Type Classes

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Type Notations

- Use the `:t` (short for `:type`) command to view the type of an expression

```
ghci> :t "Hello"
"Hello" :: [Char]
ghci> :t 'a'
'a' :: Char
```

- A types in Haskell start with an uppercase letter
- `::` means **has type**
    - `"Hello"` has type `[Char]` (list of `Char`)
    - `'a'` has type `Char`
- `:t` works on functions as well

```
ghci> :t (++)
(++) :: [a] -> [a] -> [a]
```

- If a type starts with a lowercase character, it is a **type variable**
    - It can be replaced by virtually any type
    - In the above example, `a` can be `Char`, `Int`, etc

Type Notations

- Note that ++ has type `[a] -> [a] -> [a]`
- `->` indicates a function
- For ++ (concatenation):
  - takes two arguments of type `[a]` and `[a]`
  - returns a value of type `[a]`
- In case of ++, arguments and the return value must have the same type
- If a function has type `[a] -> [b] -> [c]`
  - Arguments can have different types
  - The type of the return value can be different as well
  - But they can also be the same type

Type Notations

- `:t` also works on a user-defined type
- Consider the following:

```
data MyIntPair = MyIntPair Int Int

myFirst  (MyIntPair x _)  = x
mySecond (MyIntPair _ y)  = y

incFirst i  (MyIntPair x y)  = MyIntPair (x + i)  y
incSecond i  (MyIntPair x y)  = MyIntPair x  (y + i)
```

- Let's check some types:

```
ghci> :t MyIntPair 1 2
MyIntPair 1 2 :: MyIntPair
ghci> :t myFirst
myFirst :: MyIntPair -> Int
ghci> :t incFirst
incFirst :: Int -> MyIntPair -> MyIntPair
```

- Use `:i` (short for `:info`) to view the detail about type

```
ghci> :i MyIntPair
data MyIntPair = MyIntPair Int Int      -- Defined at ...
```

Type Classes 101

- A **type class** is an interface that defines some behavior
- If a type is an **instance** of a type class, it supports and implements the behavior the type class describes
- Sound familiar? **Java interface**
- A type class specifies a set of functions
- To make a new type and instance of a type class, we need to define what those functions mean for this new type

Type Classes 101

- Let’s check the type of the == operator:

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Everything before the => symbol is called a **class constraint**
  - `Eq` is a type class
  - `a` is a type variable
- The == operator can be used with any types that implement the `Eq` type class

```
ghci> 5 == 2
False
ghci> "Hello" == "Hello"
True
ghci> 'a' == 'A'
False
```

The Eq Type Class

- The Eq type class is used for types that support equality testing
- Functions that an instant of the Eq type class must implement are == and /=
- An instance of the type class Eq must define what == and /= mean for the type
- Here are some example:

```
ghci> pi == pi
True
ghci> pi /= pi
False
ghci> 3.1415 == pi
False
ghci> "Hello" /= "World"
True
ghci> [1,2,3] == [1,2]
False
```

The Eq Type Class

- Let's look at some information about the `Eq` type class:

```
ghci> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
   :
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
instance Eq Double -- Defined in 'GHC.Classes'
instance Eq Char -- Defined in 'GHC.Classes'
instance Eq Bool -- Defined in 'GHC.Classes'
   :
```

- Functions that instances of the `Eq` type class must impelement are == and /=
- Some instances are `Int`, `Float`, `Double`, `Char`, ...

The `Ord` Type Class

- Values of a type which is an instance of the type class `Ord` can be put in some order
- Let's check the type of >:

```
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool
```

- Functions are `compare`, <, <=, >, >=, `max`, and `min`
- Some examples:

```
ghci> compare 5 2
GT
ghci> 3.14 < pi
True
ghci> "Hello" > "World"
False
ghci> max 'c' 'd'
'd'
ghci> min [1,2,3] [1,3]
[1,2,3]
```

The `Ord` Type Class

- Check the information about the `Ord` type class:

```
ghci> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
    :
```

- Because it requires the ability to compare whether two values are equal:
  - To be an instance of the `Ord` type class, the type must first be an instance of the `Eq` type class
  - Think about this as a prerequisite
- What about the type `Ordering`:

```
ghci> :info Ordering
data Ordering = LT | EQ | GT     -- Defined in 'GHC.Types'
    :
```

- `LT`, `EQ`, and `GT` are "less than", "equal to" and "greater than", respectively

The Show Type Class

- Values whose types are instances of the Show type class can be represented as strings
- One of the most common use function is the show function
- Examples:

```
ghci> :t show
show :: Show a => a -> String
ghci> show 3
"3"
ghci> show [1,2,5]
"[1,2,5]"
ghci> show ("Hello", False)
"(\"Hello\",False)"
ghci> show pi
"3.141592653589793"
```

The `Read` Type Class

- The `Read` type class in pretty much an opposite of the `Show` type class
- The `read` function takes a string and returns the value of a specific type

```
ghci> :t read
read :: Read a => String -> a
ghci> read "False" || True
True
ghci> read "1.2" + 3.4
4.6
ghci> read "[3,4,1,2]" ++ [5]
[3,4,1,2,5]
```

- Problem:

```
ghci> read "1.2"
*** Exception: Prelude.read: no parse
```

- It has no idea whether `"1.2"` should be `1.2 :: Float` or `1.2 :: Double`
- In previous example, type inferences help

The `Read` Type Class

- Type annotations are a way to explicitly tell Haskell what the type should be
- Use `::` (has type) at the end of an expression

```
ghci> read "1.2" :: Float
1.2
ghci> read "24" :: Double
24.0
ghci> (read "2" :: Float) * 3
6.0
ghci> read "(1,2,3)" :: (Int, Float, Double)
(1,2.0,3.0)
```

The Enum Type Class

- An instance of Enum type class is a sequentially ordered type
- Common functions are succ (successor) and pred (predecessor)
- An instance can be used with **list range**

```
ghci> [3..8]
[3,4,5,6,7,8]
ghci> succ 3
4
ghci> pred 8
7
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> succ LT
EQ
ghci> pred GT
EQ
ghci> succ 'H'
'I'
```

The Bounded Type Class

- An instance of the `Bounded` type class must have an upper bound and a lower bound
- Common functions are `minBound` and `maxBound`

```
ghci> minBound :: Int
-9223372036854775808
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Ordering
GT
ghci> minBound :: Bool
False
ghci> maxBound :: (Bool, Int, Ordering)
(True,9223372036854775807,GT)
```

- Let's look at the type of `minBound` and `maxBound`

```
ghci> :t minBound
minBound :: Bounded a => a
ghci> :t maxBound
maxBound :: Bounded a => a
```

- They are polymorphic constants (depend on types)

The `Num` Type Class

- Common instances are `Int`, `Integer`, `Float`, and `Double`
- Let's look at its information:

```
ghci> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
   :
```

- The `fromInteger` function converts an integer into a more general form

```
ghci> :t fromInteger (5 :: Integer)
fromInteger (5 :: Integer) :: Num a => a
```

The `Fractional` and `Integral` Type Classes

- `Fractional`
  - Instances are `Float` and `Double`
  - Common functions are / and `recip`

```
ghci> 1.2 / 3.4
0.35294117647058826
ghci> recip 4.5
0.2222222222222222
```

- `Integral`
  - Instances are `Integer` and `Int`
  - Common functions are `div` (integer division) and `rem` (remainder)

```
ghci> div 5 2
2
ghci> mod 5 2
1
```

Multiple Constraints

- Consider the type of `fromIntegral` function:

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

- Multiple class constraints are separated by commas
- It takes an instance of `Integral` type class and turn it into an instant of `Num` type class (more general)
- Useful when integral and floating-point need to work together

A Bit of Headache

- Let's check the type of + function:

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

- + can be used with a type `a` which is an instance of the `Num` type class
- It can perform addition between two value **of the same type**

```
ghci> (5 :: Int) + (2 :: Int)
7
ghci> (5 :: Int) + (2 :: Float)

<interactive>:93:15: error:
   :
```

- But the following expression is fine because of the type inference:

```
ghci> 5 + 2.0
7.0
ghci> :t 5 + 2.0
5 + 2.0 :: Fractional a => a
```

- `5` can be a `Fractional` type class (`Float` or `Double`)

A Bit of Headache

- Suppose we want to calculate the length of a list divided by 2

```
ghci> length [1,2,3,4,5] / 2

<interactive>:103:1: error:
   :
```

- What is/are the problem(s)?

```
ghci> :t length
length :: Foldable t => t a -> Int
ghci> :t (/)
(/) :: Fractional a => a -> a -> a
```

- `length` of a list has type `Int` which is not compatible with `/`
- Either use `div` or `fromIntegral`

```
ghci> div (length [1,2,3,4,5]) 2
2
ghci> fromIntegral (length [1,2,3,4,5]) / 2
2.5
```

Conclusions

- Type classes are similar to interfaces in Java
- A type class defines a set of functions
- An instance of a type class must implement those functions
- We can also think about them as type constraints
- We will learn how to create a type that will be an instance of a type class later