User-Defined Types

Stephen A. Edwards

Columbia University

Fall 2019
Algebraic Data Types

```
data Bool = False | True
```

**Bool:** Type Constructor  **False and True:** Data Constructors

```
Prelude> data MyBool = MyFalse | MyTrue

Prelude> :t MyFalse
MyFalse :: MyBool -- A literal

Prelude> :t MyTrue
MyTrue :: MyBool

Prelude> :t MyBool
<interactive>:1:1: error: Data constructor not in scope: MyBool
Prelude> :k MyBool
MyBool :: * -- A concrete type (no parameters)
```
Algebraic Types and Pattern Matching

```haskell
data Bool = False | True
```

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

```haskell
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 \ast r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) \ast (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 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

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

```
Prelude> True == True
True   -- Eq
Prelude> False < False
False  -- Ord
Prelude> succ False
True   -- Enum
Prelude> succ True
Prelude> read "True" :: Bool
True   -- Read
Prelude> show False
"False" -- Show
Prelude> minBound :: Bool
False  -- Bounded
```
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.
Records: Naming Product Type Fields

```haskell
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.
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*).

```
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

```
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.,

```hs
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

```
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