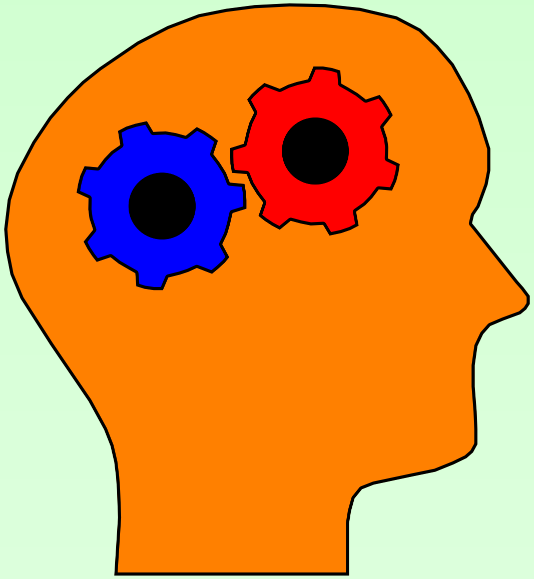
CS2104: Programming Languages Concepts

Lecture 5 : **Other Haskell Features**



"Comprehension Syntax and More Type-Classes"

Lecturer : <u>Chin</u> Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Haskell – Other Highlights

- List Comprehension
- Numbers
- Arrays
- Monoid Type Class

Comprehension

Syntactic sugar makes codes more readable

List Comprehension

- List comprehension is a useful *shorthand* for building list data structures:

```
[f x | x <- xs]
```

- Captures a list of all `f x` where `x` is drawn from `xs`. More than one generators are allowed:

```
[(x,y) | x <- xs, y <- ys]
```

- Guards are also permitted. Example :

```
quicksort []     = []
quicksort (x:xs) = quicksort [y | y <- xs, y < x]
                   ++ [x]
                   ++ quicksort [y | y <-xs, y >= x]
```

Translation for List Comprehension

- Given:

```
[f x | x <- xs]
```

- This get translated to:

```
map (\ x -> f x) xs
```

- Recall:

```
map f [] = []
map f (x:xs) = (f x):map f xs
```

Translation for List Comprehension

- Given:

```
[f x | x <- xs, x>5]
```

- This get translated to:

```
map (\x -> f x)  (filter (\ x -> x>5)  xs)
```

- Recall:

```
filter f [] = []
filter f (x:xs) = if (f x) then x : (filter f xs)
                  else filter f xs
```

Translation for List Comprehension

- Given:

```
[(x,y) | x <- xs, y <- ys]
```

- This get translated to:

```
concatMap (\ x -> map (\y -> (x,y)) ys)  xs
```

where:

```
concatMap f [] = []
concatMap f (x:xs) = (f x) ++ (concatMap f xs)
```

Translation for List Comprehension

- General translation scheme:

```
[ e | x <- xs]
        ➔ map (\x -> e) xs

[ e | x <- xs, y <- ys, rest ]
        ➔ concatMap (\x -> [ e | y <- ys, rest] ) xs

[ e | x <- xs, test, rest ]
        ➔ [ e | x <- filter (\ x -> test) xs, rest ]
```

Exercise

[e | x <- xs]   ➔ map (\x -> e) xs

[e | x <- xs, y <- ys, rest] ➔ concatMap (\x -> [e | y <- ys, rest]) xs

[e | x <- xs, test, rest] ➔ [e | x <- filter (\ x -> test) xs, rest]

[(x+x,j) | x <- [1..3], x < 2, j <- [7..8]] ➔

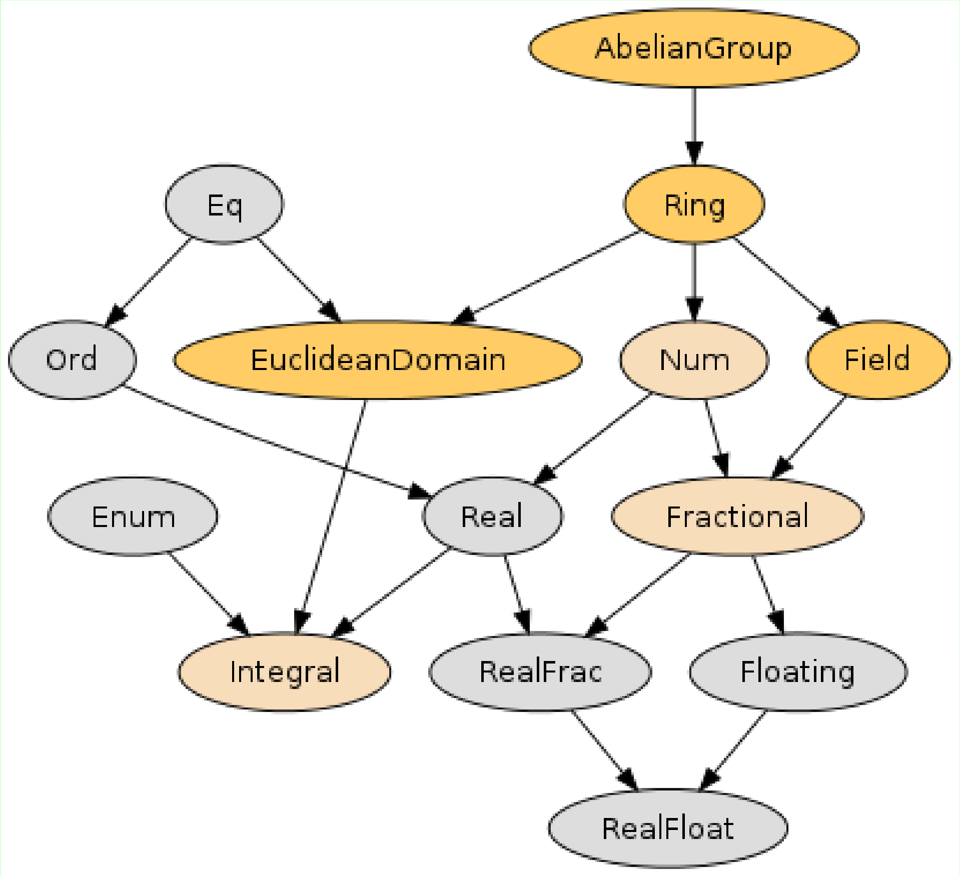
More Type Classes

Recap of Type Classes

- Supports systematic overloading via types.
- So far, mostly first-order type class
- Type sub-classes can be supported (rich hierarchy)
- Captures only type signatures but not properties, such as associativity and commutativity etc

Numbers

Hierarchy in Prelude YAP



Numbers

- `Num` class has the following basic operations: …

```
class Num a where
  (+),(-),(*)   :: a -> a -> a
  negate,abs    :: a -> a
```

- Division via `div/mod` is supported in `Integral` class, while `(/)` is supported by `Fractional` class.

- The `Floating` class contains trigonometric, logarithmic and exponential functions.

Constructed Numbers

- Standard numeric types `Int, Integer, Float, Double` are primitives and other numeric types are constructed from these

- `Complex` type is under `Floating` but made from `Realfloat`, as follows:

```
data (realFloat a) => Complex a
            = !a :+ !a deriving (Eq,text)

conjugate :: RealFloat a => Complex a -> Complex a
conjugate (x :+ y) = x :+ (-y)
```

Constructed Numbers

- Another class is `Ratio` :

```
(%)          :: (Integral a) => a => a -> Ratio a
numerator, denominator
             :: Integral a => Ratio  a -> a
```

- This is quite different from `Complex`

Arrays

Arrays

- Arrays can be regarded as functions from indices to values, but for efficient retrieval it has to be contiguous and bounded.

```
class (Ord a) => Ix a where
  range     :: (a,a) -> [a]
  index     :: (a,a) -> a -> Int
  inRange   :: (a,a) -> a -> Bool
```

- Possible index types : `Int`, `Integer`, `Char`, `Bool`, tuples of `Ix` type upto length 5

- Possible bounds:
  `(0,9)` for 1-dimensional array with 10 elements
  `((1,1),(100,100))` for 2-dimensional array

Arrays

- *range* enumerates a list of indices in index order.

```
range (0,4)          ⇒      [0,1,2,3,4]
range ((0,0),(1,2))  ⇒      [(0,0),(0,1),(0,2),(1,0),
                                   (1,1),(1,2)]
```

- *inRange* checks if an index is between a pair of bounds

- *index* calculates zero-origin offset of an index from its bounds

```
index (1,9) 2                   ⇒     1
index ((0,0),(1,2)) (1,1)       ⇒     4
```

Array Creation

- Can build an array from an association list with:

```
array  :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

- An array of squares from 1 to 100

```
squares = array (1,100) [(i,i*i) | i <- [1..100]]
```

- Array subscription done with `!` Operator.

```
squares ! 7          ⇒     49
```

- Bounds of an array determined by

```
bounds squares       ⇒     (1,100)
```

Recursive Array

- Arrays may be defined recursively

```
fibs   :: Int -> Array Int Integer
fibs n = a where a = array (0,n) [(0,1),(1,1)]++
                    [(i, a!(i-2)+a!(i-1) | i <- [2..n]]
```

- Lazy evaluation avoids need to worry on the order of evaluating such recursive arrays. A wavefront example:

```
wavefront :: Int -> Array (Int,Int) Int
wavefront n = a where
           a = array ((1,1),(n,n))
                 ([((1,j),1) | j <- [1..n]] ++
                  [((i,1),1) | i <- [2..n]] ++
                  [((i,j),a!(i,j-1) + a!(i-1,j-1)+ a!(i-1,j))
                       | i <- [2..n], j <- [2..n]])
```

Accumulation

- While index is unique for array creation, we may wish to accumulate a number of values into each index.

accumulating function   initial value   bounds

```
accumArray :: (Ix a) => (b->c->b) -> b -> (a,a)
           -> [Assoc a b] -> Array a b
```

elements to accumulate

- histogram can calculate occurrences

```
hist :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i,1) | i <- is,
                                     inRange bnds i]

e17 :: Array Char Int
e17 = hist ('a','z') "This counts the frequencies of each lowercase
letter"
```

Incremental Updates

- An incremental update operation allows a modified array to be returned, e.g. `a // [(i,v),(j,w)]`

```
(//) :: (Ix a) => Array a b -> [(a,b)]  -> Array a b
```

- Its index may appear multiple times with the last value taking precedence.

- An example to swap two rows.

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c
               -> Array (a,b) c
swapRows i i' a = a // [assoc | j <- [jLo..jHi],
                              assoc <- [((i,j),a!(i',j)),
                                 ((i',j),a!(i,j))]]
               where ((iLo,jLo),(iHi,jHi)) = bounds a
```

Algebraic Structures

Semi-Group and Monoids

- We can capture *mathematical* structures as type classes.

```
class SemiGroup a where
  op          :: a -> a -> a
class SemiGroup a => Monoid a where
  unit        :: a
```

- Above declaration only captures the type signatures. Two properties of monoids are:

```
unit `op` x = x `op` unit = x
(x `op` y) `op` z = x `op` (y `op` z)
```

- But these properties are important but not checked by Haskell. We assume that users ensure them.

Semi-Group and Monoids

```
class SemiGroup a where
  op          :: a -> a -> a
class SemiGroup a => Monoid a where
  unit        :: a
```

- Above declaration only captures the type signatures. Two properties of monoids are:

```
unit `op` x = x `op` unit = x
(x `op` y) `op` z = x `op` (y `op` z)
```

- But these properties are important but not checked by Haskell. We assume that users ensure them.

Higher-Order Type Classes

Monoid --> Monads