Computer Science 312

Records and Type Classes

Defining Structured Types

In C:

```
struct Student {
    char* name;
    int scores[10];
}
```

In Java:

```
class Student {
    private String name;
    private int [] scores;
}
```

Defining Structured Types

In C:

```
struct Student {
    char* name;
    int scores[10];
}
```

In Java:

```
class Student {
    private String name;
    private int [] scores;
}
```

In Haskell:

```
data Student = Student String [Int]
```

What Is Missing

- In most languages, structured types include the names of the components (fields) as well as their types
- Some languages also automatically generate accessors for the components
- It would be nice to selectively transform one or more of the components while automatically retaining the values of the rest of the components

The Solution: Record Types

In Java:

```
class Student {
    private String name;
    private int [] scores;
}
```

In Haskell (algebraic type):

```
data Student = Student String [Int]
```

In Haskell (record type):

```
data Student = Student {
    studentName :: String
    studentScores :: [Int]
}
```

Using Record Types: Initialization

Algebraic type:

```
data Student = Student String [Int]
```

```
newStudent :: String -> Int -> Student
newStudent name numberOfScores =
    Student name map (\x -> 0) [1..numberOfScores]
```

Record type:

```
data Student = Student {
    studentName :: String
    studentScores :: [Int]
}
```

```
newStudent :: String -> Int -> Student
newStudent name numberOfScores = Student {
    studentName = name
    studentScores = map (\x -> 0) [1..numberOfScores]
}
```

Using Record Types: Accessors

Algebraic type:

```
data Student = Student String [Int]
```

```
getName :: Student -> String
getName (Student name _) = name
```

Record type:

```
data Student = Student {
    studentName :: String
    studentScores :: [Int]
}
```

```
getName :: Student -> String
getName student = studentName student
```

You really don’t need to define your own accessors

Using Record Types: “Mutators”

Algebraic type:

```
data Student = Student String [Int]
```

```
setName :: String -> Student -> String
setName newName (Student _ scores) = Student newName scores
```

Record type:

```
data Student = Student {
    studentName :: String
    studentScores :: [Int]
}
```

```
setName :: String -> Student -> Student
setName newName student = student {studentName = newName}
```

Reuses structure for simplicity and efficiency!

Records vs Algebraic Product Types

- Records are closer to conventional notation
- Records give you named accessors
- Records make initializers and transformations easier
- Can still use pattern matching with records if desired, because they are semantically equivalent to product types

What’s in a Type?

- For generic (polymorphic) types, such as lists and tuples, the type of the structure is independent of its item types
- For quite specific types, like strings, the item type is essential to the type of the structure
- Is there a way of defining a set of types that aren’t totally specific or totally general, but might belong together by family resemblance?

Examples: Sorting

- To guarantee that a list can be sorted, its items must recognize the comparison operators **`<`**, **`>`**, **`<=`**, and **`>=`**
- To guarantee this, they must belong to the **`Ord`** type class
- This type class is like the **`Comparable`** interface in Java, which requires that implementing classes include the **`compareTo`** method, so that methods like **`Collections.sort`** can do their work correctly
- Many data types are already ***instances*** of one or more ***type classes***

Type Classes and Their Instances

Some Standard Type Classes and Their Instances

Type Class	Instances
`Enum`	`Bool`, `Char`, `Double`, `Float`, `Int`, `Rational`
`Eq`	`Bool`, `Char`, `Double`, `Float`, `Int`, `Rational`, `String`
`Num`	`Double`, `Float`, `Int`, `Rational`
`Ord`	`Bool`, `Char`, `Double`, `Float`, `Int`, `Rational`, `String`
`Read`	Most types, except for functions
`Show`	Most types, except for functions

3 Ways to Bring Type Classes to Bear

- Derive them in algebraic data type definitions
- Include them in type signatures of functions
- Create new type instances to override the default behavior of functions in the type class

Type Class Derivation

```
module BST where

data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
  deriving (Eq, Show)
```

We derive **`Eq`** for equality tests and **`Show`** for printing in the GHCI

Include in Function Signatures

```
module BST where

data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
  deriving (Eq, Show)

bstLookup :: Ord k => k -> (BST k v) -> v
bstLookup key EmptyNode = error "Key not found"
bstLookup key (InteriorNode k v left right)
  | key < k = bstLookup key left
  | key > k = bstLookup key right
  | key == k = v
```

We add **`Ord`** for comparisons specifically of the keys in functions for searching, insertions, etc.

To Derive or Not to Derive

```
module Student where

data Student = Student String [Int]
  deriving (Show)

newStudent :: String -> Int -> Student
newStudent name numberOfScores =
  let scores = map (\x -> 0) [1..numberOfScores]
  in Student name scores
```

```
*Student> newStudent "Ken" 5
Student "Ken" [0,0,0,0,0]
```

`deriving` uses the “shows” of the nested components

But we might like a string that’s better formatted

Create a Type Instance of **Show**

```
module Student where

import Data.String (unwords)

data Student = Student String [Int]

instance Show Student where
  show (Student name scores) =
    "Name:   " ++ name ++ "\n" ++
    "Scores: " ++ unwords (map (\score -> show score) scores)
```

```
*Student> newStudent "Ken" 5
Name:   Ken
Scores: 0 0 0 0 0
```

Just like implementing a Java interface, except that you need to define only the operations that your app needs

Add Instances of **Eq** and **Ord**

```
module Student where

data Student = Student String [Int]

instance Eq Student where
  (==) (Student name1 _) (Student name2 _) = name1 == name2

instance Ord Student where
  (<=) (Student name1 _) (Student name2 _) = name1 <= name2
```

```
*Student> s1 = newStudent "Ken" 5
*Student> s2 = newStudent "Martin" 5
*Student> s1 == s2
False
*Student> s1 < s2
True
```

Getting Code for Free

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

If you include a definition of **==** in your type instance, then you automatically get a definition of **/=**, or conversely

```
module Student where

data Student = Student String [Int]

instance Eq Student where
  (==) (Student name1 _) (Student name2 _) = name1 == name2
```

Getting Code for Free

```
class Eq a => Ord where
    blah, blah, blah
```

If you include a definition of **<=** in your type instance, then you automatically get definitions of all of the other **Ord** operations (**<**, **>**, **>=**, **max**, and **min**)

Note that any **Ord** type must also be an **Eq** type

```
module Student where

data Student = Student String [Int]

instance Ord Student where
  (<=) (Student name1 _) (Student name2 _) = name1 <= name2
```

Overloading vs Polymorphism

- Functions in type classes are ***overloaded***, which means that they can be implemented by different algorithms if different type instance (integer addition is different from floating-point addition, but they both use **+** to add their operands)

- Functions on generic types, like lists, are ***polymorphic***, which means that they use the same algorithm on different types of data (the **`length`**, **`head`**, and **`tail`** functions use the same implementations for lists of whatever item types)

Some Annoyances

```
Prelude> let scores = [100, 88, 77]

Prelude> sum scores / length scores

<interactive>:3:12:
No instance for (Fractional Int) arising from a use of ‘/’
In the expression: sum scores / length scores
In an equation for ‘it’: it = sum scores / length scores
```

Shouldn’t **`(/)`** take two **`Num`**s and return a **`Float`**?

Some Annoyances

```
Prelude> let scores = [100, 88, 77]

Prelude> sum scores / length scores

<interactive>:3:12:
No instance for (Fractional Int) arising from a use of ‘/’
In the expression: sum scores / length scores
In an equation for ‘it’: it = sum scores / length scores

Prelude> :type sum :: (Num a, Foldable t) => t a -> a

Prelude> :type length :: Foldable t => t a -> Int
```

Shouldn’t **`(/)`** take two **`Num`**s and return a **`Float`**?

Some Annoyances

```
Prelude> let scores = [100, 88, 77]

Prelude> sum scores / length scores

<interactive>:3:12:
No instance for (Fractional Int) arising from a use of ‘/’
In the expression: sum scores / length scores
In an equation for ‘it’: it = sum scores / length scores

Prelude> :type sum :: (Num a, Foldable t) => t a -> a

Prelude> :type length :: Foldable t => t a -> Int

Prelude> :type (/) :: Fractional a => a -> a -> a
```

Shouldn’t **`(/)`** take two **`Num`**s and return a **`Float`**?

Some Annoyances

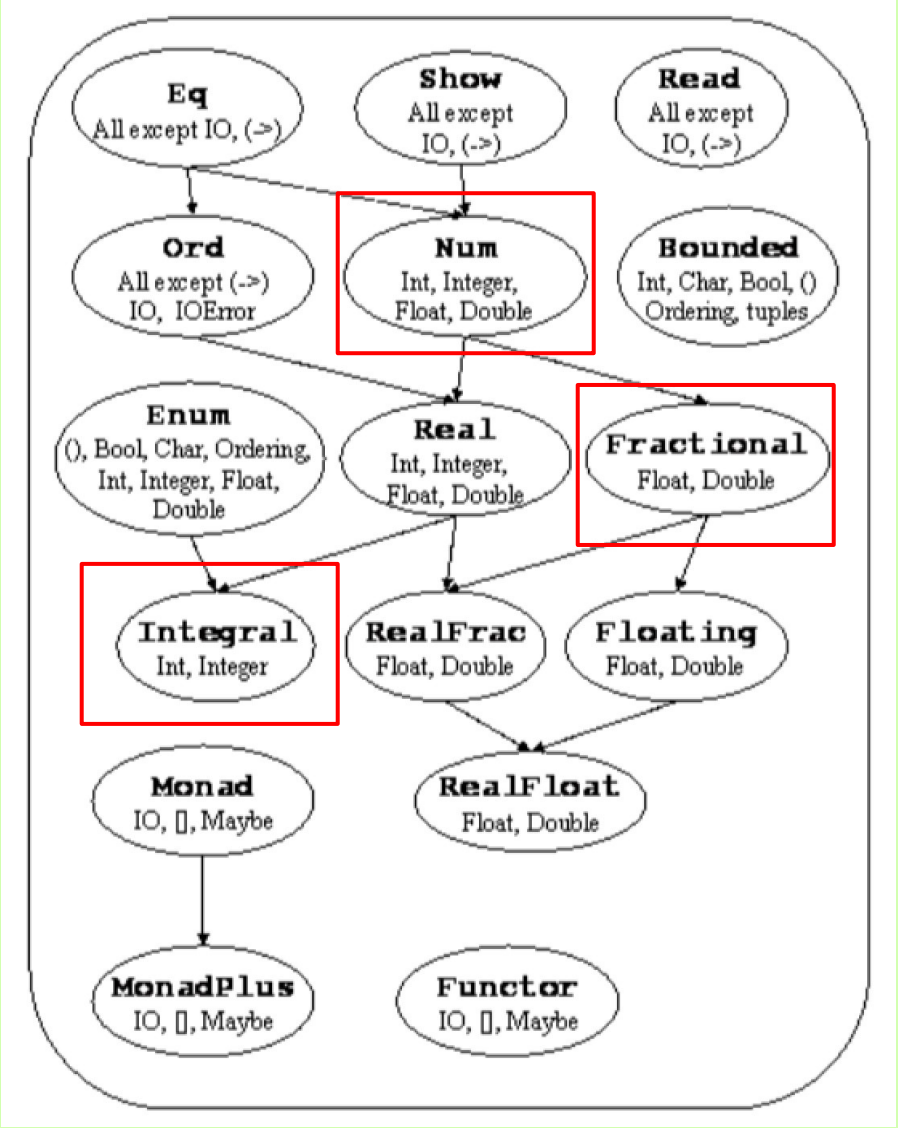
```
Prelude> 3 / 2
1.5

Prelude> :type 2
2 :: Num a => a

Prelude> let x = 2 :: Int

Prelude> :type x
x :: Int

Prelude> 3 / x
<interactive>:14:3:
No instance for (Fractional Int) arising from a use of ‘/’
In the expression: 3 / x
In an equation for ‘it’: it = 3 / x
```

If **`x`** is explicitly an **`Int`**, its type is not an instance of **`Fractional`**

The Numeric Type Class Hierarchy

`Fractional` is not connected to **`Integral`**, but it is connected to **`Num`**

So, if one can convert an **`Integral`** to a **`Num`**, one can make the connection to **`Fractional`**



Some Annoyances

```
Prelude> 3 / 2
1.5

Prelude> :type 2
2 :: Num a => a

Prelude> let x = 2 :: Int

Prelude> :type x
x :: Int

Prelude> :type fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b

Prelude> 3 / fromIntegral x
1.5
```

We must coerce an explicit **`Int`** into a **`Num`**, with **`fromIntegral`**

Whew, success!

```
Prelude> :type sum :: (Num a, Foldable t) => t a -> a

Prelude> :type length :: Foldable t => t a -> Int

Prelude> :type (/) :: Fractional a => a -> a -> a

Prelude> :type fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b

Prelude> let scores = [100, 88, 77]

Prelude> sum scores / length (fromIntegral scores)
88.33333333333333
```

`sum` already yields a **`Num`**, and we raise the type of **`length`** to a **`Num`** with **`fromIntegral`**

For next time

Introducing code with side effects: monads