Type

From HaskellWiki

In Haskell, **types** are how you describe the data your program will work with.

Contents

Data declarations

One introduces, or declares, a type in Haskell via the `data` statement. In general a data declaration looks like:

```
data [context =>] type tv1 ... tvi = con1  c1t1 c1t2... c1tn |
                       ... | conm cmt1 ... cmtq
                     [deriving]
```

which probably explains nothing if you don't already know Haskell!

The essence of the above statement is that you use the keyword `data`, supply an optional context, give the type name and a variable number of type variables. This is then followed by a variable number of constructors, each of which has a list of type variables or type constants. At the end, there is an optional `deriving`.

There are a number of other subtleties associated with this, such as requiring parameters to the data constructors to be eager, what classes are allowed in the deriving, use of field names in the constructors and what the context actually does. Please refer to the specific articles for more on each of those.

Let's look at some examples. The Haskell standard data type Maybe is typically declared as:

```
data Maybe a = Just a | Nothing
```

What this means is that the type **Maybe** has one type variable, represented by the *a* and two constructors **Just** and **Nothing**. (Note that Haskell requires type names and constructor names to begin with an uppercase letter). The **Just** constructor takes one parameter, of type *a*.

As another example, consider binary Trees. They could be represented by:

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Here, one of the constructors, **Branch** of **Tree** takes two trees as parameters to the constructor, while **Leaf** takes a value of type *a*. This type of recursion is a very common pattern in Haskell.

Type and newtype

The other two ways one may introduce types to Haskell programs are via the `type` and `newtype` statements.

`type` introduces a synonym for a type and uses the same data constructors. `newtype` introduces a renaming of a type and requires you to provide new constructors.

When using a `type` declaration, the type synonym and its base type are interchangeble almost everywhere (There are some restrictions when dealing with instance declarations). For example, if you had the declaration:

```
type Name = String
```

then any function you had declared that had `String` in its signature could be used on any element of type `Name`

However, if one had the declaration:

```

 newtype FirstName = FirstName String
```

this would no longer be the case. Functions would have to be declared that actually were defined on **FirstName**. Often, one creates a deconstructor at the same time which helps alleviate this requirement. e.g.:

```
 unFirstName :: FirstName -> String
 unFirstName (FirstName s) = s
```

This is often done by the use of field labels in the `newtype`. (Note that many consider the Haskell field implementation sub-optimal, while others use it extensively. See Programming guidelines and Future of Haskell)

A simple example

Suppose you want to create a program to play bridge. You need something to represent cards. Here is one way to do that.

First, create data types for the suit and card number.

```
data Suit = Club | Diamond | Heart | Spade
    deriving (Read, Show, Enum, Eq, Ord)

data CardValue = Two | Three | Four
    | Five | Six | Seven | Eight | Nine | Ten
    | Jack | Queen | King | Ace
   deriving (Read,  Show, Enum, Eq, Ord)
```

Each of these uses a deriving clause to allow us to convert them from / to String and Int, test them for equality and ordering. With types like this, where there are no type variables, equality is based upon which constructor is used and order is determined by the order you wrote them, e.g. `Three` is less than `Queen`.

Now we define an actual `Card`

```

data Card = Card {value :: CardValue,
                  suit :: Suit}
   deriving (Read, Show, Eq)
```

In this definition, we use fields, which give us ready made functions to access the two parts of a `Card`. Again, type variables were not used, but the data constructor requires its two parameters to be of specific types, `CardValue` and `Suit`.

The deriving clause here only specifies three of our desired classes, we supply instance declarations for Ord and Enum:

```
instance Ord Card where
     compare c1 c2 = compare (value c1, suit c1) (value c2, suit c2)

instance Enum Card where
     toEnum n  = let (v,s) = n`divMod`4 in Card (toEnum v) (toEnum s)
     fromEnum c = fromEnum (value c) * 4 + fromEnum (suit c)
```

Finally, we alias the type `Deck` to a list of `Card`s and populate the deck with a list comprehension

```
type Deck = [Card]

deck::Deck
deck = [Card val su | val <- [Two .. Ace], su <- [Club .. Spade]]
```

Please add

Further illustrative examples would be most appreciated.

See also

Read the articles about data constructors and classes. As well the Haskell 98 report (http://haskell.org/definition/haskell98-report.pdf) and your chosen implementation (e.g. GHC/Documentation) have the latest words.

- Determining the type of an expression - Let the Haskell compiler compute the type of expression
- Language extensions - many language extensions are to do with changes to the

type system.
- Smart constructors shows some interesting examples including a non-trivial usage of `newtype`.
- Unboxed type shows ways to have values closer to the bare metal :).
- Phantom type discusses types without constructors.
- Type witness gives an example of GADTs, a GHC extension.
- Existential type shows how to implement a common O-O programming paradigm.
- Type arithmetic implements the Peano numbers.
- Reified type, Non-trivial type synonyms, Abstract data type, Concrete data type, Algebraic data type.
- Research_papers/Type_systems allow the curious to delve deeper.

Languages: ja