Number Typeclasses
Pattern Matching

CSC345: Programming Languages and Paradigms

WCU WEST CHESTER UNIVERSITY

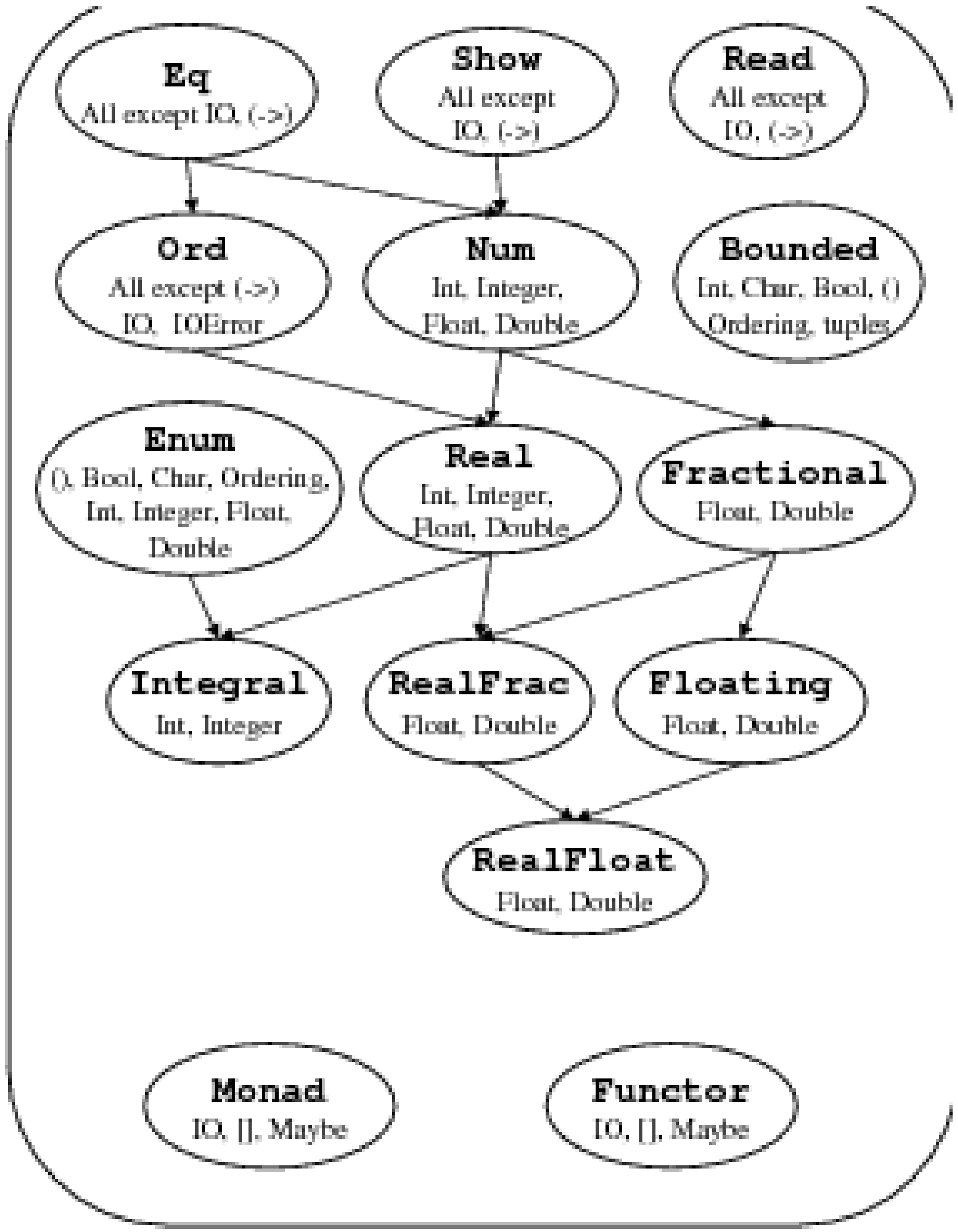
Number Typeclasses

- Initially confusing?
- All literal numbers in Haskell are constructors that create some instance of the **Num** typeclass

```
Prelude> 1::Float
1.0
```

```
Prelude> 1::Integer
1
```

```
Prelude> :info Float
type Float :: *
data Float = GHC.Types.F# GHC.Prim.Float#
  -- Defined in ‘GHC.Types’
instance Eq Float -- Defined in ‘GHC.Classes’
instance Ord Float -- Defined in ‘GHC.Classes’
instance Enum Float -- Defined in ‘GHC.Float’
instance Floating Float -- Defined in ‘GHC.Float’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Num Float -- Defined in ‘GHC.Float’
instance Real Float -- Defined in ‘GHC.Float’
instance RealFloat Float -- Defined in ‘GHC.Float’
instance RealFrac Float -- Defined in ‘GHC.Float’
instance Show Float -- Defined in ‘GHC.Float’
instance Read Float -- Defined in ‘GHC.Read’
```


Eq
All except IO, (->)
Show
All except IO, (->)
Read
All except IO, (->)
Ord
All except (->) IO, IOError
Num
Int, Integer, Float, Double
Bounded
Int, Char, Bool, () Ordering, tuples
Enum
(), Bool, Char, Ordering, Int, Integer, Float, Double
Real
Int, Integer, Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
RealFloat
Float, Double
Monad
IO, [], Maybe
Functor
IO, [], Maybe

Experiments

```
Prelude> :info (/)
Prelude> :info (+)

Prelude> 1/2
Prelude> :type 1/2
Prelude> (1/2)::Float
Prelude> (1::Float) / 2
Prelude> 3+4
Prelude> :type (3::Int)+4
```

```
Prelude> :type 3::Float
Prelude> sqrt 2::Float
Prelude> sqrt 2::Double
Prelude> :info sqrt
```

Conversion Example

```
percent :: Int -> Int -> Float
```

Pattern Matching

Another way of writing functions instead of using guards

- *Idea:* can also use literals on the LHS of equations, not just limited to "variables"

Example: (only literal)

```
not' :: Bool -> Bool
not' True  = False
not' False = True
```

instead of

```
not' :: Bool -> Bool
not' n
  | n == True  = False
  | n == False = True
```

More readable?

Pattern Matching: Another Example

```
f :: Integer -> Integer
f 0 = 10
f 1 = 11
```

Pattern Matching: (Literal + Variable)

```
xor' :: Bool -> Bool -> Bool
```

Pattern Matching: Recursion

```
-- compute the sum of the integers from 1 to n
summation :: Integer -> Integer
summation 0 = 0
summation n = n + summation(n-1)
```

Pattern Matching: (&&&)