

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

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

Outline

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

Classes and Types

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

Classes and Types

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

house = {  ,  ,  , ... }

Classes and Types

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

house = {  ,  ,  , ... }

Inheritance = set inclusion

Classes and Types

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

house = {  ,  ,  , ... }

Inheritance = set inclusion

house \subseteq building

Classes and Types

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

house = {  ,  ,  , ... }

Inheritance = set inclusion

house \subseteq building

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

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

The diagram illustrates the structure of a Haskell data type definition. It shows the following components:

- type name**: The identifier `BookInfo` underlined by a horizontal line.
- data constructor**: The identifier `Book` positioned above the equals sign.
- data type definition**: The part `= Book Int String [String]`.
- deriving clause**: The part `deriving (Show)`.

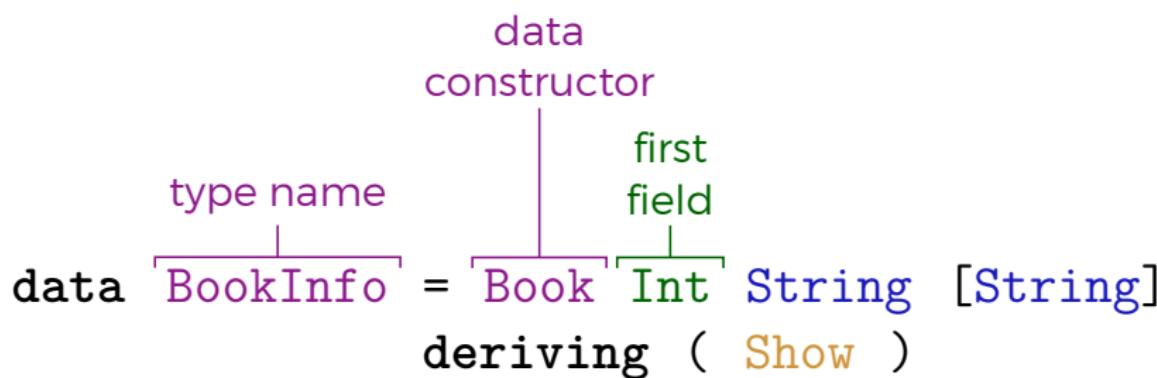
The entire definition is enclosed in a box, and the word `data` is placed above the box.

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

Defining a New Datatype

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

first field



Defining a New Datatype

A diagram illustrating the annotated components of a Haskell data type definition. The code is:

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

The annotations are:

- type name**: A bracket under `BookInfo`.
- data constructor**: A bracket under `Book`.
- first field**: A bracket under `Int`.
- second field**: A bracket under `String`.
- [String]**: A bracket under `[String]`.
- deriving (Show)**: A bracket under `(Show)`.

Annotations are color-coded: type name is purple, data constructor is pink, fields are green, and other annotations are blue.

Defining a New Datatype

```
type name          data constructor      second field      third field
|                   |                         |                   |
data BookInfo = Book Int String [String] deriving ( Show )
```

The diagram illustrates the structure of the Haskell data type definition. The type name 'BookInfo' is underlined, indicating it is a single identifier. The data constructor 'Book' is also underlined, indicating it is a single identifier. The fields 'Int', 'String', and '[String]' are grouped together by a bracket, indicating they form a tuple. Labels above the code identify these components: 'type name' for 'BookInfo', 'data constructor' for 'Book', 'first field' for 'Int', 'second field' for 'String', and 'third field' for '[String]'. The 'deriving (Show)' part is not labeled.

Defining a New Datatype

```
type name           data constructor      second field          third field
|                   |                         |                   |
data BookInfo = Book Int String [String]   first field
|           |           |           |
|           |           |           |
deriving ( Show )                           automatically derive show
```

The diagram illustrates the definition of a new datatype, `BookInfo`. It shows the type name `BookInfo` underlined, indicating it is a type. The definition begins with `data`, followed by the type name, an equals sign, and a constructor `Book`. This is followed by three fields: `Int`, `String`, and `[String]`. The first field is labeled `first field`, the second `second field`, and the third `third field`. Below the definition is the keyword `deriving` followed by `(Show)`, with `Show` underlined. A bracket below the entire definition is labeled `automatically derive show`.

Defining a New Datatype

```
type name          data constructor      second field      third field
|                   |                         |                   |
data BookInfo = Book Int String [String]
               first field
               |           |
               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

```
type Year = Int
```

Type Synonyms

```
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

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

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)
```

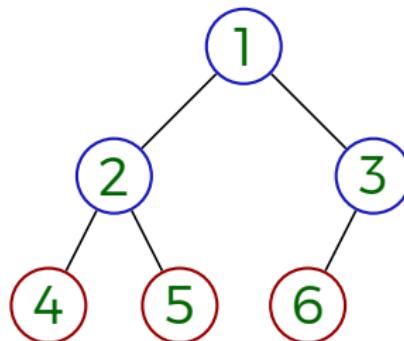
Nil

Cons 3 Nil

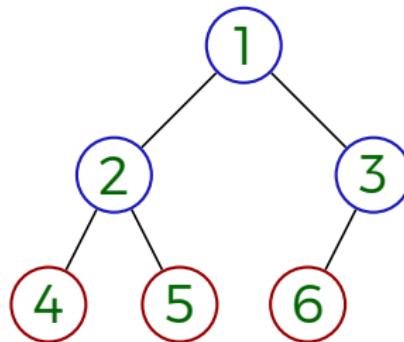
Cons 2 (Cons 3 Nil)

Cons 1 (Cons 2 (Cons 3 Nil))

The BinaryTree Type

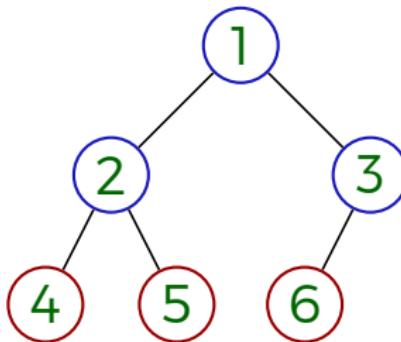


The BinaryTree Type



```
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

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

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)
```