Notes on Type Classes

Author: **Amitabha Sanyal**

October 10, 2003
Draft - not final

1 Introduction

Consider the function `member` defined as

```
member x [] = False
member x (y:ys) = (x == y) || member x ys
```

Question: What is the type of `member`? First guess – `a -> [a] -> Bool`

But then we should be able to call member as `member f [sin, cos, tan]`. Clearly this is incorrect because it requires the `member` to check two functions for equality. Instead, We would like to say `member :: a -> [a] -> Bool` for only those types `a` whose members can be compared for equality. The language of type expressions is extended to enable saying such a thing.

```
member :: (Eq a) => a -> [a] -> Bool
```

To do this we must do two things:

1. Declare a class called `Eq`. Any type belonging to this class should have an operator `==` defined on values of that class. This is done as:

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

   This is called a *class declaration.*

2. After having declared the class called `Eq`, we must populate it with types. This is done with an *instance declaration*:

   ```
     instance Eq Int where
      (==) = primEqInt     // primEqInt is a primitive
   ```

```
instance Eq Char where
 (==) = primEqChar
```

Now suppose we also wanted to add the type `[a]` to `Eq`. Surely this will require the type `a` to be in `Eq`. Thus the instance declaration for `[a]` is

```
instance (Eq a) => Eq [a] where
 (==) [] [] = True
 (==) (x:xs) (y:ys) = (x == y) && (xs == ys)
 (==) _ _ = False
```

The entity `(Eq a)` in `instance (Eq a) => Eq [a]` (and elsewhere) is called a context. Every entry in a context pairs a class name with a class variable. Now the function

```
palindrome xs = (xs == reverse xs)
```

is typed as `palindrome :: (Eq a) => [a] -> Bool` and not `(Eq [a]) => [a] -> Bool` We could also extend the class `Eq a` with a default definition of `\=`.

```
class (Eq a) where
 (==) :: a -> a -> Bool
 (/=) :: a -> a -> Bool
 (/=) x y = not(x == y)
```

2 Superclasses

Now let us introduce a class called `Ord` defined as

```
class (Eq a) => (Ord a) where
 (<) :: a -> a -> Bool
 (<=) :: a -> a -> Bool
```

We could make `Int` an instance of `Ord` as follows

```
instance Ord Int where
 (<) = primLtInt
 (<=) = primLeInt
```

Now, in

```
search x [] = False
search x (y:ys) = x == y || x > y && search x ys
```

The type of search is `(Ord a) => a -> [a] -> Bool` and not `(Eq a, Ord a) => a -> [a] -> Bool`

3 Implementing classes

A *dictionary* is a tuple which contains:

1. The dictionary of its immediate superclasses.

2. The actual function names which implement the operators of the class.

We denote as `dictEqInt` the dictionary of the `Int` instance of `Eq`. Further, if we denote the `Int` instance of `==` as $\texttt{==}_{\texttt{Int}}$, then $\texttt{dictEqInt = < ==}_{\texttt{Int}}\texttt{, /=}_{\texttt{Int}}\texttt{ >}$. The dictionary is created from the instance declaration. Similarly, $\texttt{dictOrdInt = < dictEqInt, >}_{\texttt{Int}}\texttt{, >=}_{\texttt{Int}}\texttt{ >}$. Now the idea is that the overloaded function

```
f x y = ...  x == y ...
```

is rewritten as

`f x y dEq =` 'select the operator `==` from `dEq` and apply it on `x` and `y`'.

and the two calls to it

```
...
f 1 2
...
f '3' '4'
```

are rewritten as

```
...
f 1 2 dictEqInt
...
f '3' '4' dictEqChar
```

To select the right operator, we define the overloaded operator `(==)` appearing in the body of `f` as $\texttt{\textbackslash dEq -> project}_1^1\texttt{ dEq}$. Then the translation of `f x y` becomes

```
f x y dEq = (==) dEq x y
```

Let us see how this works in the case of `f 1 2`

$$
\begin{array}{l}
\texttt{f 1 2 dictEqInt} \\
\texttt{=> (==) dictEqInt 1 2} \\
\quad \texttt{=> ==}_{\texttt{Int}}\texttt{ 1 2} \\
\texttt{=> False}
\end{array}
$$

`Similarly`

f ’3’ ’4’ dictEqChar
=> (==) dictEqChar ’3’ ’4’
=> project$^1_1$ < $==_{\texttt{Char}}$, $/=_{\texttt{Char}}$ > ’3’ ’4’
=> $==_{\texttt{Char}}$ ’3’ ’4’
=> False

The type of f is (Eq a) => a -> a -> Bool. There are two readings of this type expression:

1. f is a -> a -> Bool for all a in class Eq.
2. f needs a dictionary for overloading resolutiuon. f can be implemented if it is provided with a dictionary of the type Eq. The reading becomes apparent in the translation.

For the Ord translation, we have the dictionary <dictEqX, $>_X$, $>=_X$ > for a type X. Further we have the selectors

```
(>) = \dOrd -> project$^3_2$ dOrd}
(>=) = \dOrd -> project$^3_3$ dOrd}
getEqfromOrd = \dOrd -> project$^3_1$ dOrd
```

so that that the translation of

```
g x y = x > y || x == y
```

will be

```
g x y dOrd = (> dOrd x y) || (==) (getEqfromOrd dOrd) x y
```

Another example:

```
search x ys = not (null ys) &&
                (x == head ys ||
                 x < head ys && search x (tail ys))

search :: (Ord a) => a -> [a] -> Bool
```

translates to

```
search x ys dOrd = not(null ys) &&
                 (==(getEqfromOrd) x (head ys) ||
                  < dOrd (head ys) && search x (tail ys) dOrd)
```

Last example:

```
instance (Eq a) => Eq [a] where

(==) xs ys = (null xs) && (null ys) ||
             (not(null xs) && not(null ys)) &&
             head xs == head ys &&
             tail xs == tail ys
```

A dictionary for [a] can be produced, provided a dictionary for a is supplied.

```
dictEqList dEq = < \xs ys ->  (null xs) && (null ys) ||
                              (not(null xs) && not(null ys)) &&
                              (==) dEq (head xs) (head ys) &&
                              (==) (dictEqList dEq) (tail xs) (tail ys) >
```

Once again consider

```
f x y = ...  x == y ...


f x y dEq = (==) dEq x y
```

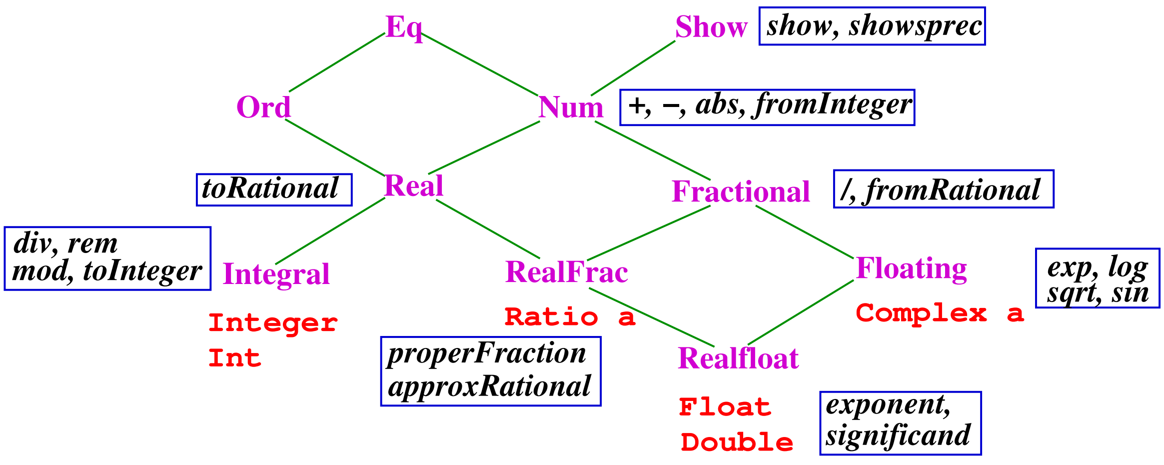
Therefore

```
f [1,2] [3,4]
```

rewrites to

```
   f [1,2] [3,4] (dictEqList dictEqInt)
=> (==) (dictEqList dictEqInt) [1,2] [3,4]
=> (==) <\xs ys ->  (null xs) && (null ys) ||
                    (not(null xs) && not(null ys)) &&
                    (==) dEq (head xs) (head ys) &&
                    (==) (dictEqList dEq) (tail xs) (tail ys) > [1,2] [3,4]
=> (\xs ys ->  (null xs) && (null ys) ||
               (not(null xs) && not(null ys)) &&
               (==) dictEqInt (head xs) (head ys) &&
               (==) (dictEqList dictEqInt) (tail xs) (tail ys)) [1,2] [3,4]
=> (==) dictEqInt 1 3 &&
   (==) (dictEqList dictEqInt) [2] [4]
=> (==Int) 1 3 && (==) (dictEqList dictEqInt) [2] [4]
=> False
```

4 Numeric Classes in Haskell



Number literals

$integer \rightarrow digit\ \{digit\}$

$float \rightarrow integer\ .\ integer[\ e\ [-]\ integer]$

Constructed numbers:

```
  data (Integral a) => Ratio a = a :% a
  data (RealFloat a) => Complex a = a :+ a
type Rational = Ratio Integer
```

a ∈ Num

1. Basic arithmetic operations +, -, abs.
2. a should be obtainable from an Integer.

Num does not need to be under Ord since complex types cannot be compared.

a ∈ Fractional

1. Should support general division (/)
2. a should be obtainable from an Rational. (fromRational)

a ∈ Floating

1. Should support floating point operations exp, log, sqrt, sin, cos, sinh, cosh.

a ∈ Real

1. a should be approximable to a Rational. (toRational).

```
toRational 45.3 = 11875123 % 262144
```

a $\in$ Integral

1. Should support basic integer operations div, rem, mod.

2. a should be approximable to a Integer.

a $\in$ RealFrac

1. Should support functions properFrac, approxRational.

```
properFraction 45.3 = (45,0.299999)
approxRational 45.3 0.1 = 136 % 3
approxRational 45.3 0.01 = 453 % 10
approxRational 45.3 0.000001 = 11875123 % 262144
approxRational 45.3 0.000000001 = 11875123 % 262144
```

a $\in$ RealFloat

1. Should support general division and functions like exponent, significant.

```
exponent 45.3 = 6
significand 45.3 = 0.707812
```

5 Overloading in Numeric Classes

Haskell allows the literal 7 to be regarded as any of Int, Integer, Float, Double, Complex or Ratio.

Similarly 3.4 can be regarded as any of Float, Double, Complex or Ratio.

What 7 really is depends on the context

```
val :: Integer
val = 4 + 7           Here both 4 and 7 are Integers.

val :: Float
val = 4 + 7           Here both 4 and 7 are Floats.
```

The compiler rewrites 7 as fromInteger 7 and 3.4 as fromRational 3.4, where

```
fromInteger :: (Num a) => Integer -> a, and
fromRational :: (Fractional a) => Rational -> a,
```

In other words, fromInteger 7 provides a way of regarding the numeral 7 of the type Integer as any numeric type. Similarly, fromRational 3.4 provides a way of regarding the Rational numeral 3.4 as any non-Integral numeric type.

Each type defines its own instance of fromInteger:

Type	Instance
Int	fromInteger = primIntegertoInt
Integer	fromInteger x = x
Ratio a	fromInteger x = fromInteger x :% 1
Complex a	fromInteger x = fromInteger x :+ 1
Double, Float	fromInteger = encodeFloat x

Similarly

Type	Instance
Ratio a	fromRational (x :% y) = fromRational x :% fromRational y
Complex a	fromRational x = fromInteger x :+ 1
Double, Float	fromRational = rationaltoFloating

6 Unresolved Overloading and Defaults

We shall study a series of examples

1. 2 rewrites to fromInteger 2

```
fromInteger :: (Num a) => Integer -> a
2 :: Integer

fromInteger 2 :: (Num a) => a

translation \dNuma -> (fromInteger dNuma 2)
```

   If this is the entire program then there is no context to resolve the overloading. Ambiguities in the class Num are very common, so Haskell provides a way to resolve them---with a default declaration:

```
default (t1 , ... , tn)
```

   where n >= 0 , and each ti must be a monotype for which Num ti holds.

   Each ambiguous type variable is replaced by the first type in the default list that is an instance of all the ambiguous variable's classes.

```
Only one default declaration is permitted per module, and its
effect is limited to that module. If no default declaration is
given in a module then it assumed to be:

default (Integer, Double)
```

In other words:

```
(\dNuma -> (fromInteger dNuma 2)) dictNumInteger
= fromInteger dictNumInteger 2
= fromInteger_Integer 2
```

2.
```
5.7 :: (Fractional  a) => a

and rewrites to
\dFractionala -> (fromRational dFractionala 57 :% 10

After overloading resolution rewrites to
fromRational_Double 57 :% 10
```

3.
```
len1 l = if (l == []) then 0 else 1 + len1(tail l)
len1 :: (Eq a, Num b) => [a] -> b

translation of len1
len1 dEqa dNumb l = if (== (dictEqList dEqa) l [])
                          then fromInteger dNumb 0
                    else (+ dNumb) 1  (len1 dEqa dNumb (tail l))

What does len1 [1,2,3] rewrite to?

overloaded type of list - \dNumc [fromInteger dNumc 1, fromInteger dNumc 2,
                                  fromInteger dNumc 3] :: (Num c) => [c]

with overloading removed - list dNumc :: [c]
with overloading removed - len1 dEqa dNumb :: [a] -> b

unification would give c / a and therefore dNuma = dNumc. Make
this change in list - replace dNumc by dNumb

(len1 dEqa dNumb) list :: b
```

```
overloaded type of the same expression is:
   \dEqa \dNuma \dNumb (len1 dEqa dNumb) list :: (Eq a, Num a, Num b) => b
=> \dNuma \dNumb (len1 (getEqfromNum dNuma) dNumb) list :: (Num a, Num b) => b

default declaration gives

   (\dNuma \dNumb (len1 (getEqfromNum dNuma) dNumb) list) dictNumInteger
                 dictNumInteger :: Integer
=> len1 dictEqInteger dictNumInteger [fromInteger_Integer 1,
                                      fromInteger_Integer 2,
                                      fromInteger_Integer 3]

What happens in the case of len1 []

[] :: [c]
len1 dEqa dNumb :: [a] -> b

(len1 dEqa dNumb) []  :: b
with overloading
\dEqa \dNumb (len1 dEqa dNumb) []  :: (Eq a, Num b) => b

default declaration gives

\dEqa (len1 dEqa dictNumInteger) []  :: (Eq a) => Integer

Top level unresolved overloading
```

4.
```
len2 [] = 0
len2 (x:xs) = 1 + len1 xs

len2 :: Num b => [a] -> b
len2 [] :: Num b => b
because of default declarations, this resolves to
len2 [] :: Integer
```

5.
```
funny l = if (l == [1]) then head l else 1 + funny (tail l)

l::[a] |- (l == [1]):: (Num a) => Bool
l::[a] |- (head l)  :: (Num a) => a
l::[a] |- 1 + funny (tail l) :: (Num a) => a
funny :: (Num a) => [a] -> a
```

6.
```
funny' l = if (l == [1]) then 1  else 1 + funny' (tail l)
```

```
funny' :: (Num a, Num b) => [a] -> b
```

7. `f x = 1 + funny []`
   `f :: (Num a, Num b) => c -> b`

8. `read :: (Read a) => String -> a`
   `read  "2.0" :: (Read a) => a -- top level unresolved overloading`
   `read  "2.0" + 3.0 :: (Fractional a, Read a) => a`
   `default rewrites to Double`

9. `show  :: (Show a) => a -> String`
   `show (2+2) :: (Show a, Num a) => String`
   `default rewrites to String`

10. `show ( read "123") :: (Show a, Read a) => String`
    `top level unresolved overloading`