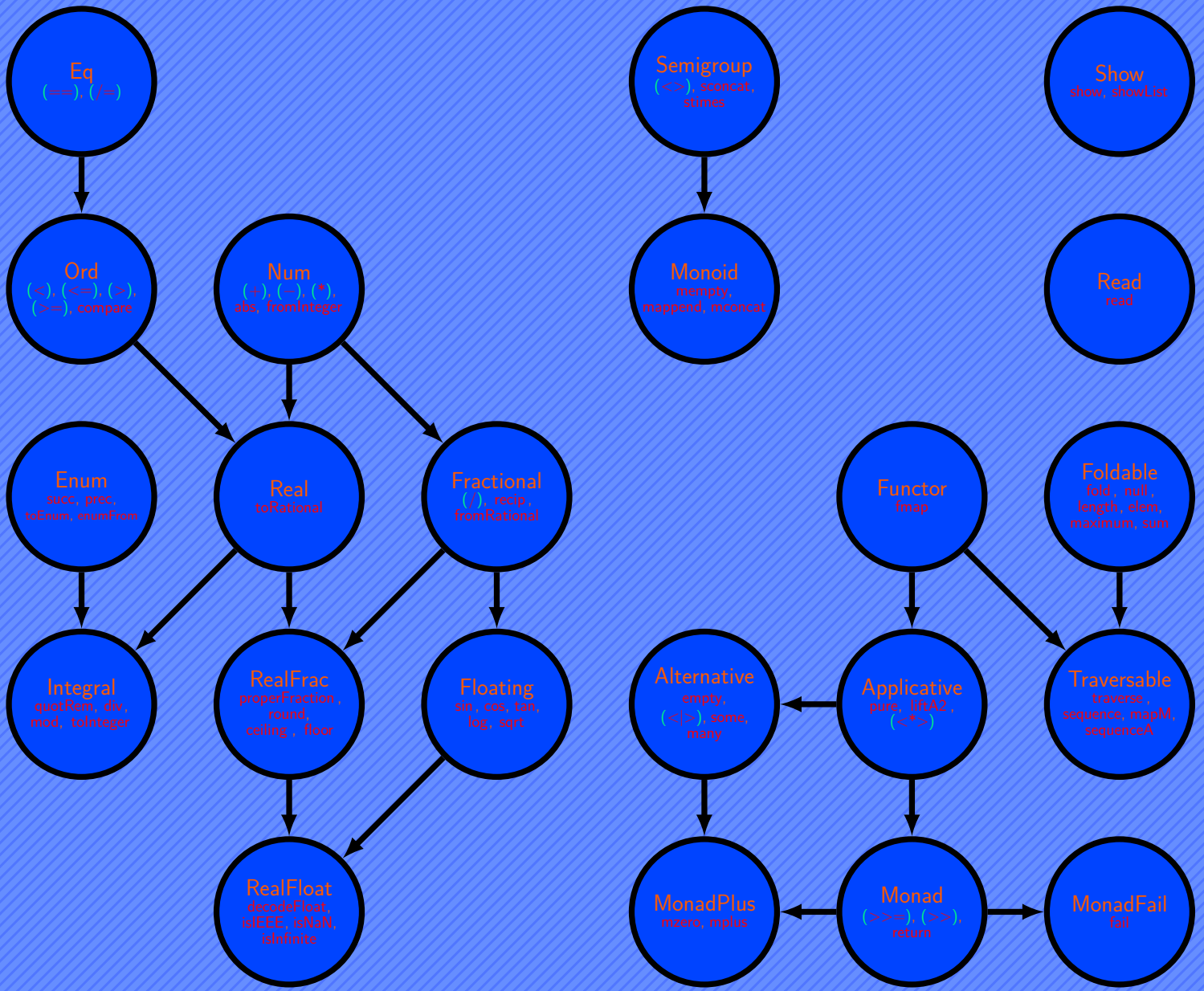
```haskell
  -- | List of elements of a structure, from left to right.
  -- @since 4.8.0.0

  toList :: t a -> [a]
  toList t = build (\ c n -> foldr c n t)
  -- | Test whether the structure is empty.
  -- @since 4.8.0.0

  null :: t a -> Bool
  null = foldr (\_ _ -> False) True
  -- | Returns the size/length of a finite structure as an 'Int'.
  -- @since 4.8.0.0

  length :: t a -> Int
  length = foldl' (\c _ -> c + 1) 0
  -- | Does the element occur in the structure?
  -- @since 4.8.0.0

  elem :: Eq a => a -> t a -> Bool
  elem = any . (==)
```

La classe Foldable (Implémentation 4)

```haskell
-- | The largest element of a non-empty structure.
-- @since 4.8.0.0
maximum :: forall a. Ord a => t a -> a
maximum = fromMaybe (errorWithoutStackTrace "maximum: empty structure") . getMax .
      foldMap (Max #. (Just :: a -> Maybe a))

-- | The least element of a non-empty structure.
-- @since 4.8.0.0
minimum :: forall a. Ord a => t a -> a
minimum = fromMaybe (errorWithoutStackTrace "minimum: empty structure") . getMin .
      foldMap (Min #. (Just :: a -> Maybe a))

-- | The 'sum' function computes the sum of the numbers of a structure.
-- @since 4.8.0.0
sum :: Num a => t a -> a
sum = getSum #. foldMap Sum

-- | The 'product' function computes the product of the numbers of a structure.
-- @since 4.8.0.0
product :: Num a => t a -> a
product = getProduct #. foldMap Product
```

Vue d'ensemble des classes

Organisation des classes Haskell standards



Coordonnées

Coordonnées

Jean-Luc JOULIN
jean-luc-joulin@orange.fr
www.jeanjoux.fr