Types and Classes II

1 Simple functions

```
foo :: Int -> Int
foo x = x + 3
```

Two parts

1. Type declaration `foo :: Int -> Int`
2. Definition `foo x = x + 3`

Any type declaration you write will be checked by the type inference engine. Error if incorrect

Reasoning about types is a core part of understanding (and writing) Haskell code so always decorate function definitions with their type

Syntax conventions:

- Function application is so important that it is written as quietly as possible: with whitespace
- All functions are called in prefix form: "`foo a b`" not "`a foo b`"
- But there is special syntax for binary functions

2 Binary functions

All binary functions (which have type `a -> b -> c`) can be written as infix functions

2.1 Symbol only names

Names consisting only of symbols

```
1 + 2 -- infix notation
(+) 1 2 -- prefix notation
False && True  -- infix notation
(&&) False True -- prefix notation
```

2.2 "Normal" names

Names with alpha-numeric characters

```
mod 3 2 -- prefix notation
3 `mod` 2 -- infix notation using backticks
```

3 Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
-- you always have to have the else branch
-- avoids ambiguity problems in the type checker
```

4 Guarded Equations

As an alternative to conditions, functions can also be defined using guarded equations

```
abs n | n >= 0    =n
      | otherwise = -n
```

Otherwise is True

5 Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments

```
not :: Bool -> Bool
not False = True
not True = False
```

Functions can often be defined in many different ways using pattern matching. For example

```
(&&) :: Bool -> Bool -> Bool
True && True = True
False && False = False
False && True = False
False && False = False
```

Or more compactly

```
True && True = True
_    && _    = False
```

And more efficiently

```
True  && b = b
False && _ = False
```

_ is a wildcard - matches anything

6 Polymorphism

Recall that haskell is strictly typed, which looks like a problem for length

```
length [True, False, True] -- :: [Bool] -> Int ?
length [1,2,3] -- :: [Int] -> Int
```

Polymorphic types solve this problem

```
Prelude> :type length
length :: [a] -> Int
```

This says that length takes a list of values of any type a and returns an int

6.1 Contrast with OO Languages

Definition: Parametric Polymorphism

Write a single implementation of a function that applies generically and identically to values of any type

Definition: ”ad-hoc” polymorphism

Write multiple implementations of a function, one for each type you wish to support

Definition: Subtype polymorphism

Relate datatypes by some ”substitutability”. Write a function for a supertype instance. Now all subtypes can use it. This is duck typing

7 Haskell Classes

Important: Class and instance

The words class and instance are the same as in object oriented programming languages, but their meaning is very different

Definition: Class

A collection of types that support certain, specified overloaded operations called methods

Definition: Instance

A concrete type that belongs to a class and provided implementations of the required methods

- Compare: type "a collection of related values"
- This is not like subclassing and inheritance in Java/C++
- Closest to a combination of Java interfaces and generics
- Also similar to C++ "Concepts"

7.1 Type classes

```
Num -- Numeric Types
Eq -- Equality Types
Ord -- Ordered Types
```

7.2 Defining Classes

We define the interface the type should support

```
class Foo a where:
        isfoo :: a -> Bool
```

Now we say how types implement this

```
instance Foo Int where
        isfoo _ = False

instance Foo Char where
        isfoo c = c `elem` ['a' .. 'c']
```

- Can add new interfaces to old types, and new types to old interfaces
- Contrasted to Java, where if I implement a new interface it is very difficult to make existing classes implement it
- Classes (interfaces) can provided default implementation
- Example, the `Eq` class representing equality requires both (==) and (/=)
- Since `a == b` ⇔ `not (a /= b)`, we can provide default implementations and only require that an instance implements one

```haskell
class Eq a where
        (==) :: a -> a -> Bool
        x == y = not (x /= y)
        (/=) :: a -> a -> Bool
        x /= y = not (x == y)

-- instance for MyType only needs to provide one of (==) or (/=)
instance Eq MyType where
        x == y = ..
```