Chapter 1

Clase de tipuri

Un rol important în Haskell îl joacă clasele de tipuri.

O clasă de tipuri (engl. typeclass) este o mulțime de tipuri care au câteva funcții în comun.

Atenție! O clasa de tipuri nu este un tip.

Există o serie de clase de tipuri predefinite în biblioteca standard.

1.1 Clasa `Eq`

Clasa `Eq` este clasa tipurilor de date ale căror valori pot fi comparate. Putem afla mai multe informații despre această clasă de tipuri folosind comanda `:info`:

```
*Main> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
   -- Defined in ‘GHC.Classes’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
instance Eq Word -- Defined in ‘GHC.Classes’
instance Eq Ordering -- Defined in ‘GHC.Classes’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Eq Float -- Defined in ‘GHC.Classes’
[...]
```

Linia `class Eq a where` se citește “tipul `a` face parte din clasa `Eq` dacă” (sau “tipul `a` este instanță a clasei `Eq` dacă”), iar liniile ce urmează

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

conțin funcțiile care trebuie să fie definite pe tipul `a` în acest sens.

În exemplul de mai sus, `a` face parte din clasa `Eq` dacă sunt definite funcțiile `(==) :: a -> a -> Bool` (sunt două valori de tip `a` egale?) și `(/=) :: a -> a -> Bool` (sunt două valori de tip `a` diferite?).

Vom înțelege linia `[...] MINIMAL (==) | (/=) [...]` mai târziu.

În continuare sunt prezentate tipurile care sunt instanță a clasei `Eq`: `Word`, `Ordering`, `Int`, `Float` ș.a.m.d. De asemenea, linia `instance Eq a => Eq [a]` marchează că tipul `[a]` (listă de valori de tip `a`) este instanță a clasei `Eq` dacă tipul `a` este instanță a clasei `Eq`.

Pentru a face ca un tip de date să devină instanță a clasei `Eq`, este suficient să adăugăm clauza `deriving Eq` la sfârșitul declarației tipului:

```
data Nat = Zero | Succ Nat deriving Eq
```

Această clauză va genera funcții implicite de testare a egalității și disegalității care se bazează pe constructori: orice două valori care încep cu constructuri diferiți sunt diferite, iar orice două valori care încep cu același constructor sunt egale ddacă argumentele de pe fiecare poziție în parte sunt la rândul lor egale.

Câteodată, această definiție a egalității nu este potrivită, deoarece am putea defini un tip de date cu două reprezentări diferite pentru aceeași construcție semantică:

```
data Z = Pos Nat | Neg Nat

zero1 = Pos Zero
zero2 = Neg Zero
```

În exemplul de mai sus, definim tipul numerelor întregi `Z`, iar numărul 0 are două reprezentări (definite în `zero1` și `zero2`). Pentru acest caz, este utilă implementarea individualizată a testului de egalitate, care se poate face folosind următoarea sintaxă:

```
instance Eq Z where
  (==) (Pos x) (Pos y) = (==) x y
  (==) (Neg x) (Neg y) = (==) x y
  (==) (Pos x) (Neg y) = (x == Zero) && (y == Zero)
  (==) (Neg x) (Pos y) = (x == Zero) && (y == Zero)
```

Nu este nevoie de declararea explicită a ambelor funcții (`(==)` și `(/=)`), deoarece sistemul generează automat o implementare implicită pentru `(/=)` din `(==)` (și invers, ar putea genera `(==)` din `(/=)`). Acest lucru este marcat în linia `[...] MINIMAL (==) | (/=) [...]`, care indică că este suficient de implementat oricare dintre cele două funcții pentru a instanția clasa.

Clasa `Eq` este definită în biblioteca standard, iar utilizatorul ar putea defini o clasă similară folosind următoarea sintaxă:

```
class MyEq a where
  equals :: a -> a -> Bool
  notEquals :: a -> a -> Bool
  equals x y = not (notEquals x y)
  notEquals x y = not (equals x y)

instance MyEq Nat where
  equals Zero Zero = True
  equals Zero (Succ _) = False
  equals (Succ _) Zero = False
  equals (Succ x) (Succ y) = equals x y
  -- notEquals Zero Zero = False
  -- notEquals (Succ _) Zero = True
  -- notEquals Zero (Succ _) = True
  -- notEquals (Succ x) (Succ y) = notEquals x y
```

Observați cum `equals` este definită în termeni de `notEquals` și invers. Din acest motiv, este suficient de particularizat doar una dintre cele două funcții pentru a instanția clasa.

În schimb, clauza `deriving` nu poate fi folosită decât pentru clasele predefinite.

1.2 Clasa `Ord`

Clasa `Ord` este clasa tipurilor ale căror valori pot fi ordonate:

```
> :i Ord
:i Ord
type Ord :: * -> Constraint
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
   -- Defined in ‘GHC.Classes’
instance Ord a => Ord [a] -- Defined in ‘GHC.Classes’
instance Ord Word -- Defined in ‘GHC.Classes’
instance Ord Ordering -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
instance Ord Float -- Defined in ‘GHC.Classes’
instance Ord Double -- Defined in ‘GHC.Classes’
[...]
> :i Ordering
type Ordering :: *
data Ordering = LT | EQ | GT
```

Pentru a instanția clasa, este suficientă definirea funcției `compare` sau a funcției `(<=)`.

Sintaxa `class Eq a => Ord a where` marchează că orice tip instanță a lui `Ord` este în mod necesar și instanță a lui `Eq`.

Iată cum se poate instanția clasa `Ord` pentru tipul `Z` definit mai sus:

```
instance Ord Z where
  (<=) (Neg _) (Pos _) = True
  (<=) (Neg x) (Neg y) = y <= x
  (<=) (Pos x) (Pos y) = x <= y
  (<=) (Pos x) (Neg y) = (x == Zero) && (y == Zero)

> Pos Zero <= Neg Zero
True
> Neg Zero <= Pos Zero
True
> Neg Zero <= Neg Zero
True
> Pos Zero <= Pos Zero
True
> Pos two <= Pos three
True
> Pos three <= Pos two
False
```

```
> Pos three <= Pos three
True
> Neg three <= Neg three
True
> Neg three <= Neg two
True
> Neg two <= Neg three
False
```

1.3 Clasa `Bounded`

Clasa `Bounded` este clasa tipurilor mărginite.

```
> :i Bounded
type Bounded :: * -> Constraint
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
   -- Defined in ‘GHC.Enum’
instance [safe] Bounded Dow -- Defined at curs5.hs:65:83
instance Bounded Word -- Defined in ‘GHC.Enum’
instance Bounded Ordering -- Defined in ‘GHC.Enum’
instance Bounded Int -- Defined in ‘GHC.Enum’
instance Bounded Char -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
[...]
```

Tipurile mărginite pot instanția `Bounded` folosind sintaxa:

```
data Dow = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Eq, Show, Read, Ord, Bounde
```

Funcțiile `minBound` și `maxBound` pot fi

```
> minBound :: Dow
Mon
> maxBound :: Dow
Sun
> minBound :: Word
0
> maxBound :: Word
18446744073709551615
> 2^63
9223372036854775808
> 2^64
18446744073709551616
> 2^64 - 1
18446744073709551615
> maxBound :: Word
18446744073709551615
```

1.4 Clasa `Functor`

1.4.1 Kind-uri

În Haskell, tipurile sunt organizate în *kind*-uri.

Un kind este pentru un tip ceea ce este tipul pentru o valoare.

De exemplu, `3` este de tip `Int`, iar `Int` este de kind `*`.

Mai mult sau mai puțin, kind-ul este tipul unui tip.

Cel mai simplu kind este `*` (citit: stea/star).

Tipurile de kind `*` sunt tipuri *concrete*: `Int`, `Integer`, `Bool`, `[Bool]`.

Putem afla kind-ul unui tip folosind comanda `:kind`:

```
> :k Int
Int :: *
> :k Integer
Integer :: *
> :k Word
Word :: *
> :k Dow
Dow :: *
> :k Maybe
Maybe :: * -> *
> :k (Maybe Int)
(Maybe Int) :: *
> :k []
[] :: * -> *
> :k [Int]
[Int] :: *
```

Observați că `Maybe` și `[]` (listă) au kind-ul `* -> *`, adică sunt *constructori de tip*: primesc un tip concret ca argument și întorc un tip concret.

Kind-ul unui tip este afișat și când folosiți comanda `:info`:

```
*Main> :i Maybe
type Maybe :: * -> *
data Maybe a = Nothing | Just a
[...]
```

1.4.2 `Functor`

Până acum, am văzut doar clase de tipuri de kind `*`.

Un tip de kind `* -> *` face parte din clasa `Functor` dacă:

```
> :i Functor
:i Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
   -- Defined in ‘GHC.Base’
```

Cu alte cuvinte, `f` este `Functor` dacă există o funcție `fmap` care transformă o funcție de la `a` la `b` într-o funcție de la `f a` la `f b`.

Funcția `fmap` poate fi privită ca o generalizare a funcției `map` de la liste la orice tip care reprezintă un container.

```
[...]
instance Functor (Either a) -- Defined in ‘Data.Either’
instance Functor [] -- Defined in ‘GHC.Base’
instance Functor Maybe -- Defined in ‘GHC.Base’
[...]
```

Iată cum poate fi folosită pentru liste, `Maybe`-uri și `Either`-uri:

```
> fmap (\x -> x + 1) [1,2,3]
[2,3,4]
> fmap (\x -> x + 1) Nothing
Nothing
> fmap (\x -> x + 1) (Just 123)
Just 124
> fmap (\x -> x + 1) (Left 123)
Left 123
> fmap (\x -> x + 1) (Right 123)
Right 124
```

Când definim containere proprii (e.g., arbori), este util să instanțiem clasa `Functor` pentru a permite maparea unei funcții peste containerul definit folosind `fmap`.

1.5 Analogie cu alte limbaje

Atenție, o clasă de tipuri în Haskell (mulțime de tipuri) nu seamănă cu o clasă în C++ (un tip). În schimb, seamănă cu noțiunea de concept în C++ (noțiunea de concept în C++ e inspirată din noțiunea de clasă de tipuri).

De asemenea, există o oarecare similaritate între clasele de tipuri în Haskell și interfețele din Java (sau clasele abstracte în C++).