5 Types and typeclasses

Table of contents

[TOC]

Creating data types

Enumeration

Like in many other programming language, Haskell allows the definition of an **enumeration type**. Enumerated types introduce a new type and an associated set of elements, known as **value constructors**, of that type, for example

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

The type name and value constructors both have to be capital-cased, and the constructors must be unique. This is to allow Haskell to infer the type of a value if it is passed to a function, which is impossible if the same constructor belongs to two different types.

We can also pattern match against value constructors as with any other type:

```haskell
data Switch = Off | On

bothOn :: Switch -> Switch -> Bool
bothOn On On = True
bothOn On Off = False
bothOn Off On = False
bothOn Off Off = False
```

Algebraic data types

Enumerations are only a special case of Haskell's more general **algebraic data types**, which has the following general form

```
data AlgDataType = Constr1 Type11 Type12 ... Type 1i
                 | Constr2 Type21 Type22 ... Type 2j
                 | Constr3 Type31 Type32 ... Type 3k
                 .
                 .
                 .
```

We see that when we write a value constructor, we can optionally add some types after the constructor; these types define the values it will contain. The literal values in enumerated types can be thought of as constructors with no types.

For example, we can implement the following algebraic data types

```
-- Point represents coordinate in 3D space
data Point = Point Float Float Float
-- Circle constructor takes Point for centre and radius
-- Rectangle constructor takes two 'Point's for diagon-
-- ally opposite vertices of a rectangle
data Shape = Circle Point Float | Rectangle Point Point

-- A type alias is NOT the same as a user-defined type
-- Name is just a label for String
type Name = String
data Person = Firstname Name | Lastname Name

data Time = Hours Int | Minutes Int | Seconds Int
```

Pattern matching

With algebraic data types, we can revisit pattern matching in more depth. In general, the following grammar defines what can be used in a pattern

```
pat ::=
     | _
     | var
     | var @ ( pat )
     | (Constructor pat1 pat2 ... patn)
```

The first line shows that an underscore is a pattern, which is used for a field or constructor we don't care about.

The second line shows that a variable by itself is a pattern and binds the given variable name to the matches value.

The third line specifies @-patterns, which is used to maintain a reference to the entire value being matched.

The fourth line shows that a constructor name followed by a sequence of patterns is itself a pattern

Record syntax

Record syntax allows us to quickly extract the contents of value constructors, especially where many constructors are used.

For instance, suppose we defined the following data type for `Person`:

```haskell
-- firstname, lastname, age, height, phone number, address
data Person = Person String String Int Float String String
```

So far, if we wanted to extract a certain value in `Person`, we would need to pattern match for every single value

```haskell
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> String
age (Person _ _ age _ _ _) = age

...
```

To avoid doing this, we can write data types using record syntax as follows:

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , address :: String
                     }
```

With this syntax, Haskell automatically creates functions that performs a lookup for any field in the data type. Unlike pattern matching, the fields do not have to be defined in order.

Polymorphic data types

We can further generalise algebraic data types to **polymorphic data types**, which can take other types as parameters to produce new types.

For example, suppose that we wanted to define our own `List` type; we don't want to create separate types only for `Int`, or `Bool`, or `String` etc., so we introduce a **type parameter**

```haskell
-- E represents empty list
-- C represent cons (:) for prepending to list
```

```
data List t = E | C t (List t)
```

This is also an example of a **recursive data type** (see below), where value constructors may be defined in terms of the type itself.

A **type constructor** can take types as parameters to produce new types. In the example above, `List` is a type constructor. It is *not* a type in and of itself; `List Int`, `List String` etc. are different types.

Recursive data types

We can define constructors for a type with the type constructor to create a **recursive data type**.

For instance, if we wanted to represent the natural numbers using the Peano axiomatisation, we might write

```
data Nat = Zero | Succ Nat deriving (Eq, Ord, Show)

-- For example
one = Succ Zero
two = Succ (Succ Zero)
```

Lists in Haskell can also be thought of as a recursive data type

```
data [a] = [] | a : [a]
```

although **this is not correct syntax**; lists are implemented in a different way under the hood.

Two useful ADTs

`Maybe`

The `Maybe a` type is an important data type in Haskell, defined as follows

```
data Maybe a where
    Nothing :: Maybe a
    Just    :: a -> Maybe a
```

`Maybe` is frequently used to represent computations that may fail in a recoverable way. For example, we can use `Maybe` to implement safe division:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv m 0 = Nothing
```

```haskell
safeDiv m n = Just (div m n)
```

`Either`

The `Either` type is useful for giving more information on the result of a failed computation compared to `Maybe`.

```haskell
data Either a b = Left a | Right b
```

By convention, `Left` is passed some information about a *failed* computation, and `Right` is passed the result of a *successful* computation.

Typeclasses

A **typeclass** is a set of types which have certain operations defined for them. Specifying what typeclass a particular type is an instance of (known as a **class constraint**) gives more information on the operations that can be performed on a type passed to a function.

For example, if we wanted to created a general `equal` function that is able to determine whether two values of any type are equal, we might have the signature

```haskell
equal :: a -> a -> Bool
```

But we know absolutely nothing about type `a`; it may not be a type which has a clear concept of equality.

To implement this sort of functionality, we use the `Eq` typeclass, which is used for types that have some notion of equality:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x == y = not (x /= y)
    x /= y = not (x == y)
```

The individual functions (or operators in this case) defined in a typeclass are known as **methods**.

The type variable `a` represents the type that we will make an instance of `Eq`. Notice that because `==` was defined in terms of `/=` and `/=` was defined in terms of `==`, we only need to give a definition for either `==` or `/=`.

For example, for the type `TrafficLight`,

```haskell
data TrafficLight = Red | Yellow | Green
```

we can make it an instance of `Eq` as follows

```haskell
instance Eq TrafficLight where
    (==) :: TrafficLight -> TrafficLight -> Bool
    Red == Red       = True
    Green == Green   = True
    Yellow == Yellow = True
    _ == _           = False
```

GHC is then able to derive `\=` from the definition for `==`.

So if we wanted to define our own function for checking whether two lists are equal, we need to include the class constraint `Eq a` to the type signature

```haskell
listEqual :: Eq a => [a] -> [a] -> Bool
```

If needed, we can also include class constraints on parameters of algebraic data types

```haskell
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

We can also include class constraints in class declarations to make one typeclass a sub-class of another typeclass, for example `Ord` which is a sub-class of `Eq`

```haskell
class Eq a => Ord a where
...
```

However, typeclass constraints should never be added to data declarations. This is because the same constraint in the type declaration will have to be included anyway in function signatures that use the user-defined data type.

Derived instances

Instead of manually implementing the functions defined in a typeclass for some type, we can make a type become an instance of a typeclass using the `deriving` keyword, for example

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Eq, Ord, Show)
```

If an enumerated type is made an instance of `Ord`, earlier values are considered smaller than later values in the type declaration.

Important typeclasses

`Eq`

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Used for types with values that are equatable.

`Ord`

```haskell
class Eq a => Ord a where
    (<=) :: a -> a -> Bool

    (<) :: a -> a -> Bool
    x < y = x <= y && x /= y

    (>) :: a -> a -> Bool
    (>) = flip (<)
    (>=) :: a -> a -> Bool
    (>=) = flip (<=)

    compare :: a -> a -> Ordering
    compare x y
        | x < y     = LT
        | x == y    = EQ
        | otherwise = GT
```

Used for types that can be ordered (and have already derived `Eq`).

`Show`

```haskell
class Show a where
    show :: a -> String
```

Used for types that can be converted to `String`.

`Read`

```haskell
class Read a where
    read :: String -> a
```

Use for type which can be converted from `String`.

In general, instances of `Read` and `Show` should satisfy `read . show = id`, but this is difficult to do manually, so `Show` and `Read` generally should be derived.

`Num`

```
class Ord a => Num a where
    fromInteger :: Integer -> a
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a

    abs :: a -> a
    signum :: a -> a
```

Used for numerical types.

Advanced topics

Generic algebraic data types (GADTs)

A **generic algebraic data type (GADT)** extends the functionality of other algebraic data types by providing type signatures for the value constructors. For example, suppose we define an `Expr` type for representing arithmetic expressions

```
data Expr a = I Int
            | B Bool
            | Add (Expr a) (Expr a)
            | Mul (Expr a) (Expr a)
            | Eq (Expr a) (Expr a)
```

If we wanted to write an `eval` function for `Expr`, we have to take into account that expressions can now represent integers or Booleans, so we might write the following signature for `eval`:

```
eval :: Expr a -> a
```

But then if we try to extract the value from `I`

```
eval (I n) = n
```

this won't type check since the compiler does not know that encountering `I` means `a = Int`. This is because `a` can be anything, so `I 5 :: Expr String` is still valid here.

GADTs allow us to restrict the return types of the constructors themselves, helping to improve type safety.

To use GADTs, add the `{-# LANGUAGE GADTs #-}` pragma to the beginning of the file.

We can then define `Expr` as

```haskell
data Expr a where
    I :: Int -> Expr Int
    B :: Bool -> Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Mul :: Expr Int -> Expr Int -> Expr Int
    Eq :: Eq a => Expr a -> Expr a -> Expr Bool
```

As another example, consider the `head` function for extracting the first element in a list. Normally, we would need to pattern match for when we have an empty list, and return a `Maybe`:

```haskell
head :: [a] -> Maybe a
head [] = Nothing
head (a : as) = Just a
```

Returning a `Maybe` makes it very inconvenient to use this function, so we would ideally like to type check so that `head` only accepts a list with at least one element.

We begin by defining a data type `SafeList`

```haskell
data Empty
data NonEmpty

data SafeList a b where
    Nil :: SafeList a Empty
    Cons :: a -> SafeList a b -> SafeList a NonEmpty
```

Note that with ordinary algebraic data types, we wouldn't be able to do this since all value constructors must return the same type of list.

We then define a `safeHead` function that can only be called on non-empty lists.

```haskell
-- Cannot pass an empty SafeList
safeHead :: SafeList a NonEmpty -> a
safeHead (Cons x _) = x
```

Therefore, if we try to pass an empty list to `safeHead`, it will return a compilation error.

Existentially quantified types

Existential quantification allows type variables to be used in constructors without having to include them in the data type itself.

For example, we may want to create a heterogenous list whose members all implement `Show`, for example

```
data Obj = forall a. (Show a) => Obj a

instance Show Obj where
    show :: Obj -> String
    show (Obj x) = show x

doShow :: [Obj] -> String
doShow = concatMap show
```

We can also think of existentially quantified type as an infinite collection of types, for example

```
data ShowBox = forall s. Show s => SB s
```

can be conceptualised as

```
data ShowBox = SBUnit | SBInt Int | SBBool Bool | SBIntList [Int] | ...
```

Similarly, if `x` has the following type signature

```
x :: forall a. Num a => a
```

we can think of it as a value with type `Int` and `Double` and `Float` etc.

Kinds

A **kind** is the type of a type constructor. For example, we `Int` has kind `*`. A single `*` indicates that the type is a concrete type; it doesn't take any type parameters. `Maybe` has type `* -> *` since it takes a concrete type like `String` and returns `Maybe String`. `Either` has kind `* -> * -> *` and so on.