Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

October 13, 2016 Week 3 Lab1

LAB MANUAL: HASKELL TYPE SYSTEM

In this tutorial you will learn about

- Type System in Haskell
- Type Inference
- Basic Types in Haskell
- Types of Functions
- Type Classes

Haskell Type System

Haskell has a static type system. The type of every expression is known at compile time, which leads to safer code. If you write a program where you try to divide a boolean type with some number, it won't even compile. That's good because it's better to catch such errors at compile time instead of having your program crash.

Type Inference

Unlike Java or Pascal, Haskell has type inference. If we write a number, we don't have to tell Haskell it's a number. It can infer that on its own, so we don't have to explicitly write out the types of our functions and expressions to get things done.

Type

A type is a kind of label that every expression has. It tells us in which category of things that expression fits. The expression True is a boolean, "hello" is a string, etc.

```
Hugs> Hugs> :t 'a'
'a' :: Char
Hugs> :t True
True :: Bool
Hugs> :t "HELLO!"
"HELLO!" :: [Char]
Hugs> :t (True, 'a')
(True, 'a') :: (Bool, Char)
Hugs> :t 4 == 5
4 == 5 :: Bool
```

- `::` is read as "has type of"

Functions and their types

Functions in Haskell also have types. Recall the `doubleMe` function we wrote in first week

```
Hugs> doubleMe x = x + x
```

- What is the type of this function?
- How can we determine the type of a function?
- When writing our own functions, we can choose to give them an explicit type declaration. This is generally considered to be good practice except when writing very short functions.
- Write the following code in an editor and save it with a .hs extension

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

- Can you guess what this function is supposed to do?
- `removeNonUppercase` has a type of `[Char] -> [Char]`, meaning that it maps from a `string` to a `string`. That's because it takes one string as a parameter and returns another as a result.
- Now comment out the first line using `– (Single line comments)`
- Check the type of the function `removeNonUppercase`
- Compiler can infer the type of simple functions

Functions with multiple arguments

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

The parameters are separated with `->` and there's no special distinction between the parameters and the return type. The return type is the last item in the declaration and the parameters are the first three.

Common Data types in Haskell

- `Int`: It is bounded, which means that it has a minimum and a maximum value. Usually on 32-bit machines the maximum possible Int is 2147483647 and the minimum is -2147483648.
- `Integer`: It's not bounded so it can be used to represent really big numbers
- `Float`: It is a real floating point with single precision.
- `Double`: It is a real floating point with double precision.
- `Bool`: It is a boolean type. It can have only two values: `True` and `False`.
- `Char`: It represents a character. It's denoted by single quotes. A list of characters is a string.

Note: Types in Haskell are written in Capital case.

Type variables

- What is the type of the `head` function?

```
Hugs> :t head
head :: [a] -> a
```

- What is this `a`? Is it a type?
- It is called a Type Variable
- a can be of any type. It is much like generics in other languages
- Help in writing Polymorphic Functions.
- The type declaration of head states that it takes a list of any type and returns one element of that type

```
Hugs> :t fst
fst :: (a, b) -> a
```

`fst` takes a tuple which contains two types and returns an element which is of the same type as the pair's first component. Note that just because a and b are different type variables, they don't have to be different types. It just states that the first component's type and the return value's type are the same.

Typeclasses

- They are like interfaces that define some behaviour
- If a type is a part of a typeclass, that means that it supports and implements the behaviour the typeclass describes.
- Look at the type of the == operator

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

- Everything before the => symbol is called a **class constraint**.
- The equality function takes any two values that are of the same type and returns a `Bool`
- The type of those two values must be a member of the **`Eq`** class

Typeclasses in Haskell

1. `Eq`
2. `Ord`
3. `Show`
4. `Read`
5. `Enum`
6. `Bounded`

7. Num
8. Integral
9. Floating

Eq is used for types that support equality testing. The functions its members implement are == and /=. So if there's an Eq class constraint for a type variable in a function, it uses == or /= somewhere inside its definition.

```
Hugs> 5 == 5
True
Hugs> 5 /= 5
False
Hugs> 'a' == 'a'
True
Hugs> "Ho Ho" == "Ho Ho"
True
Hugs> 3.432 == 3.432
True
```

Ord is for types that have an ordering.

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

- Ord covers all the standard comparing functions such as >, <, >= and <=

```
Hugs> "Lion" < "Zebra"
True
Hugs> "Lion" `compare` "Zebra"
LT
Hugs> 5 >= 2
True
Hugs> 5 `compare` 3
GT
```

- How can we check the type of `compare` function?
- `compare` is an infix function as it is coming in between both of its arguments.

```
Hugs> :t (compare)
compare :: Ord a => a -> a -> Ordering
```

- The compare function takes two Ord members of the same type and returns an **ordering**.

- Ordering is a type that can be **GT**, **LT** or **EQ**, meaning greater than, lesser than and equal, respectively.
- To be a member of Ord, a type must first have membership in the prestigious and exclusive Eq club

Show: Members of Show can be presented as strings. All types covered so far except for functions are a part of Show. The most used function that deals with the Show typeclass is show. It takes a value whose type is a member of Show and presents it to us as a string.

```
Hugs> show 3
"3"
Hugs> show 5.334
"5.334"
Hugs> show True
"True"
Hugs> :t show
show :: Show a => a -> String
```

Read is sort of the opposite typeclass of Show. The read function takes a string and returns a type which is a member of Read.

```
Hugs> read "True" || False
True
Hugs> read "8.2" + 3.8
12.0
Hugs> read "5" - 2
3
Hugs> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
Hugs> :t read
read :: Read a => String -> a
```

- What happens when we execute the following

```
Hugs> read “2”
```

- Compiler is unable to resolve the expression as it can’t infer the return type from the given expression
- We can use explicit type annotations. **Type annotations** are a way of explicitly saying what the type of an expression should be.
- It can be done by adding :: at the end of the expression and then specifying a type

```
Hugs> 5
Hugs> read "5" :: Float
5.0
Hugs> (read "5" :: Float) * 4
20.0
Hugs> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
Hugs> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

`Enum` members are sequentially ordered types — they can be enumerated. The main advantage of the `Enum` typeclass is that we can use its types in list ranges. They also have defined successors and predecesors, which you can get with the `succ` and `pred` functions. Types in this class: `(), Bool, Char, Ordering, Int, Integer, Float` and `Double`.

```
Hugs> succ 3
4
Hugs> pred 5
4
Hugs> :t succ
succ :: Enum a => a -> a
```

`Bounded` members have an upper and a lower bound

```
Hugs> minBound :: Int
-2147483648
Hugs> maxBound :: Char
'\1114111'
Hugs> maxBound :: Bool
True
Hugs> minBound :: Bool
False
Hugs> :t minBound
minBound :: Bounded a => a
```

- All tuples are also part of `Bounded` if the components are also in it.

```
Hugs> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Num is a numeric typeclass. Its members have the property of being able to act like numbers.

```
Hugs> :t 20
20 :: (Num t) => t
Hugs> 20 :: Int
20
Hugs> 20 :: Integer
20
Hugs> 20 :: Float
20.0
Hugs> 20 :: Double
20.0
Hugs> :t (*)
(*) :: (Num a) => a -> a -> a
```

- It takes two numbers of the same type and returns a number of that type.
- `(5 :: Int) * (6 :: Integer)` will result in a type error whereas `5 * (6 :: Integer)` will work just fine and produce an Integer because `5` can act like an `Integer` or an `Int`.

`Integral` is also a numeric typeclass. `Num` includes all numbers, including real numbers and integral numbers, `Integral` includes only integral (whole) numbers. In this typeclass are `Int` and `Integer`.

`Floating` includes only floating point numbers, so `Float` and `Double`.

What will be the output of the following expressions

- `length [1,2,3,4] + 3.2`
- Check the type of length function and try to guess
- Now try `fromIntegral (length [1,2,3,4]) + 3.2`
- `fromIntegral` takes an integral number and turns it into a more general number
- How can you check the type of `fromIntegral?`

References

- http://learnyouahaskell.com/types-and-typeclasses