Notes on Functional Programming in Haskell

from various online and offline resources

2015

Contents

1 Typeclasses

1.1 Introduction

Typeclasses offer a mechanism for dealing with *ad-hoc* polymorphism which "occurs when a function is defined over several different types, acting in a different way for each type" in contrast to *parametric* polymorphism, which happens when a function is defined over a range of types and acts in the *same* way for each type i.e. the `length` function. [Wadler and Blott, 1988]

Typeclasses may be "thought of as a kind of bounded quantifier, limiting the types that a type variable may instatiate to" where *type coercion* is not allowed. They may also be thought of as *abstract data types* where each type specifies a number of methods (functions) but does not say how they are to be implemented (that is left up to the individual types). [Wadler and Blott, 1988]

1.2 Declaring Typeclasses

Typeclass names and constructors begin with a capital letter, i.e. `Num, Real, Float` while *type variables* are given by lower case letters i.e. `a, b, ...`

A typeclass is declared to have one or more methods (functions); the methods may, or may not, have default implementations. For example, if the `Num` typeclass has the following declaration:

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

it essentially declares that any type `a` belonging to the typeclass `Num` will have an *instance* declaration that provides implementations for each of the typeclass `Num`'s declared methods. For example, the `Int` type is added to the `Num` typeclass by declaring an `instance` as follows:

```
instance Num Int where
    (+)    = addInt
    (*)    = mulInt
    negate = negateInt
```

where the type `Int` replaces the type variable, `a`, and `addInt`, `mulInt` and `negateInt` are all functions that perform the required behaviour.

The standard prelude includes similar instances for all the number types so that `+,*,/`, etc. work across all `Num` types.

To see the full declaration for any typeclass, load up `ghci` and enter `:info` or `:i`

```
Prelude> :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
    -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int     -- Defined in 'GHC.Num'
instance Num Float   -- Defined in 'GHC.Float'
instance Num Double  -- Defined in 'GHC.Float'
```

1.3 Adding Class Constraints (Subclasses)

If we want every member of a typeclass to also be a member of another typeclass we can add a *typeclass constraint* to the typeclass declaration. For example, the `Integral` class is declared as:

```
class (Real a, Enum a) => Integral a where
    ...
```

which states that any type `a` that belongs to the `Integral` typeclass must also belong to the `Real` and `Enum` typeclasses. And that essentially means that *all* `Integral` types also have the behaviours (methods) of all `Real` and `Enum` types.

Bibliography

Philip Wadler and Steven Blott. How to make ad-hoc polymorphism less ad-hoc. October 1988. URL http://202.3.77.10/users/karkare/courses/2010/cs653/Papers/ad-hoc-polymorphism.pdf.