函数式语言程序设计

类型类（Typeclass）

类型类 (typeclass)

- 类型：值的集合
  - Bool: True 和 False 的集合
  - Int: 整数的集合
- 类型类：类型的集合
  - Eq: 可以判断是否相等的类型的集合
  - Ord: 可以比较大小的类型的集合
  - Num: 可以进行加减乘除等运算的类型

定义类型类

- 上周作业中的 Heap 类型类为例

```
data Node a = Node { root :: a
                   , rank :: Int
                   , children :: [Node a]
                   } deriving (Show)

class Heap h where
  link :: (Ord a) => h a -> Node a -> Node a -> Node a
  empty :: h a -> Bool
  insert :: (Ord a) => a -> h a -> h a
  meld :: (Ord a) => h a -> h a -> h a
  findMin :: (Ord a) => h a -> a
  deleteMin :: (Ord a) => h a -> h a
```

类型类语法

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

    {-# INLINE (/=) #-}
    {-# INLINE (==) #-}
    x /= y               = not (x == y)
    x == y               = not (x /= y)
    {-# MINIMAL (==) | (/=) #-}


elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) | x == y = True
              | otherwise = elem x ys
```

如何定义类型类的实例

```
type ID = Int
type Name = String
type Score = Int
data Student = Stu ID Name Score

zhao = Stu 1 "Zhao" 99
qian = Stu 2 "Qian" 82
classX = [zhao, qian]

instance Eq Student where
  Stu x _ _ == Stu y _ _ = x == y
```

使用实例

```
> zhao == qian
False
> [zhao, qian] == [zhao, qian]
True
> :i Eq
...
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
...
```

定义[a]的实例

```
instance (Eq a) => Eq [a] where
    {-# SPECIALISE instance Eq [[Char]] #-}
    {-# SPECIALISE instance Eq [Char] #-}
    {-# SPECIALISE instance Eq [Int] #-}
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _      == _      = False
```

实例方法(method)的互相实现

```
class  Eq a  where
    ....
    x /= y               = not (x == y)
    x == y               = not (x /= y)
    {-# MINIMAL (==) | (/=) #-}

instance Eq Student where
    Stu x _ _ == Stu y _ _ = x == y
    -- s1 /= s2 = not (s1 == s2)

> zhao /= qian
True
```

Ord类型类

```
data Ordering = LT | EQ | GT

class (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT
    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }
    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
    {-# MINIMAL compare | (<=) #-}
```

Enum类型类

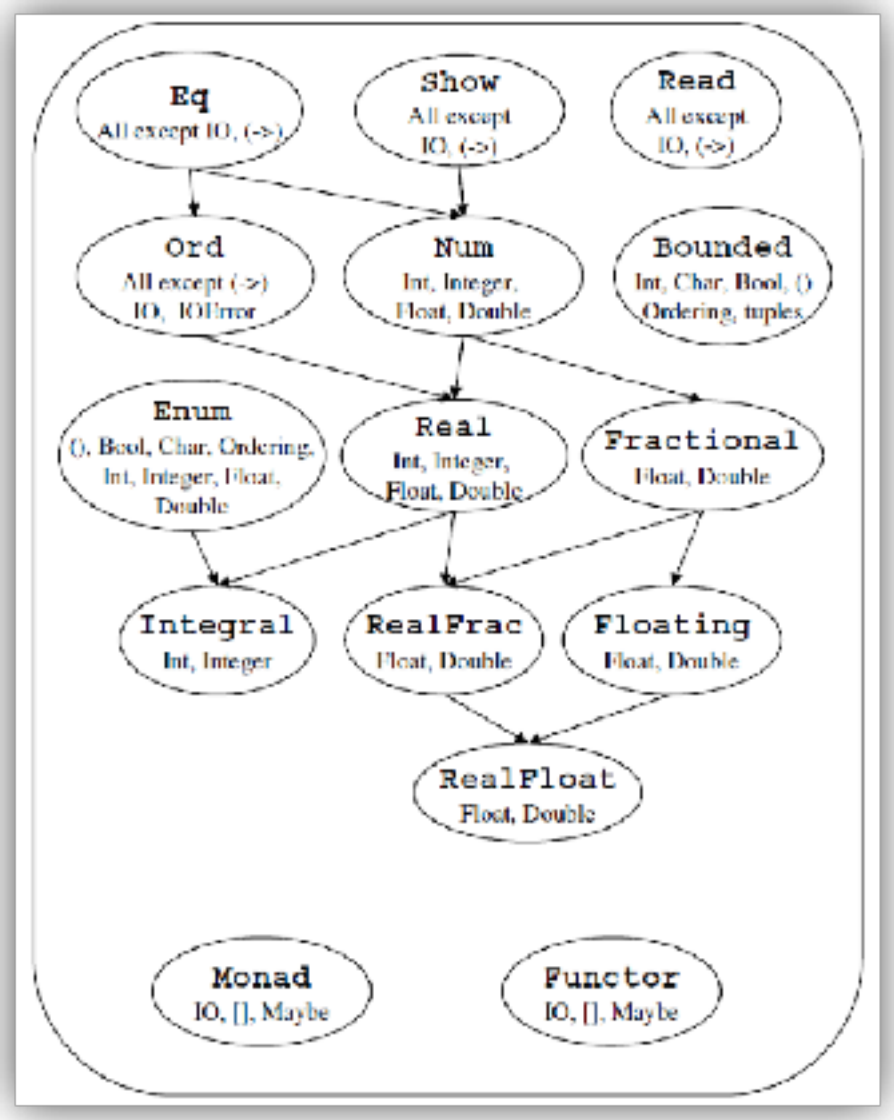
```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  {-# MINIMAL toEnum, fromEnum #-}

> fromEnum 'A'
65
> toEnum 95 :: Char
'_'
```

Bounded类型类

```
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
> maxBound :: Int
9223372036854775807
> maxBound :: Bool
True
> maxBound :: Char
'\1114111'
> maxBound :: String
error:
    • No instance for (Bounded [Char]) arising from a use of
‘maxBound’
    • In the expression: maxBound :: String
      In an equation for ‘it’: it = maxBound :: String
```

Haskell98标准库中的类型类



读写类型类 Show, Read

```
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}

type ShowS = String -> String

> show 3.14
"3.14"
> show True
"True"
```

读写类型类 Show, Read

```
> :t read
read :: Read a => String -> a
> read "1"
*** Exception: Prelude.read: no parse
> read "1" :: Int
1
> read "[1,2,3]" :: [Int]
[1,2,3]
```

deriving关键字

```
data Expr = Lit Double -- literal
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
  deriving (Show, Read, Eq, Ord) -- works
  deriving (Bounded, Enum) -- doesn't work
```

The Haskell 98 Report 规定 Haskell 编译器至少可以推导的typeclass
Eq, Ord, Enum, Bounded, Show, or Read

词典 Dictionary

```
data EqDict a = EqDict a
  { eq :: a -> a -> Bool
  , ne :: a -> a -> Bool
  }
-- eq :: EqDict a -> a - > a -> Bool

stuEqDict :: EqDict Student
stuEqDict = EqDict
  { eq (Stu id1 _ _) (Stu id2 _ _) = eq intEqDict id1 id2
  , ne s1 s2 = not (eq stuEqDict s1 s2)
  }

elem :: EqDict a -> a -> [a] -> Bool
elem (EqDict eq _)  _ [] = False
elem (EqDict eq _) x (y:ys) | x `eq` y = True
                            | otherwise = elem x ys

elem stuEqDict zhao [zhao, qian]
```

词典 Dictionary

```
data EqDict a = EqDict a
  { eq :: a -> a -> Bool
  , ne :: a -> a -> Bool
  }

stuEqDict1 :: EqDict Student
stuEqDict1 = ...

stuEqDict2 :: EqDict Student
stuEqDict2 = ...

elem stuEqDict1 ...
elem stuEqDict2 ...
```

实例的一致性（coherence）

```
class Eq where
    ...

instance Eq Student where ...
...
instance Eq Student where ...
-- duplicated instances
```

其他语言中的动态调用(dynamic dispatch)

- C++, virtual table

```
class Foo {
public:
  virtual ~Foo() {}
  virtual void foo() {}
  void* foo_payload;
}

class Bar : public Foo {
public:
  void foo() {...}
  void* bar_payload;
}
```

```
VTable of Foo (for Bar)
+----------------------------+
| Bar :: foo                 |
+----------------------------+


a Bar object
+----------------------------+
| pointer to VTable          |
+----------------------------+
| void* foo_payload          |
+----------------------------+
| void* bar_payload          |
+----------------------------+
```

其他语言中的动态调用(dynamic dispatch)

- C++, virtual table

```
class Qux {
public:
  virtual ~Qux() {}
  virtual void qux() {}
}

class Bar : public Foo
  , public Qux {
public:
  void foo() {...}
  void* bar_payload;
  void qux() {...}
}
```

```
a Bar object
+---------------------------+
| pointer to VTable(Foo4Bar)|
+---------------------------+
| void* foo_payload         |
+---------------------------+
| pointer to VTable(Qux4Bar)|
+---------------------------+
| void* bar_payload         |
+---------------------------+
```

```
Bar *b = new Bar();
Foo *f = b;
Qux *q = b;
```