Haskell4Life:
Types and Typeclasses

Sergiu Ivanov

sergiu.ivanov@lacl.fr

Slides and code examples available online:

http://lacl.fr/~sivanov/doku.php?id=en:haskell_for_life

Warm Up

https://openclipart.org/

What is a class?

What is a type?

Outline

Outline

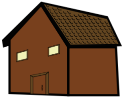
Classes and Types

A class/type is a set of objects sharing a property.

https://openclipart.org/

Classes and Types

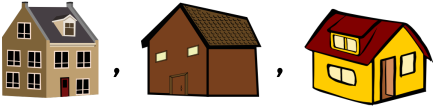
A class/type is a set of objects sharing a property.

house = { , , , . . . }

https://openclipart.org/

Classes and Types

A class/type is a set of objects sharing a property.

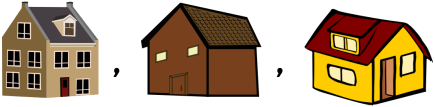
house = {  , . . . }

Inheritance = set inclusion

https://openclipart.org/

Classes and Types

A class/type is a set of objects sharing a property.

house = {  , ... }

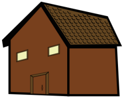
Inheritance = set inclusion

$\text{house} \subseteq \text{building}$

https://openclipart.org/

Classes and Types

A class/type is a set of objects sharing a property.

house = {  ,  ,  , $\ldots$ }

Inheritance = set inclusion

$\text{house} \subseteq \text{building}$

Every house is a building, but not every building is a house.

https://openclipart.org/

Typeclasses in Haskell: Vague Idea

A typeclass is a collection (class) of types.

Typeclasses in Haskell: Vague Idea

A typeclass is a collection (class) of types.

Typeclasses in Haskell $\neq$ classes in other languages.

Typeclasses in Haskell: Vague Idea

A typeclass is a collection (class) of types.

Typeclasses in Haskell $\neq$ classes in other languages.

Classes in other languages $\sim$ types in Haskell

The use of the word class to refer to different things is just a “coincidence”.

Outline

1. Vague Definitions
2. Algebraic Data Types
3. Parameterised and Recursive Data Types
4. Typeclasses

Defining a New Datatype

```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

Defining a New Datatype

type name

```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

Defining a New Datatype

data constructor

type name

```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

Defining a New Datatype

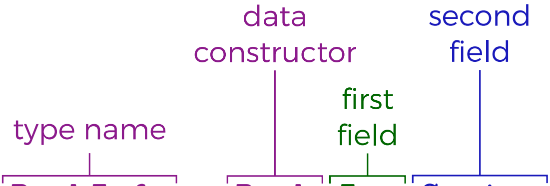
data
constructor

first
field

type name

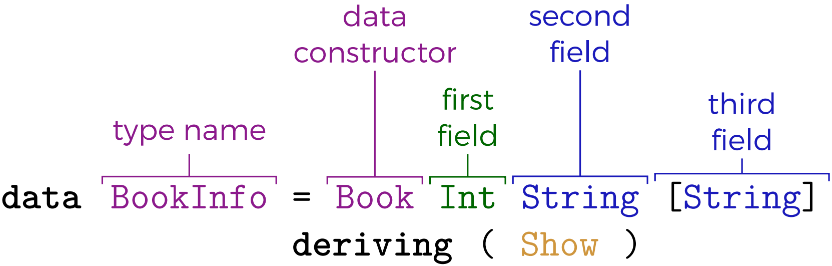
```
data BookInfo = Book Int String [String]
              deriving ( Show )
```

Defining a New Datatype




```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

Defining a New Datatype



Defining a New Datatype

```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

- type name: BookInfo
- data constructor: Book
- first field: Int
- second field: String
- third field: [String]
- automatically derive `show`: Show

Defining a New Datatype

data constructor

second field

first field

third field

type name

```
data BookInfo = Book Int String [String]
                deriving ( Show )
```

automatically derive `show`

`show` converts “anything” to `String`.

Constructing and Deconstructing a Value

```
λ> let hp = Book 1997 "Harry Potter"
                ["J. K. Rowling"]
```

Constructing and Deconstructing a Value

```
λ> let hp = Book 1997 "Harry Potter"
                ["J. K. Rowling"]

λ> let (Book year title authors) = hp
```

Constructing and Deconstructing a Value

```
λ> let hp = Book 1997 "Harry Potter"
              ["J. K. Rowling"]

λ> let (Book year title authors) = hp

λ> year
1997

λ> title
"Harry Potter"

λ> authors
["J. K. Rowling"]
```

Naming Conventions

Type names start with capital letters.

Naming Conventions

Type names start with capital letters.

Function and variable names start with lowercase letters.

Naming Conventions

Type names start with capital letters.

Function and variable names start with lowercase letters.

Type variables start with lowercase letters.

Naming Conventions

Type names start with capital letters.

Function and variable names start with lowercase letters.

Type variables start with lowercase letters.

Often the same name is used for the type and the data constructor.

```
data IntPair = IntPair Int Int
```

Type Synonyms

```haskell
type Year = Int
```

Type Synonyms

```haskell
type Year = Int
type Title = String
type Author = String
data BookInfo = BookInfo Year Title [Author]
```

Type Synonyms

```
type Year = Int
type Title = String
type Author = String
data BookInfo = BookInfo Year Title [Author]
```

Symbolic names are not enforced:

```
("J. K. Rowling" :: Author) :: String
```

is a valid expression.

Data Types vs. Tuples

```
type Cartesian2D = (Double,Double)
```

vs.

```
data Cartesian2D = Cartesian2D Double Double
```

Data Types vs. Tuples

```
type Cartesian2D = (Double,Double)
```

vs.

```
data Cartesian2D = Cartesian2D Double Double
```

Suppose we want to represent polar coordinates.

Data Types vs. Tuples

```
type Cartesian2D = (Double,Double)
```

vs.

```
data Cartesian2D = Cartesian2D Double Double
```

Suppose we want to represent polar coordinates.

```
type Polar2D = (Double,Double) == Cartesian2D
```

On the other hand,

```
type Polar2D = Polar2D Double Double /= Cartesian2D
```

Data Types vs. Tuples

```
type Cartesian2D = (Double,Double)
```

vs.

```
data Cartesian2D = Cartesian2D Double Double
```

Suppose we want to represent polar coordinates.

```
type Polar2D = (Double,Double) == Cartesian2D
```

On the other hand,

```
type Polar2D = Polar2D Double Double /= Cartesian2D
```

Data types allow distinguishing at compile time between semantically different types which have the same syntactical structure.

Algebraic Data Types

Algebraic data types have ≥ 1 data constructor.

Algebraic Data Types

Algebraic data types have ≥ 1 data constructor.

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

Algebraic Data Types

Algebraic data types have ≥ 1 data constructor.

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

```
type Mass = Double
type Volume = Double
data Cargo = Solid Mass | Liquid Volume
           | Gas Volume | Plasma Volume
```

Outline

Parameterised Data Types

```
data Pair a = Pair a a
```

Parameterised Data Types

```
data Pair a = Pair a a
```

```
λ> let p1 = Pair True False
λ> :type p1
p1 :: Pair Bool
```

Parameterised Data Types

```
data Pair a = Pair a a
```

```
λ> let p1 = Pair True False
λ> :type p1
p1 :: Pair Bool

λ> let p2 = Pair "Hello" "World"
λ> :type p2
p2 :: Pair String
```

The `Maybe` Type

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

Very often used to represent a container which may transport a data item or may be empty.

The List Type

The List Type

```
data List a = Cons a (List a) | Nil
          deriving (Show)
```

```
Nil
```

The List Type

```
data List a = Cons a (List a) | Nil
            deriving (Show)
```

```
      Nil
Cons 3 Nil
```

The List Type

```
data List a = Cons a (List a) | Nil
        deriving (Show)
```

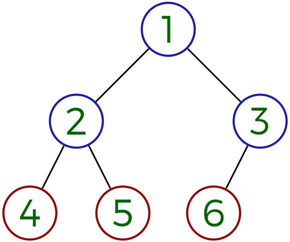
```
                Nil
         Cons 3 Nil
Cons 2 (Cons 3 Nil)
```

The List Type

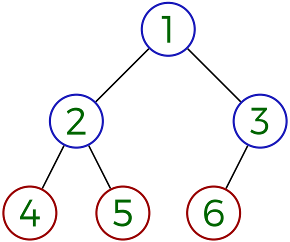
```
data List a = Cons a (List a) | Nil
            deriving (Show)
```

```
                    Nil
             Cons 3 Nil
      Cons 2 (Cons 3 Nil)
Cons 1 (Cons 2 (Cons 3 Nil))
```

The BinaryTree Type

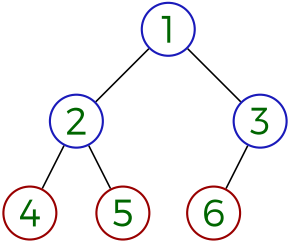


The BinaryTree Type




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

The BinaryTree Type




```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving (Show)
```

Outline

Typeclasses

A typeclass is a collection (class) of types.

Typeclasses

A typeclass is a collection (class) of types.

A typeclass specifies what properties a type must have.

Typeclasses

A typeclass is a collection (class) of types.

A typeclass specifies what properties a type must have.

A typeclass specifies which functions should be defined for a type.

An Example of a Typeclass

```
class BasicEq foo where
  isEqual :: foo -> foo -> Bool
```

A type `foo` belongs to the typeclass `BasicEq` if there exists a version of the function `isEqual` defined for it.

An Example of a Typeclass

```
class BasicEq foo where
  isEqual :: foo -> foo -> Bool
```

A type `foo` belongs to the typeclass `BasicEq` if there exists a version of the function `isEqual` defined for it.

One can say that, to verify whether a value of type `a` appears in a list of `a`, we need to be able to compare values of type `a` in the following way:

```
isElement :: BasicEq a => a -> [a] -> Bool
```

Saying That a Type Belongs to a Typeclass

```
instance BasicEq Bool where
  isEqual True True = True
  isEqual False False = True
  isEqual _ _ = False
```

Typeclasses Show and Read (in a nutshell)

```
class Show a where
  show :: a -> String

class Read a where
  read :: String -> a
```

Automatic Derivation of Instances

Instances of `Show`, `Read`, `Eq`, `Ord`, `Bounded`, `Enum` can be derived automatically.

```
data Maybe a = Just a | Nothing
                 deriving (Show,Read,Eq)
```