# Introduction to Haskell II

Rolf Fagerberg

Spring 2007

# Type Synonyms and Copies

**Type Synonyms**

```
type String = [Char]
type Name = String
type Address = String
type Age = Int
type Coordinate2D = (Float,Float)
```

Not a new type, just another (more informative) name.

**Type Copying**

```
newtype MyString = MString String
```

A new type–a copy of the old. Class memberships (see later) may be independent from old type.

# Algebraic Types

Besides simple type synonymes (using `type` and `newtype`), more advanced (and very useful) user defined types - denoted algebraic types - can be created with the `data` keyword.

The general syntax is:

```
data Typename zero_or_more_type-variables
   = Constructor1 zero_or_more_type_expr |
   = Constructor2 zero_or_more_types_expr |
   :
   = ConstructorN zero_or_more_types
   deriving (list_of_certain_classes)
```

The identifiers for the type name and the constructor names must be capitalized.

# Examples

Enumerated types:

```
data Bool = False | True
data Ordering = LT | EQ | GT
data Seasons = Winter | Spring | Summer | Fall
data WeekDays
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun

workDays = [Mon, Tue, Wed, Thu, Fri]
```

Product types (alias tuples, alias records):

```
data DBRecord = DBRec Name Address Age
type Name = String
type Address = String
type Age = Int

person1 = DBRec "Joe Dole" "Main Street 10" 42
```

# Examples

Alternatives:

```
data Shape
  = Circle Float | Rectangle Float Float
```

Note: constructors are functions:

```
Circle ::  Float -> Shape
shape1 = Circle 3.0
Rectangle ::  Float -> Float -> Shape
shape2 = Rectangle 45.9 87.6
```

Additionally, they can (like the built-in constructors `[]`, `:`, etc.) be used as patterns in pattern matching:

```
area ::  Shape -> Float
area (Circle r)     = pi*r*r
area (Rectangle w h) = w*h
```

# Examples

Algebraic types can be recursive:

```
data IntList = EmptyList | Cons Int IntList

data IntExpr = Literal Int |
               Add IntExpr IntExpr |
               Sub IntExpr IntExpr

data IntTree = IntLeaf |
               IntNode Int IntTree IntTree

tree = IntNode 7 IntLeaf (IntNode 13 IntLeaf IntLeaf)
```

Constructors can be infix operators (identifier must then start with ´:´):

```
data IntList = EmptyList | Int ::: IntList
```

# Examples

Algebraic types can be parametric:

```
data List a = EmptyList | Cons a (List a)

data Tree a = Leaf |
              Node a (Tree a) (Tree a)
```

Example functions on trees:

```
depth :: Tree a -> Int
depth Leaf          = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)

inorder :: Tree a -> [a]
inorder Leaf          = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
```

# Haskell Classes

Class = a specified set of functions (to be overloaded among several types).

Classes created by giving their signature = the types of the set of functions (so class ≈ interface in Java).

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

Existing types can be made instances of a class by providing definitions of the functions:

```
instance Eq MyBool where
   (==) MyTrue  MyTrue  = True
   (==) MyFalse MyFalse = True
   (==) _       _       = False
```

# Context

Classes can be used as context, i.e. constraints on type variables in parametric types:

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

Can also be used in instance declarations:

```
instance Eq a => Eq [a] where
  (==) []       []      = True
  (==) (x:xs) (y:yx) = (x == y) && (xs == ys)
  (==) _        _            = False
```

Note: not all types are in the (built-in) class `Eq`. E.g. function types are not (it seems difficult to give an operational feasible definition of function equality).

# Overloading vs. Polymorphism

**Polymorphism**

>One definition of function works for many types.

**Overloading**

>Several definitions of the same function (i.e. same identifier), one for each type.

OO languages like Java normally have overloading but not polymorphism.

In Haskell, overloading eases coding (imagine naming a version of `==` for each type) and makes the notion of polymorphism stronger (more functions can be defined with the same code).

# Default Definitions

Class declarations can contain default definitions:

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

Now, instance declarations only need to define /= or ==.
Defining (overriding) both is OK.

# Derived Classes

Classes can be derived from other classes (again using the context notation):

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=) ::  a -> a -> Bool
  max, min                ::  a -> a -> a
  compare                 ::  a -> a -> Ordering
```

When declaring a type an instance of `Ord`, the methods of `Eq` are inherited.

Thus, type classes form a hierarchy rather like the class hierarchy in OO languages.

# Some Built-In Classes and Types

The standard prelude contains many predefined type classes.

E.g. for equality (`Eq`), ordering (`Ord`), enumeration(`Enum`), serialization (`Show`, `Read`), and a collection of classes for structuring the standard numeric types.

Literals may be overloaded, which can lead to ambiguities for Haskell. Of what type is e.g. `2+3`? It may be necessary to resolve explicitly:

```
(2+3)::Int
```

# All Built-In Classes
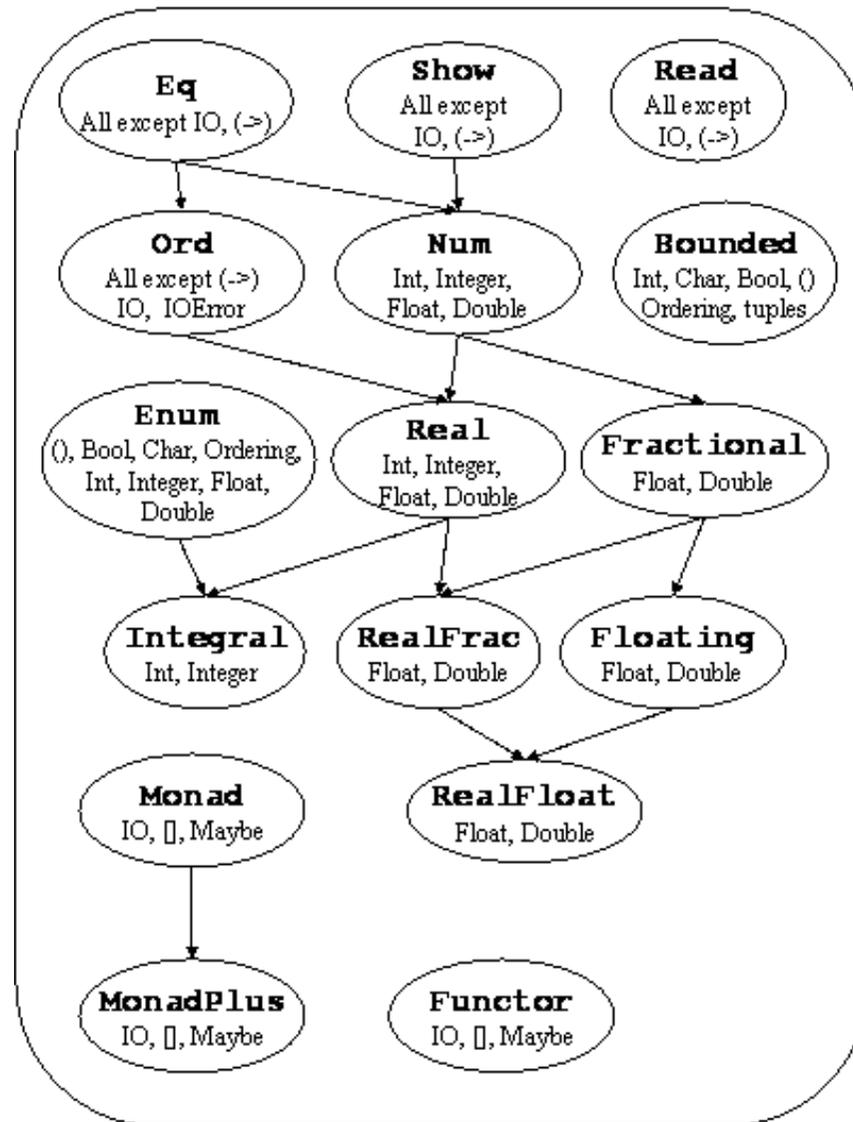


See www.haskell.org/onlinereport/

# Deriving Membership of Classes

Membership of certain standard type classes can be generated automatically in Haskell:

```haskell
data WeekDays
    = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Eq, Ord, Enum, Show, Read)
```

The operations of the classes are automatically defined using obvious (recursive) definitions (with ordering of constructors going from left to right, and using analogy with lexicographic ordering for non-nullary constructors). The derivation of `Enum` can only be done for enumeration types (nullary constructors only). More on `Enum` in later slides.

```haskell
[Mon,Wed .. Sat]  ⤳  [Mon,Wed,Fri]
```

# More Haskell Syntax

**List comprehensions**

Math: $$\{x \in S \mid x \geq 1,\ x \text{ even}\}$$

Haskell:     `[ x | x <- S, x >= 1, isEven x ]`

General form:  `[ exp | generators, guards]`

Examples:

```
[ x+y | (x,y) <- [(1,2),(7,8)], y > 5 ]  ⤳  [15]

[ (i,j) | i<-[1,2,3,4], j<-[8,9], isEven i ]
        ⤳ [(2,8),(2,9),(4,8),(4,9)]

[ j^2 | i<-[[1,2],[10,20]], j<-i ]  ⤳  [1,4,100,400]

[ [j^2|j<-i] | i<-[[1,2],[10,20]] ]  ⤳  [[1,4],[100,400]]
```

# More Haskell Syntax

**Lambda definitions**

Nameless functions defined inline:

```
zipWith (\x y -> x^2 + y^2) [1,2,3] [2,3,4]
                    ⤳ [5,13,25]


compose2 f g = \x y -> g (f x) (f y)
```

**Enumeration expression**

Easy generation of lists of certain types (types in the `Enum` class).

```
[3 .. 10]  ⤳  [3,4,5,6,7,8,9,10]
[3, 3.3 .. 4]  ⤳  [3.0,3.3,3.6,3.9]
['a', 'c' .. 'i']  ⤳  "acegi"
[False ..]  ⤳  [False,True]
```

# More Haskell Syntax

**Local definitions**

`where:` (often used)

```
f x y
   | x  < 0 = -(sqx*sqy + sqx + sqy) + g y
   | x >= 0 =   sqx*sqy + sqx + sqy
     where
       sqx = x*x
       sqy = y*y
       g z = (max x z) + t
         where t = x*y*z
```

`let:` (rarely used)

```
f x = let y=x^3; z=log x in y*z + z^2
```

# More Haskell Syntax

**Choice**

`case:` (rarely used)

```
isOdd x
  = case (x `mod` 2) of
     0 -> False
     1 -> True
```

or (not using "layout"):

```
isOdd x = case (x `mod` 2) of {0 -> False; 1 -> True}
```

`if then else:` (somewhat used, especially in textbook)

```
isOdd x = if (x `mod` 2)==0 then "Even" else "Odd"
```

# Lists

A very useful type. Many powerful and generic functions in standard Prelude for working with lists, including (see Section 8.1 in the Haskell Report):

```
map, ++, filter, concat, concatMap, head, last,
tail, init, null, length, !!, foldl, foldl1,
scanl, scanl1, foldr, foldr1, scanr, scanr1,
iterate, repeat, replicate, cycle, take, drop,
splitAt, takeWhile, dropWhile, span, break,
lines, words, unlines, unwords, reverse, and,
or, any, all, elem, notElem, lookup, sum,
product, maximum, minimum, zip, zip3, zipWith,
zipWith3, unzip, unzip3
```

Textbook covers the most important of these in Chapter 4 (not necessarily with the same implementations). Even more functions can be found in the standard library List.