Polymorphism – Overview (1A)

Young Won Lim
2/20/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

General Monad - MonadPlus

Haskell's **Control.Monad** module defines a typeclass, **MonadPlus**,
that enables abstract the common pattern eliminating **case** expressions.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

class (Monad m) => MonadPlus m where

```
instance MonadPlus [] where
  mzero = []
  Mplus = (++)
```

```
instance MonadPlus Maybe where
  mzero = Nothing

  Nothing `mplus` ys  = ys
  xs      `mplus` _ = xs
```

http://book.realworldhaskell.org/read/programming-with-monads.html

General Monad - MonadPlus Laws

The class **MonadPlus** is used for monads that have a zero element and a plus operation:

```
class (Monad m) => MonadPlus m  where
    mzero          :: m a
    mplus          :: m a -> m a -> m a
```

For lists, the zero value is [], the empty list.
The I/O monad has no zero element and
is not a member of this class.

```
m >>= \x -> mzero      =      mzero
mzero >>= m            =      mzero
```

The zero element laws:

```
m `mplus` mplus        =      m
mplus `mplus` m        =      m
```

The laws governing the mplus operator

The mplus operator is ordinary list concatenation in the list monad.

http://book.realworldhaskell.org/read/programming-with-monads.html

Vector, Matrix : addition and subtraction

```
data Vector = Vector Int Int deriving (Eq, Show)
data Matrix = Matrix Vector Vector deriving (Eq, Show)
```

overloading Haskell's Num class:

```
instance Num Vector where
  Vector a1 b1 + Vector a2 b2 = Vector (a1+a2) (b1+b2)
  Vector a1 b1 - Vector a2 b2 = Vector (a1-a2) (b1-b2)
  {- ... and so on ... -}
```

```
instance Num Matrix where
  Matrix a1 b1 + Matrix a2 b2 = Matrix (a1+a2) (b1+b2)
  Matrix a1 b1 - Matrix a2 b2 = Matrix (a1-a2) (b1-b2)
  {- ... and so on ... -}
```

https://wiki.haskell.org/Functional_dependencies

Vector, Matrix : multiplication

need a multiplication function which overloads to different types:

```
(*) :: Matrix -> Matrix -> Matrix
(*) :: Matrix -> Vector -> Vector
(*) :: Matrix -> Int -> Matrix
(*) :: Int -> Matrix -> Matrix
{- ... and so on ... -}
```

Too many cumbersome
{- ... and so on ... -}

```
class Mult a b c where
  (*) :: a -> b -> c
```

```
instance Mult Matrix Matrix Matrix where
  {- ... -}
```

```
instance Mult Matrix Vector Vector where
  {- ... -}
```

https://wiki.haskell.org/Functional_dependencies

Vector, Matrix : the result type

even a simple expression has an ambiguous type
unless you supply an additional type declaration on the intermediate expression:

```
m1, m2, m3 :: Matrix
(m1 * m2) * m3              -- type error;
(m1 * m2) :: Matrix * m3    -- this is ok
```

type of (m1*m2) is ambiguous

```
instance Mult Matrix Matrix (Maybe Char) where
  {- whatever -}
```

The problem is that
the third shouldn't really be a free type variable.
When you know the types of multiplicand and multiplier,
the result type should be determined
by the types of those things to be multiplied

https://wiki.haskell.org/Functional_dependencies

Functional Dependency (fundep)

```
class class Mult | a b -> c where
  (*) :: a -> b -> c
```

c is uniquely determined from **a** and **b**

.

Fundeps are not standard Haskell 98.
(Nor are multi-parameter type classes, for that matter.)
They are, however, supported at least in GHC and Hugs
and will almost certainly end up in Haskell'.

https://wiki.haskell.org/Functional_dependencies

Functional Dependency | (vertical bar)

```
class Monad m => MonadState s m | m -> s where …
```

functional dependencies

to constrain the parameters of type classes. s and m

s can be determined from m, m → s

so that s can be the return type State s → s

but m can not be the return type

in a multi-parameter type class,

one of the parameters can be determined from the others,

so that the parameter determined by the others can be the return type

but none of the argument types of some of the methods.

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
```

$$m\ a \begin{cases} \text{Maybe } a \\ \text{IO } a \\ \text{ST } a \\ \text{State } s\ a \end{cases}$$

https://stackoverflow.com/questions/23149318/get-put-and-state-in-monadstate

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf