# Notes on Type Classes

Author: **Amitabha Sanyal**

April 6, 2005

## 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 `member` to check whether two functions are equal. A more accurate description of the type of `member` is – `member :: a -> [a] -> Bool` for only those types `a` whose members can be compared for equality. The language of type expressions is extended to express such a type.

```
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 $==_{\texttt{Int}}$, then `dictEqInt = <` $==_{\texttt{Int}}$ `,` $/=_{\texttt{Int}}$ `>`. The dictionary is created from the instance declaration. Similarly, `dictOrdInt = < dictEqInt,` $>_{\texttt{Int}}$ `,` $>=_{\texttt{Int}}$ `>`. 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 `\dEq ->` $\texttt{project}_1^1$ `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`

```
f 1 2 dictEqInt
=> (==) dictEqInt 1 2
=> ==Int 1 2
=> False
```

Similarly

```
f '3' '4' dictEqChar
=> (==) dictEqChar '3' '4'
=> project_1^1 < ==_Char , /=_Char > '3' '4'
=> ==_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, >`$_{\texttt{X}}$`, >=`$_{\texttt{X}}$` >` for a type `X`. Further we have the selectors

```
(>) = \dOrd -> project_2^3 dOrd
(>=) = \dOrd -> project_3^3 dOrd
getEqfromOrd = \dOrd -> project_1^3 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
```

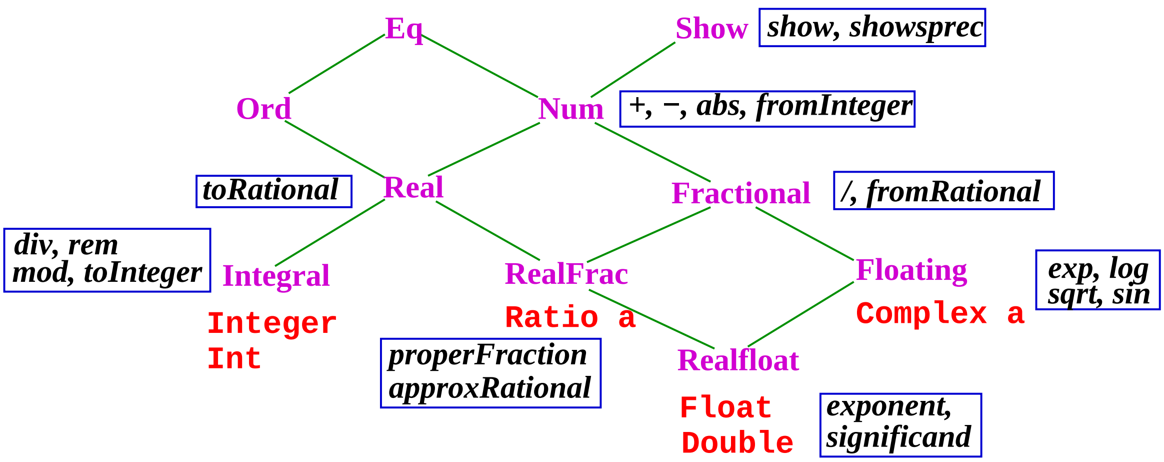
Therefore

```
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

$integerLiteral \to digit\ \{digit\}$

$floatLiteral \to integerLiteral\ .\ integerLiteral[\ e\ [-]\ integerLiteral]$

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. Represents the non-integral types. Should support general division (`/`)

2. `a` should be obtainable from an `Rational`. (`fromRational`)

`a` ∈ `Floating`

1. Contains all floating point types, both real and complex. Should support floating point operations `exp`, `log`, `sqrt`, `sin`, `cos`, `sinh`, `cosh`.

`a` ∈ `Real`

1. Contains all numeric types a which have an order. Also, should support a function `toRational` to convert `a` to a `Rational` without loss of precision.

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

a ∈ `Integral`

1. Should support basic integer operations `div, rem, mod`.
2. `a` should be approximable to a `Integer` (without loss of precision).

a ∈ `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 ∈ `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 `Integer`s. |
| `val :: Float` | |
| `val = 4 + 7` | Here both `4` and `7` are `Float`s. |

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` $(t_1$ , ... , $t_n)$

   where $n \geq 0$ , and each $t_i$ must be a monotype for which `Num` $t_i$ 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
```
   $=$ $\texttt{fromInteger}_{\texttt{Integer}}$ `2`

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

   This rewrites to:

   `\dFractional a -> (fromRational dFractionala 57 :% 10)`

   After overloading resolution this rewrites to:

   $\texttt{fromRational}_{\texttt{Double}}$ `57 :% 10`

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

   The translation of `len1` is:

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

   Now consider the application of `len1 [1,2,3]`. `[1,2,3]` has the type `(Num c) => [c]` and rewrites to:

   `\dNumc [fromInteger dNumc 1, fromInteger dNumc 2, fromInteger dNumc 3]`

   Igonring the context:

   `[1,2,3] :: [c]`, and
   `len1 :: (Eq a, Num b) => [a] -> b`

   unification would give `c / a`. Therefore the type of `len1 [1,2,3]` is `(Eq a, Num a, Num b) => b`, and the translation is:

   `\dEqa \dNuma \dNumb -> (len1 dEqa dNumb) [fromInteger dNuma 1, fromInteger dNuma 2, fromInteger dNuma 3]`

   But we can derive a `Eq` dictionary from a `Num` dictionary. Therefore,

   `len1 [1,2,3] :: (Num a, Num b) => b`, and its translation is:

   `\dNuma \dNumb -> (len1 (getEqfromNum dNuma) dNumb) [fromInteger dNuma 1, fromInteger dNuma 2, fromInteger dNuma 3]`

   The default declaration gives `a` and `b` as `Integer`. Therefore, we have:

   `len1 dictEqInteger dictNumInteger [` $1_{\texttt{Integer}}$, $2_{\texttt{Integer}}$, $3_{\texttt{Integer}}$`,]`

   What happens in the case of `len1 []`? Since `[] :: [c]`,

   `len1 [] :: (Eq a, Num b) => b`, and translates to

   `\dEqa \dNumb (len1 dEqa dNumb) []`

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

6.
```
funnier l = if (l == [1]) then 1  else 1 + funnier (tail l)
funnier :: (Num a, Num b) => [a] -> b
```

7.
```
funniest x = 1 + funny []
funniest :: 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