Chapter 3 - Defining types, streamlining functions

October 3, 2017

1 Defining types

Here is an example of a data type

```
data BookInfo = Book Int String [String]
  deriving (Show)
```

BookInfo is a *type constructor* an is the name of the type.It must start with a capital letter.

Book is a *value constructor*, used to create a value of the BookInfo type. It must also start with a capital letter. It can be treated as a function, where the type is

$Book :: Int \rightarrow String \rightarrow [String] \rightarrow BookInfo$

The names of types and values are independent of each other. Therefore, we could rename BookInfo as Book and it would still be valid.

The Int, String and [String] are the components of the type, like fields of a class in an object oriented language.

```
data MagazineInfo = Magazine Int String [String]
  deriving (Show)
```

This type has the same structure as the other one but is a distinct type, because the type and value constructors have different names.

Both types are instances of the Show typeclass. We can use the show function to print their values to the screen.

2 Type synonyms

We can give another name to any type, as a *type synonym*, eg:

type *CustomerID* = *Int*
type *ReviewBody* = *String*

and form types with them:

data *BookReview* = *BookReview BookInfo CustomerID ReviewBody*

and other synonyms:

type *BookRecord* = (*BookInfo*, *BookReview*)

3 Algebraic Data Types

An *algebraic data type* can have more than one constructor, such as bool:

data *Bool′* = *True′* | *False′*

Each constructor is a case (or alternative) used to create a value of the type. bBy putting an algebraic data type in a module and exporting certain functions, one can make an algebraic data type into an ADT (abstract data type).

Each value can take zero or more arguments. Eg.

type *CardHolder* = *String*
type *CardNumber* = *String*
type *Address* = [*String*]

data *BillingInfo* = *CreditCard CardNumber CardHolder Address*
| *CashOnDelivery*
| *Invoice CustomerID*
deriving (*Show*)

Types can be considered as tuples with extra description,so that pairs,etc. referring to different things can't be accidentally equal, improving readability and helping type safety. In general it is best to use algebraic data types as opposed to tuples unless the situation is very trivial.

3.1 Comparison to C

Algebraic data types correspond to structs in C, except that the fields in Haskell are anonymous and positional. They can also correspond to enums (if the Haskell type values have no paramters), but we cannot use haskell type values in place of integers like values of enums in C.

As algebraic data types have muliple cases, they are like unions in C. However unions don't tell us which alternative is actually present.

4 Pattern Matching

We can use *pattern matching* to interpret values of algebraic data types and extract data from their parameters. A simple example:

$$myNot\ True = False$$
$$myNot\ False = True$$

When we apply myNot, Haskell checks the value we supply against each line in turn, until it finds a match. If a match is not match found at all, we would have got a "non exhaustive patterns" exception.

We can add the "-fwarn-incomplete-patterns" compilation option to ghc to warn us of this at the compilation stage.

Pattern matching is sometimes called destruction (as opposed to type constructors), but nothing is destroyed - we are just looking inside the type.

We don't have to use exact values, we can use place holders too/instead:

$$complicated :: (Num\ a, Eq\ a) \Rightarrow (Bool, t1, [t], a) \rightarrow (t1, [t])$$
$$complicated\ (True, a, x : xs, 5) = (a, xs)$$

x:xs (or any letter replacing x) is the general convention for writing pattern matching functions on lists.

We can use _ as a wildcard character, which means match any value of the given type here. It can be used when we are not interested in that value in the body of the function, or want to except any value of that type:

$$nicerID\ (Book\ id\ \_\ \_) = id$$
$$nicerTitle\ (Book\ \_\ title\ \_) = title$$
$$nicerAuthors\ (Book\ \_\ \_\ authors) = authors$$

5 Record Syntax

Instead of writing repetitive accessor functions as above, we could have just used record syntax:

data *Customer* = *Customer* {
  customerID :: *CustomerID*,
  customerName :: *String*,
  customerAddress :: *Address*
} **deriving** (*Show*)

Now we can just use the lower case names we have given to our constructors to get their values. The lower case names are functions from Customer to their type (eg. $customerId :: Customer \rightarrow CustomerID$).

We can use the record syntax to create a value of this type (in addition to the syntax we have already seen):

```
customer2 = Customer {
  customerID = 271828
  , customerName = "Jane Q. Citizen"
  , customerAddress = ["1048576 Disk Drive",
    "Milpitas, CA 95134",
    "USA"]
  }
```

Note we don't have to preserve the order of constructors in this notation. Also the type's value will be printed with the constructor names:

Customer customerID = 271828, customerName = "Jane Q. Citizen", customerAddress = ["1048576 Disk Drive","Milpitas, CA 95134","USA"]

Haskell's System.Time module uses record syntax in the definiition of the CalendarTime datatype. It has 12 paramaters and extracting one would be harder without the record syntax.

6 Parameterised Types

We can add parameters to any type we define. For example, the Maybe type is defined as:

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

where a is a parameter that can represent any type. Therefore the Maybe type is polymorphic. Any type we make with it is distinct, so Maybe Bool is distince from Maybe [[Int]].

Parameterised types are like templates in C++ and generics in Java, but unlike these languages, parameterised types have always been part of Haskell and not added in later versions.

We can nest uses of parameterised types inside each other, provided we use brackets:

```
wrapped = Just (Just "wrapped")
```

7 Reporting errors

The standard error reporting function in Haskell is error :: String -¿ a. The string parameter is the error message. The type parameter is so we can call

it anywhere and it will always be of the right type, as we always need error messages! It immediately aborts execution and throws an exception, so its type doesn't need to be anything we can use, so its vagueness is ok. The only down side is we can't use it to denote recoverable errors - this is where we would use the Maybe type, eg.:

$$
\begin{aligned}
&\mathit{safeSecond} :: [a] \to \mathit{Maybe}\ a \\
&\mathit{safeSecond}\ [\,] = \mathit{Nothing} \\
&\mathit{safeSecond}\ \mathit{xs} \\
&\quad \mid \mathit{null}\ (\mathit{tail}\ \mathit{xs}) = \mathit{Nothing} \\
&\quad \mid \mathit{otherwise} = \mathit{Just}\ (\mathit{head}\ (\mathit{tail}\ \mathit{xs}))
\end{aligned}
$$

If the list is too short to have a second position, we return Nothing, instead of forcing a crash and also improving readability.

8 Introducing local variables