Table of Contents

Custom Types

Benson Joeris
benson@bjoeris.com
http://bjoeris.com

pluralsight
hardcore developer training

Overview

- Type Synonyms
- Newtype
- Records
- Algebraic Data Types

Table of Contents

Type Synonyms

```
type String = [Char]
```

Type Synonyms

```
type Point = (Double,Double)
```

```
midpoint :: (Double,Double) -> (Double,Double)
            -> (Double,Double)
midpoint (x1,y1) (x2,y2) =
   ((x1 + x2) / 2, (y1 + y2) / 2)
```

```
midpoint' :: Point -> Point -> Point
midpoint' (x1,y1) (x2,y2) =
   ((x1 + x2) / 2, (y1 + y2) / 2)
```

Type Synonyms

```
type Point = (Double,Double)
```

```
midpoint :: Point -> Point -> Point
midpoint (x1,y1) (x2,y2) =
   ((x1 + x2) / 2, (y1 + y2) / 2)
```

```
p1 :: (Double,Double)
p1 = (1,2)

p2 :: Point
p2 = (3,4)

mid :: (Double,Double)
mid = midpoint p1 p2
```

Type Synonyms

- Make code more readable
- Semantic meaning
- Completely ignored by compiler

Table of Contents

Newtype

```
newtype CustomerId = MakeCustomerId Int
```

~~badCustomer :: CustomerId~~
~~badCustomer = 13~~

```
customer :: CustomerId
customer = MakeCustomerId 13
```

Newtype

```
newtype CustomerId = MakeCustomerId Int
```

```
customerToInt :: CustomerId -> Int
customerToInt (MakeCustomerId i) = i
```

Newtype

```
newtype CustomerId = CustomerId Int
```

```
customer :: CustomerId
customer = CustomerId 13
```

```
customerToInt :: CustomerId -> Int
customerToInt (CustomerId i) = i
```

Newtype

- Create a new type *represented* by an existing type
- New type and representation cannot be mixed up
- Add semantic meaning, checked by compiler

Table of Contents

Records

```
data Customer = MakeCustomer
  { customerId  :: CustomerId
  , name        :: String
  , luckyNumber :: Int
  }
```

```
alice :: Customer
alice = MakeCustomer
  { customerId  = MakeCustomerId 13
  , name        = "Alice"
  , luckyNumber = 42
  }
```

Records

```
data Customer = MakeCustomer
  { customerId  :: CustomerId
  , name        :: String
  , luckyNumber :: Int
  }
```

```
GHCi> customerId alice
Result: MakeCustomerId 13
GHCi> name alice
Result: "Alice"
GHCi> luckyNumber alice
Result: 42
```

Records

```
sally = alice { name = "Sally", luckyNumber = 84 }
```

```
GHCi> name sally
Result: "Sally"
GHCi> customerId sally
Result: MakeCustomerId 13
GHCi> name alice
Result: "Alice"
```

Records

- Not extensible

```
data Person = Person { name :: String }
data Customer = Customer extends
                      Person { luckyNumber :: Int }
```

Records

- Not extensible
- No shared field names

```
data Customer = Customer
   { name       :: String
   , customerId :: CustomerId
   }
data Supplier = Supplier
   { name       :: String
   , supplierId :: SupplierId
   }
```

Records

- Not extensible
- No shared field names

Table of Contents

Algebraic Data Types

```
data Customer = MakeCustomer CustomerId String Int
```

Algebraic Data Types

```
data Customer = Customer CustomerId String Int
```

```
alice :: Customer
alice = Customer (CustomerId 13) "Alice" 42
```

```
getCustomerId :: Customer -> CustomerId
getCustomerId (Customer cust_id name luckyNumber) =
  cust_id
```

Algebraic Data Types

```
getCustomerId :: Customer -> CustomerId
getCustomerId (Customer cust_id _ _) = cust_id
```

Algebraic Data Types

- Newtype, but with more arguments

```
data Customer = Customer CustomerId String Int
```

```
newtype CustomerId = CustomerId Int
```

- Tuples, but with names

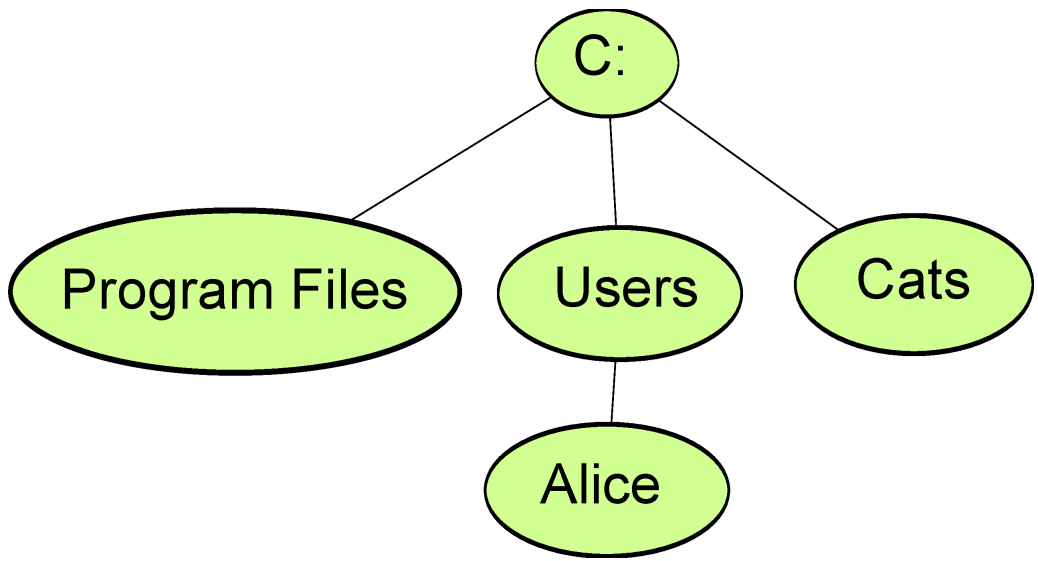
```
x :: (Double,Double,Double)
```

```
data RGB = RGB Double Double Double
x :: RGB
```

Algebraic Data Types

```
data StringTree = StringTree String [StringTree]
```

```
hierarchy = StringTree "C:"
              [ StringTree "Program Files" []
              , StringTree "Users"
                  [StringTree "Alice" []]
              , StringTree "Cats" []
              ]
```



Algebraic Data Types

- Package some values together
- Named container

Table of Contents

Algebraic Data Type Constructors

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

```
x :: Bool
x = False
y :: Bool
y = True
```

```
negate :: Bool -> Bool
negate True = False
negate False = True
```

Algebraic Data Type Constructors

```
data DialogResponse = Yes | No | Help | Quit
```

Algebraic Data Type Constructors

```
data MaybeInt = NoInt | JustInt Int
```

```
defaultInt :: Int -> MaybeInt -> Int
defaultInt defaultValue NoInt = defaultValue
defaultInt _ (JustInt x) = x
```

Algebraic Data Type Constructors

```
data StringList = EmptyStringList
                | ConsStringList String StringList
```

```
lengthStringList :: StringList -> Int
lengthStringList EmptyStringList = 0
lengthStringList (ConsStringList _ xs) =
  1 + lengthStringList xs
```

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Algebraic Data Type Constructors

- Constructors –different kinds of values for a type

Table of Contents

Parameterized Types

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

```
x :: Maybe Int
x = Nothing
```

```
fromMaybe :: a -> Maybe a -> a
fromMaybe defaultVal Nothing = defaultVal
fromMaybe _ (Just x) = x
```

Parameterized Types

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

Parameterized Types

```
data Map k a  = ···
```

Parameterized Types

- Parametrized Types –Hold values of any type

Table of Contents

Summary

- Type synonyms
- Newtype
- Records
- Algebraic Data Types
  - Constructor –packages together values
  - Multiple Constructors –different kinds of values
  - Parametrized Types