User-Defined Types

Stephen A. Edwards

Columbia University

Fall 2020
Algebraic Data Types

Show and other derived type classes

Records: Naming Fields

Parameterized Types: Maybe

The type keyword

The Either Type

Lists as an Algebraic Data Type

Defining Your Own Infix Operators

Specifying and Implementing Type Classes

The Functor Type Class
Algebraic Data Types

\[
\text{data \hspace{2em} Bool = False \mid True}
\]

*Bool: Type Constructor* \hspace{3em} *False and True: Data Constructors*

\[
\text{Prelude}\text{> data MyBool = MyFalse \mid MyTrue}
\]

\[
\text{Prelude}\text{> :t MyFalse}
\]

\[
\text{MyFalse :: MyBool} \quad \text{-- A literal}
\]

\[
\text{Prelude}\text{> :t MyTrue}
\]

\[
\text{MyTrue :: MyBool}
\]

\[
\text{Prelude}\text{> :t MyBool}
\]

\[
<\text{interactive}>:1:1: \text{error: Data constructor not in scope: MyBool}
\]

\[
\text{Prelude}\text{> :k MyBool}
\]

\[
\text{MyBool :: *} \quad \text{-- A concrete type (no parameters)}
\]
Algebraic Types and Pattern Matching

\[\text{data} \ \text{Bool} = \text{False} \ | \ True\]

Type constructors may appear in type signatures; data constructors in expressions and patterns

Prelude> {:
Prelude| myAnd :: Bool -> Bool -> Bool
Prelude| myAnd False _ = False
Prelude| myAnd True x = x
Prelude| :}

Prelude> [(a,b,myAnd a b) | a <- [False, True], b <- [False, True]]
[(False,False,False),(False,True,False),
 (True,False,False),(True,True,True)]
An Algebraic Type: A Sum of Products

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

Sum = one of A or B or C...

Product = each of D and E and F...

A.k.a. tagged unions, sum-product types

Mathematically,

\[ \text{Shape} = \text{Circle} \cup \text{Rectangle} \]

\[ \text{Circle} = \text{Float} \times \text{Float} \times \text{Float} \]

\[ \text{Rectangle} = \text{Float} \times \text{Float} \times \text{Float} \times \text{Float} \]
An Algebraic Type: A Sum of Products

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

area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)

*Main> :t Circle
Circle :: Float -> Float -> Float -> Shape

*Main> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape

*Main> :k Shape
Shape :: *

*Main> area $ Circle 10 20 10
314.15927
*Main> area $ Rectangle 10 10 20 30
200.0
Printing User-Defined Types: Deriving Show

*Main> Circle 10 20 30

<interactive>:9:1: error:
  * No instance for (Show Shape) arising from a use of 'print'
  * In a stmt of an interactive GHCi command: print it

Add deriving (Show) to make the compiler generate a default show:

```haskell
data Shape = Circle Float Float Float Float
            | Rectangle Float Float Float Float
deriving Show
```

*Main> Circle 10 20 30
Circle 10.0 20.0 30.0
*Main> show $ Circle 10 20 30
"Circle 10.0 20.0 30.0"
Every Possible Automatic Derivation

```
data Bool = False | True   -- Standard Prelude definition
deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

```
Prelude> True == True
True

Prelude> False < False
False

Prelude> succ False
True

Prelude> succ True

Prelude> read "True" :: Bool
True

Prelude> show False
"False"

Prelude> minBound :: Bool
False
```

---

**Exception:** Prelude.Enum.Bool.succ: bad argument
Types as Documentation

When in doubt, add another type

data Point = Point Float Float deriving Show
data Shape = Circle Point Float
  | Rectangle Point Point
  deriving Show

area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) =
  (abs $ x2 - x1) * (abs $ y2 - y1)

*Main> area $ Rectangle (Point 10 20) (Point 30 40)
400.0
*Main> area $ Circle (Point 0 0) 100
31415.928
moveTo :: Point -> Shape -> Shape
moveTo p (Circle _ r) = Circle p r
moveTo p@(Point x0 y0) (Rectangle (Point x1 y1) (Point x2 y2)) = Rectangle p $ Point (x0 + x2 - x1) (y0 + y2 - y1)

origin :: Point
origin = Point 0 0

originCircle :: Float -> Shape
originCircle = Circle origin -- function in "point-free style"

originRect :: Float -> Float -> Shape
originRect x y = Rectangle origin (Point x y)

Prelude> :l Shapes
[1 of 1] Compiling Shapes ( Shapes.hs, interpreted )
Ok, one module loaded.
*Shapes> moveTo (Point 10 20) $ originCircle 5
Circle (Point 10.0 20.0) 5.0
*Shapes> moveTo (Point 10 20) $ Rectangle (Point 5 15) (Point 25 35)
Rectangle (Point 10.0 20.0) (Point 30.0 40.0)
module Shapes

(Point(..)  -- Export the Point constructor
, Shape(..)  -- Export Circle and Rectangle constructors
, area
, moveTo
, origin
, originCircle
, originRect
) where

data Point = Point Float Float deriving Show
-- etc.
data Person = Person { firstName :: String,
    , lastName :: String,
    , age :: Int,
    , height :: Float,
    , phoneNumber :: String,
    , flavor :: String
} deriving Show

hbc = Person { lastName = "Curry",
    firstName = "Haskell",
    age = 42,
    height = 6.0,
    phoneNumber = "555-1212",
    flavor = "Curry" }

*Main> :t lastName
lastName :: Person -> String
*Main> lastName hbc
"Curry"
Updating and Pattern-Matching Records

*Main> hbc
Person {firstName = "Haskell", lastName = "Curry", age = 42,
        height = 6.0, phoneNumber = "555-1212", flavor = "Curry"}

*Main> hbc { age = 43, flavor = "Vanilla" }
Person {firstName = "Haskell", lastName = "Curry", age = 43,
        height = 6.0, phoneNumber = "555-1212", flavor = "Vanilla"}

*Main> sae = Person "Stephen" "Edwards" 49 6.0 "555-1234" "Durian"

fullName :: Person -> String
fullName (Person { firstName = f, lastName = l }) = f ++ " " ++ l

*Main> map fullName [hbc, sae]
["Haskell Curry","Stephen Edwards"]
Record Named Field Puns In Patterns

:set -XNamedFieldPuns in GHCi or put a pragma at the beginning of the file

{-# LANGUAGE NamedFieldPuns #-}

favorite :: Person -> String
favorite (Person { firstName, flavor } ) =
    firstName ++ " loves " ++ flavor

*Main> favorite hbc
"Haskell loves Curry"

Omitting a field when constructing a record is a compile-time error unless you :
:set -Wno-missing-fields, which allows uninitialized fields. Evaluating an
uninitialized field throws an exception.
Record Wildcards

:set -XRecordWildCards in GHCi or add a pragma:

{--# LANGUAGE RecordWildCards #-}
Parameterized Types: Maybe

A safe replacement for null pointers

```haskell
data Maybe a = Nothing | Just a
```

The `Maybe` type constructor is a function with a type parameter (`a`) that returns a type (`Maybe a`).

```haskell
Prelude> :k Maybe
Maybe :: * -> *

Prelude> Just "your luck"
Just "your luck"

Prelude> :t Just "your luck"
Just "your luck" :: Maybe [Char]

Prelude> :t Nothing
Nothing :: Maybe a

Prelude> :t Just (10 :: Int)
Just (10 :: Int) :: Maybe Int
```
**Maybe In Action**

Useful when a function may “fail” and you don’t want to throw an exception

```haskell
Prelude> :m + Data.List
Prelude Data.List> :t uncons
uncons :: [a] -> Maybe (a, [a])
Prelude Data.List> uncons [1,2,3]
Just (1,[2,3])
Prelude Data.List> uncons []
Nothing

Prelude Data.List> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
Prelude Data.List> lookup 5 [(1,2),(5,10)]
Just 10
Prelude Data.List> lookup 6 [(1,2),(5,10)]
Nothing
```
Data.Map: Multiple Type Parameters

Prelude Data.Map> :k Map
Map :: * -> * -> *

Prelude Data.Map> :t empty
empty :: Map k a

Prelude Data.Map> :t singleton (1::Int) "one"
singleton (1::Int) "one" :: Map Int [Char]

Note: while you can add type class constraints to type constructors, e.g.,

```
data Ord k => Map k v = ...
```

it’s bad form to do so. By convention, to reduce verbosity, only functions that actually rely on the type classes are given such constraints.
The type Keyword: Introduce an Alias

Prelude> type AssocList k v = [(k, v)]
Prelude> :k AssocList
AssocList :: * -> * -> *
Prelude> :{
Prelude| lookup :: Eq k => k -> AssocList k v -> Maybe v
Prelude| lookup _ [] = Nothing
Prelude| lookup k ((x,v):xs) | x == k = Just v
Prelude| | otherwise = lookup k xs
Prelude| :}
Prelude> :t lookup
lookup :: Eq k => k -> AssocList k v -> Maybe v
Prelude> lookup 2 [(1,"one"),(2,"two")]
Just "two"
Prelude> lookup 0 [(1,"one"),(2,"two")]
Nothing
Prelude> :t [(1,"one"),(2,"two")]
[(1,"one"),(2,"two")]::Num a => [(a, [Char])]
Either: Funky Type Constructor Fun

```haskell
data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show)
```

Prelude> :k Either
Either :: * -> * -> *

Prelude> Right 20
Right 20

Prelude> Left "Stephen"
Left "Stephen"

Prelude> :t Right "Stephen"
Right "Stephen" :: Either a [Char]  -- Only second type inferred

Prelude> :t Left True
Left True :: Either Bool b

Prelude> :k Either Bool
Either Bool :: * -> *
```
Either: Often a more verbose Maybe

By convention, Left = “failure,” Right = “success”

```
Prelude> type AssocList k v = [(k,v)]
Prelude> :{
Prelude| lookup :: String -> AssocList String a -> Either String a
Prelude| lookup k [] = Left $ "Could not find " ++ k
Prelude| lookup k ((x,v):xs) | x == k = Right v
Prelude| | otherwise = lookup k xs
Prelude| :}
Prelude> lookup "Stephen" [(["Douglas",42],["Don",0])
Left "Could not find Stephen"
Prelude> lookup "Douglas" [(["Douglas",42],["Don",0])
Right 42
```
data List a = Cons a (List a)  -- A recursive type
       | Nil
    deriving (Eq, Ord, Show, Read)

*Main> :t Nil
Nil :: List a  -- Nil is polymorphic
*Main> :t Cons
Cons :: a -> List a -> List a  -- Cons is polymorphic
*Main> :k List
List :: * -> *  -- Type constructor takes an argument
*Main> Nil
Nil
*Main> 5 `Cons` Nil
Cons 5 Nil
*Main> 4 `Cons` (5 `Cons` Nil)
Cons 4 (Cons 5 Nil)
*Main> :t 'a' `Cons` Nil
'a' `Cons` Nil :: List Char  -- Proper type inferred
Lists of Our Own with User-Defined Operators

```
infixr 5 :.
data List a = a :: List a
    | Nil
    deriving (Eq, Ord, Show, Read)
```

Haskell symbols are  ! # $ % & * + . / < = > ? @ \ ^ | - ~

A (user-defined) operator is a symbol followed by zero or more symbols or :

A (user-defined) constructor is a : followed by one or more symbols or :

```
*Main> (1 :: 2 :: 3 :: Nil) :: List Int
1 :: (2 :: (3 :: Nil))
*Main> :t (::)
(::) :: a -> List a -> List a
```
Fixity of Standard Prelude Operators

<table>
<thead>
<tr>
<th>Precedence Level</th>
<th>Operators</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>infixr 9 ., !!</td>
<td>Highest precedence</td>
</tr>
<tr>
<td>8</td>
<td>infixr 8 ^, ^^, **</td>
<td>Right-associative</td>
</tr>
<tr>
<td>7</td>
<td>infixl 7 *, /, <code>quot</code>, <code>rem</code>, <code>div</code>, <code>mod</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>infixl 6 +, -</td>
<td>Left-associative</td>
</tr>
<tr>
<td>5</td>
<td>infixr 5 :, ++</td>
<td>: is the only builtin</td>
</tr>
<tr>
<td>4</td>
<td>infix 4 ==, /=, &lt;, &lt;=, &gt;=, &gt;, <code>elem</code></td>
<td>Non-associative</td>
</tr>
<tr>
<td>3</td>
<td>infixr 3 &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>infixr 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>infixl 1 &gt;&gt;, &gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>infixr 0 $, $!, <code>seq</code></td>
<td>Lowest precedence</td>
</tr>
</tbody>
</table>

*Main> (1::Int) == 2 == 3
<interactive>:9:1: error:
  Precedence parsing error
  cannot mix '==' [infix 4] and '==' [infix 4] in the same infix expression
The List Concatenation Operator

```haskell
infixr 5 ++. -- Define operator precedence & associativity
(++) :: List a -> List a -> List a
Nil ++. ys = ys
(x :: xs) ++. ys = x :: (xs ++. ys)
```

```
*Main> (1 :: 2 :: 3 :: Nil ++. 4 :: 5 :: Nil) :: List Int
1 :: (2 :: (3 :: (4 :: (5 :: Nil))))
```

The only thing special about lists in Haskell is the [,] syntax

```
*Main> :k List
List :: * -> *
*Main> :k []
[] :: * -> *
```

Our *List* type constructor has the same kind as the built-in list constructor []
data Tree a = Node a (Tree a) (Tree a)  -- Unbalanced binary tree  
   | Nil
   deriving (Eq, Show, Read)

singleton :: a -> Tree a
singleton x = Node x Nil Nil

insert :: Ord a => a -> Tree a -> Tree a
insert x Nil = singleton x
insert x n@(Node a left right) = case compare x a of
   LT -> Node a (insert x left) right
   GT -> Node a left (insert x right)
   EQ -> n

fromList :: Ord a => [a] -> Tree a
fromList = foldr insert Nil

toList :: Tree a -> [a]
toList Nil = []
toList (Node a l r) = toList l ++ [a] ++ toList r
member :: Ord a => a -> Tree a -> Bool
member _ Nil = False
member x (Node a left right) = case compare x a of
  LT -> member x left
  GT -> member x right
  EQ -> True

*Main> t = fromList ([8,6,4,1,7,3,5] :: [Int])
*Main> t
Node 5 (Node 3 (Node 1 Nil Nil) (Node 4 Nil Nil))
     (Node 7 (Node 6 Nil Nil) (Node 8 Nil Nil))
*Main> toList t
[1,3,4,5,6,7,8]
*Main> 1 `member` t
True
*Main> 42 `member` t
False
Specifying and Implementing Type Classes

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

x /= y = not (x == y)

x == y = not (x /= y)

data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
  Red  == Red   = True
  Green == Green = True
  Yellow == Yellow = True
  _     == _     = False

*Main> Red == Red
True

*Main> Red /= Yellow
True
Implementing Show

```haskell
instance Show TrafficLight where
  show Red  = "Red Light"
  show Green = "Green Light"
  show Yellow = "Yellow Light"
```

```
*Main> show Yellow
"Yellow Light"
*Main> [Red, Yellow, Green]
[Red Light, Yellow Light, Green Light]  -- GHCi uses show

*Main> :k Maybe
Maybe :: * -> *
  -- A polymorphic type constructor

*Main> :k Eq
Eq :: * -> Constraint
  -- Like a polymorphic type constructor

*Main> :k Eq TrafficLight
Eq TrafficLight :: Constraint  -- Give it a type to make it happy
```
The MINIMAL Pragma: Controlling Compiler Warnings

```
infix 4 ==., /=.

class MyEq a where
  {# MINIMAL (==.) | (/=.) #}  
  (==.), (/=.) :: a -> a -> Bool
  x /=. y  =  not (x ==. y)
  x ==. y  =  not (x /=. y)

instance MyEq Int where

instance MyEq Integer where
  x ==. y = (x `compare` y) == EQ
```

The MINIMAL pragma tells the compiler what to check for. Operators are , (and) and | (or). Parentheses are allowed.

Prelude> :load myeq
[1 of 1] Compiling Main

myeq.hs:9:10: warning:
  [-Wmissing-methods]
  * No explicit implementation for either '==.' or '/=.'
  * In the instance declaration for 'MyEq Int'
| 9 | instance MyEq Int where
|    ^^^^^^^^^
Eq (Maybe t)

```
data Maybe t = Just t | Nothing

instance Eq t => Eq (Maybe t) where
  Just x == Just y   = x == y    -- This comparison requires Eq t
  Nothing == Nothing = True
  _    == _          = False

The Standard Prelude includes this by just deriving Eq
```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

{-# MINIMAL (==) | (/=) #-}

instance [safe] Eq TrafficLight
instance (Eq a, Eq b) => Eq (Either a b)
instance Eq a => Eq (Maybe a)
instance Eq a => Eq [a]
instance Eq Ordering
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
instance (Eq a, Eq b) => Eq (a, b)
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
instance (Eq a, Eq b, Eq c, Eq d) => Eq (a, b, c, d)
class ToBool a where
  toBool :: a -> Bool

instance ToBool Bool where
  toBool = id -- Identity function

instance ToBool Int where
  toBool 0 = False -- C-like semantics
  toBool _  = True

instance ToBool [a] where
  toBool []  = False -- JavaScript, python semantics
  toBool _  = True

instance ToBool (Maybe a) where
  toBool (Just _) = True
  toBool Nothing  = False
Now We Can toBool Bools, Ints, Lists, and Maybes

*Main> :t toBool

```
toBool :: ToBool a => a -> Bool
```

*Main> toBool True

```
True
```

*Main> toBool (1 :: Int)

```
True
```

*Main> toBool "dumb"

```
True
```

*Main> toBool []

```
False
```

*Main> toBool [False]

```
True
```

*Main> toBool $ Just False

```
True
```

*Main> toBool Nothing

```
False
```
The Functor Type Class: Should be “Mappable”†

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  ( <$> ) :: b -> f a -> f b
m <$> b = fmap (_ -> b)
```

If \( f :: a \rightarrow b \),

\[
bs = \text{fmap } f \text{ as}
\]

applies \( f \) to every \( a \) in \( \text{as} \) to give \( bs \); \( bs = \text{as} \text{ <$ x \) replaces every \( a \) in \( \text{as} \) with \( x \).} \)

Here, \( f \) is a type constructor that takes an argument, like Maybe or List.

```haskell
instance Functor (Either a)
instance Functor []
instance Functor Maybe
instance Functor IO
instance Functor ((->) r)
instance Functor ((, a)
```

— Many others; these are
— just the Prelude’s

† “Functor” is from Category Theory
Functor Instances for \( \ast \rightarrow \ast \) Kinds

```haskell
data [] a = [] | a : [a] -- The List type: not legal syntax

instance Functor [] where -- Prelude definition
  fmap = map -- The canonical example

data Maybe t = Nothing | Just t -- Prelude definition

instance Functor Maybe where
  fmap _ Nothing = Nothing -- No object a here
  fmap f (Just a) = Just (f a) -- Apply f to the object in Just a

data Tree a = Node a (Tree a) (Tree a) | Nil -- Our binary tree

instance Functor Tree where
  fmap f Nil = Nil
  fmap f (Node a lt rt) = Node (f a) (fmap f lt) (fmap f rt)
```
Functor Either a

data Either a b = Left a | Right b

instance Either does not type check because Either :: * -> * -> *

The Prelude definition of fmap only modifies Right

instance Functor (Either a) where
  fmap _ (Left x) = Left x
  fmap f (Right y) = Right (f y)

This works because Either a :: * -> * has the right kind
Kinds: The Types of Types

Prelude> :k Int
Int :: *     -- A concrete type
Prelude> :k [Int]
[Int] :: *    -- A specific type of list: also concrete
Prelude> :k []
[] :: * -> * -- The list type constructor takes a parameter
Prelude> :k Maybe
Maybe :: * -> * -- Maybe also takes a type as a parameter
Prelude> :k Maybe Int
Maybe Int :: * -- Specifying the parameter makes it concrete
Prelude> :k Either
Either :: * -> * -> * -- Either takes two type parameters
Prelude> :k Either String
Either String :: * -> * -- Partially applying Either is OK
Prelude> :k (,)
(,) :: * -> * -> * -- The pair (tuple) constructor takes two
Type class *Tofu* expects a single type argument *t*

*j* must take an argument *a* and produce a concrete type, so *j* :: * → *

*t* must take arguments *a* and *j*, so *t* :: * → (* → *) → *

Let’s invent a type constructor of kind * → (* → *) → *. It has to take two type arguments; the second needs to be a function of one argument

*data* What a b = What (b a) *deriving* Show

---

---

---
data What a b = What (b a) deriving Show

Prelude> :t What "Hello"
What "Hello" :: What Char []
Prelude> :t What (Just "Ever")
What (Just "Ever") :: What [Char] Maybe

*What holds any type that is a “parameterized container,” what Tofu wants:*

Prelude> :k What
What :: * -> (* -> *) -> *
Prelude> :k Tofu
Tofu :: (* -> (* -> *) -> *) -> Constraint
Prelude> instance Tofu What where tofu x = What x
Prelude> tofu (Just 'a') :: What Char Maybe
What (Just 'a')
Prelude> tofu "Hello" :: What Char []
What "Hello"
Prelude> data Barry t k a = Barry a (t k)
Prelude> :k Barry
Barry :: (* -> *) -> * -> * -> *  -- Bizarre kind, by design
Prelude> :t Barry (5::Int) "Hello"
Barry (5::Int) "Hello" :: Barry [] Char Int

A *Barry* is two objects: any type and one built from a type constructor

Prelude> :k Functor
Functor :: (* -> *) -> Constraint  -- Takes a one-arg constructor

instance Functor (Barry t k) where  -- Partially applying Barry
  fmap f (Barry x y) = Barry (f x) y  -- Applying f to first object

Prelude> fmap (+1) (Barry 5 "Hello")
Barry 6 "Hello"  -- It works!
Prelude> fmap show (Barry 42 "Hello")
Barry "42" "Hello"
Prelude> :t fmap show (Barry 42 "Hello")
fmap show (Barry 42 "Hello") :: Barry [] Char String
class Eq a where
    (==), (=/=) :: a -> a -> Bool

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

class Num a where
    (+), (-), (*) :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger :: Integer -> a

class (Num a, Ord a) => Real a where
    toRational :: a -> Rational

class Enum a where
    succ, pred :: a -> a
    toEnum :: Int -> a
    fromEnum :: a -> Int
    ...
Integral Typeclasses and Conversion

```haskell
class (Real a, Enum a) => Integral a where
  quot, rem, div, mod :: a -> a -> a
  quotRem, divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer

instance Integral Int
instance Integral Word
instance Integral Integer
```

Conversion among Integrals:

```haskell
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```
RealFrac Typeclasses and Conversion

class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a

class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate, round, ceiling, floor :: Integral b => a -> b

Conversions among Reals and Fractionals:

realToFrac :: (Real a, Fractional b) => a -> b
realToFrac = fromRational . toRational

instance RealFrac Float
instance RealFrac Double

type Rational = GHC.Real.Ratio Integer
Conversion Examples

Prelude> :t 42
42 :: Num p => p
Prelude> :t 42.0
42.0 :: Fractional p => p

Prelude> (fromIntegral (42 :: Int)) :: Word
42
Prelude> (realToFrac (42 :: Int)) :: Double
42.0
Prelude> (realToFrac (42.5 :: Float)) :: Double
42.5
Prelude> (floor (42.5 :: Double)) :: Int
42

https://wiki.haskell.org/Converting_numbers