Functional Programming in Haskell

The University of Glasgow

Get Unlimited learning

€23.99/month

TP

- Your learning
- Wishlist
- Profile
- Settings
- Account
- Sign out

Get Unlimited learning

€23.99/month

- To do
- Conversations
- Progress

Week 1

Haskell First Steps

Introduction

1.1

Welcome to the Course

6.3

You’ve completed 2 steps in Week 6

Type Classes in more Detail

Type class and instance declarations

Defining type classes

- A *type class* is a set of types for which some operations are defined.
- Haskell has some standard type classes that are defined in the Standard Prelude.
- You can also define your own.

A type for bright colors

Suppose we're computing with colors. Here's a type, and a couple of functions.

```
data Bright
  = Blue
  | Red
  deriving (Read, Show)

darkBright :: Bright -> Bool
darkBright Blue = True
darkBright Red  = False

lightenBright :: Bright -> Bright
lightenBright Blue = Red
lightenBright Red = Red
```

A type for milder colors

Now, suppose we have a different type that needs similar functions.

```
data Pastel
  = Turquoise
  | Tan
  deriving (Read, Show)

darkPastel :: Pastel -> Bool
darkPastel Turquoise = True
darkPastel Tan       = False

lightenPastel :: Pastel -> Pastel
lightenPastel Turquoise = Tan
lightenPastel Tan       = Tan
```

Defining a type class

- Both of our color types have functions to decide whether it's dark, or to lighten it.
- We can define a class *Color* and its corresponding functions.

```
class Color a where
  dark :: a -> Bool
  lighten :: a -> a
```

This says

- *Color* is a type class
- The type variable *a* stands for a particular type that is in the class *Color*
- For any type *a* in *Color*, there are two functions you can use: *dark* and *lighten*, with the specified types.

Defining instances for the type class

- An *instance* declaration says that a type is a member of a type class.
- When you declare an instance, you need to define the class functions.
- The following says that the type *Bright* is in the class *Color*, and for that instance, the *dark* function is actually *darkBright*.

```
instance Color Bright where
  dark = darkBright
  lighten = lightenBright
```

- Similarly, we can declare that *Pastel* is in *Color*, but it has different functions to implement the class operations.

```
instance Color Pastel where
  dark = darkPastel
  lighten = lightenPastel
```

Predefined type classes

Haskell provides several standard type classes. We have a look at two of them: *Num* and *Show*.

The Num class

- *Num* is the class of numeric types.
- Here is (part of) its class declaration:

```
class Num a where
  (+), (-), (*) :: a -> a -> a
```

Num instances

- There are many numeric types; two of them are *Int* and *Double*.
- There are primitive monomorphic functions that perform arithmetic on these types (these aren't the real names):

```
addInt, subInt, mulInt :: Int -> Int -> Int
addDbl, subDbl, mulDbl :: Double -> Double -> Double

instance Num Int where
  (+) = addInt
  (-) = subInt
  (*) = mulInt

instance Num Double where
  (+) = addDbl
  (-) = subDbl
  (*) = mulDbl
```

Hierarchy of numeric classes

Video
2.4
Do it Yourself: Logical Thinking
Exercise
2.5
Nothing but the Truth
Quiz
Input and Output
2.6
Why I/O?
Video
2.7
Do it Yourself: Input/Output
Exercise
2.8
I/O and a First Encounter with Monads
Article
Installing GHC
2.9
Installing Haskell for Yourself
Article
2.10
How to Run GHCi
Video
2.11
Guessing Game
Article

2.12
What do you know about Haskell?
Quiz
2.13
End of Week 2
Video
Week 3
Data Structures and Types
Functions on Lists
3.1
Welcome to Week 3
Video
3.2
Recursive Functions on Lists
Article
3.3
Functional Maps and Folds versus Imperative Loops
Video
3.4
Do it Yourself: Lists and Recursion
Exercise
3.5
Do it Yourself: Function Composition
Exercise
3.6
What Have We Learned About Lists?
Quiz
3.7

- There are some operations (addition) that are valid for all numeric types.
- There are some others (e.g. trigonometric functions) that are valid only for *some* numeric types.
- Therefore there is a rich hierarchy of subclasses, including
  - $Integral$ — class of numeric types that represent integer values, including Int, $Integer$, and more.
  - $Fractional$ — class of types that can represent fractions.
  - $Floating$ — class containing $Float$, $Double$, etc.
  - $Bounded$ — class of numeric types that have a minimal and maximal element.
  - $Bits$ — class of types where you can access the representation as a sequence of bits, useful for systems programming and digital circuit design.
- If you want to get deeply into numeric classes and types, refer to the Haskell documentation.

The Show class

- We have been using $show$ to convert a data value to a string, which can then be written to output.
- Some values can be "shown", but not all.
- For example, it is impossible in general to show a function.
- Therefore $show$ needs a type class!
- $show :: Show\, a \Rightarrow a \to \; String$

Defining your own Show instance

```
data Foo = Bar | Baz
```

We might want our own peculiar string representation:

```
instance Show Foo where
  show Bar = "it is a bar"
  show Baz = "this is a baz"
```

Recall that when you enter an expression exp into ghci, it prints $showexp$. So we can try out our strange instance declaration:

```
*Main> Bar
it is a bar
*Main> Baz
this is a baz
```

Deriving Show

This is a similar type, but it has a $deriving$ clause.

```
data Foo2 = Bar2 | Baz2
  deriving (Read, Show)
```

Haskell will automatically define an instance of $show$ for $Foo2$, using the obvious definition:

Write a Spelling Book Generator
Exercise
3.8

```
*Main> Bar2
Bar2
*Main> Baz2
Baz2
```

More standard typeclasses

Summary
Article

Here is a summary of some of the type classes defined in the standard libraries.

- *Num* — numbers, with many subclasses for specific kinds of number.
- *Read* — types that can be "read in from" a string.
- *Show* — types that can be "shown to" a string.
- *Eq* — types for which the equality operator $==$ is defined.
- *Ord* — types for which you can do comparisons like $<$, $>$, etc.
- *Enum* — types where the values can be enumerated in sequence; this is used for example in the notation $[1..n]$ and $'a'..'z'$.

```
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
*Main> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Custom Data Types
3.9
Define Your Own Data Types
Video
3.10
Grow a Tree
Article

Share this article:

3.11
Type Classes
Video
Haskell History
2 comments
3.12
Previous
Mark as complete
Interview with Simon Peyton Jones
Next
Mark as complete
Video
3.13

Comments

Brief History of Haskell
Article

TP
Tomas P.

Add a comment... (plain text and markdown available)

0/1200

Post Learn more about markdown

Article
3.15
End of Week 3

Show:
All comments

Sort by:

Newest

Filter

- GY

Gwee YC

Follow

31 OCT31 OCT

I think it can be coded in ways more than an interface can, but to me, it seems basically like overloading operators to deal with a set of meaningful objects the programmer wants to lump together, where the various ‘domains’ of the operations are all abstracted as the ‘one domain’ of the type class.

(edited)

Like

Reply

Bookmark

Report

- 

Xavier Góngora

Follow

20 OCT20 OCT

So a type class can be described equivalently as a:

1. Set of types (instantiated) with an implementation of a family of functions that describe the behavior of the class.
2. An interface to restrict a family of polymorphic function to a set of types.

Is this accurate? I’m trying to have my computer science concepts right.

Like

Reply

Bookmark

Report

- Help Centre
- Child safety
- Privacy
- T&Cs

Contact FutureLearn for Support

Our website is updated regularly so this content may now be out of date, please go to https://www.futurelearn.com for the most up to date information.