Intro to Haskell Notes: Part 10

Adrian Brasoveanu*

October 13, 2013

Contents

1 Making our own types: Intro to algebraic data types (ADTs)

So far, we've run into a lot of types (technically, data types): *Bool*, *Int*, *Char*, *Maybe* etc. But how do we make our own?

One way is to use the **data** keyword to define a type. For example, here's how the *Bool* type is defined in the standard library.

(1) **data** *Bool* = *False* | *True*

- **data** means that we're defining a new data type
- the part before the = denotes the type, which is *Bool*
- the parts after the = are value constructors: they specify the different values that this type can have
- the | is read as 'or'
- so we can read this as: the *Bool* type has *True* and *False* as values

*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, http://learnyouahaskell.com/.

- both the type name *Bool* and the value constructors *True* and *False* have to start with a capital letter

1.1 Representing shapes

Let's think about how we would represent a shape in Haskell. One way would be to use tuples.
For example, a circle could be denoted as $(43.1, 55.0, 10.4)$:

- the first and second fields are the (x, y)-coordinates of the circle's center;
- the third field is the radius.

Sounds OK, but the same tuple could also represent a 3D vector or many other things.
A better solution would be to design our own type for shapes, and to do it in such a way that it would be clear to anyone using that type that it represents shapes.
Let's say that a shape can be a circle or a rectangle. Then the definition for our type could be:

```
ghci 1> data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Note that the *Circle* value constructor has three fields, which take floats.
In general:

- when we write a value constructor, we can optionally add some types after it and those types define the values it will contain.

For *Circle*, the first two fields are the coordinates of its center, the third one is its radius.
The *Rectangle* value constructor has four fields which accept floats. The first two are the coordinates of its upper left corner and the second two are coordinates of its lower right one.
By fields, we actually mean parameters:

- value constructors are actually functions that ultimately return a value of the data type they are constructors of.

Let's take a look at the type signatures for these two value constructors.

```
ghci 2> :t Circle
Circle :: Float → Float → Float → Shape
```

```
ghci 3> :t Rectangle
Rectangle :: Float → Float → Float → Float → Shape
```

So value constructors are functions like everything else.

1.2 A *surface* function for shapes

Let's make a function that takes a shape and returns its surface.

```
ghci 4> let {surface :: Shape → Float;
            surface (Circle _ _ r) = pi * r ↑ 2;
            surface (Rectangle x1 y1 x2 y2) = (abs $ x2 − x1) * (abs $ y2 − y1) }
```

First, note the type declaration:

- it says that the function takes a shape and returns a float;
- we couldn't write a more specific type declaration, e.g., *Circle* → *Float*, because *Circle* is not a type, *Shape* is;
- this is as expected: we can't write a function with a type declaration of *True* → *True*: *True* is a value (constructor), not a type; our function would have to be of type *Bool* → *Bool*.

Second, note that:

- we can pattern match against value constructors.

We pattern matched against constructors before – all the time, actually. We pattern matched against value constructors when we pattern matched against values like [], *False* or 5. The only difference is that those value constructors don't have any fields.

To pattern match against a constructor, we just write the constructor and then bind its fields to names.

For example, when we pattern match against the constructor *Circle* to get the area of the circle, we're only interested in the radius. Since we don't actually care about the first two fields (they just tell us where the circle is), we use the anonymous 'variable' _ to name them.

```
ghci 5> surface $ Circle 10 20 10
    314.15927
```

```
ghci 6> surface $ Rectangle 0 0 100 100
    10000.0
```

1.3 Deriving typeclasses

If we try to just print out *Circle* 10 20 5 at the prompt, we'll get an error.

```
ghci 7> Circle 10 20 5
```

That's because Haskell doesn't know how to display our data type as a string (yet). Recall that when we try to print out a value at the prompt, Haskell first runs the *show* function to get the string representation of our value and then it prints that out to the terminal.

To make our *Shape* type part of the *Show* typeclass, we modify it like this:

```
ghci 8> data Shape = Circle Float Float Float | Rectangle Float Float Float Float
          deriving (Show)
```

We will examine **deriving** in more detail in the next set of lecture notes. For now, we only need to know that if we add **deriving** (*Show*) at the end of a data declaration, Haskell makes that type part of the *Show* typeclass automatically.

So now, we can do this:

```
ghci 9> let {surface :: Shape → Float;
             surface (Circle _ _ r) = pi * r ↑ 2;
             surface (Rectangle x1 y1 x2 y2) = (abs $ x2 − x1) * (abs $ y2 − y1)}
```

```
ghci 10> Circle 10 20 5
         Circle 10.0 20.0 5.0
```

```
ghci 11> Rectangle 50 230 60 90
         Rectangle 50.0 230.0 60.0 90.0
```

1.4 Mapping over value constructors

Value constructors are functions, so we can partially apply them, map them over lists etc.

For example, if we want a list of concentric circles, we can do this:

```
ghci 12> map (Circle 10 20) [4, 5, 6, 7]
         [Circle 10.0 20.0 4.0, Circle 10.0 20.0 5.0, Circle 10.0 20.0 6.0, Circle 10.0 20.0 7.0]
```

1.5 Layering data types

Our *Shape* data type is good, but it could be better: we don't know at a glance the role of all those floats that the value constructors take as arguments.

Let's make an intermediate data type that defines a point in two-dimensional space. Then we can use that to make our shapes more understandable.

```
ghci 13> data Point = Point Float Float deriving (Show)
```

When we defined a point in 2D space, we used the same name for both the data type and the value constructor. This has no special meaning, it's just a common Haskell idiom (confusing for beginners, but very useful once you get used to it):

- it's common to use the same name for the data type and for the value constructor if there's only one value constructor.

```
ghci 14> data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

So now the *Circle* has two fields, one of type *Point* and the other of type *Float*. This makes it easier to understand what's what. The same goes for *Rectangle*.

We have to adjust our *surface* function to reflect these changes:

```
ghci 15> let {surface :: Shape → Float;
              surface (Circle _ r) = pi * r ↑ 2;
              surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 − x1) * (abs $ y2 − y1)}
```

The only thing we had to change were the patterns.

- note how we disregarded the whole point in the *Circle* pattern;
- also note how in the Rectangle pattern, we just used a nested pattern matching to get the fields of the points.

```
ghci 16> surface (Rectangle (Point 0 0) (Point 100 100))
    10000.0
```

```
ghci 17> surface (Circle (Point 0 0) 24)
    1809.5574
```

1.6 Creating entities of new data types

How would we go about defining a function that nudges a shape?

The function needs to take a shape, the amount to move it on the x axis and the amount to move it on the y axis. It then returns a new shape that has the same dimensions, only it's located somewhere else.

```
ghci 18> let {nudge :: Shape → Float → Float → Shape;
              nudge (Circle (Point x y) r) a b = Circle (Point (x + a) (y + b)) r;
              nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b =
                Rectangle (Point (x1 + a) (y1 + b)) (Point (x2 + a) (y2 + b))}
```

That is, we add the nudge amounts to the points that denote the position of the shape.

```
ghci 19> nudge (Circle (Point 34 34) 10) 5 10
    Circle (Point 39.0 44.0) 10.0
```

If we don't want to deal directly with points, we can make some auxilliary functions that create shapes of some size at the 0 coordinates and then nudge those.

```
ghci 20> let {baseCircle :: Float → Shape;
              baseCircle r = Circle (Point 0 0) r;
              baseRect :: Float → Float → Shape;
              baseRect width height = Rectangle (Point 0 0) (Point width height)}
```

```
ghci 21> nudge (baseRect 40 100) 60 23
    Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

1.7 Exporting data types

We can export data types in our modules:

- we just write the type along with the functions we are exporting, then specify in parentheses the value constructors that we want to export for that type (separated by commas);
- if we want to export all the value constructors for a given type, we just write (..).

For example, if we put the functions and types that we defined above in a module and want to export them, we can start the module like this:

```
module Shapes
(Point (..)
,Shape (..)
,surface
,nudge
,baseCircle
,baseRect
) where
```

Shape (..) exports all the value constructors for *Shape*, so that means that whoever imports our module can make shapes by using the *Rectangle* and *Circle* value constructors. It's the same as writing *Shape* (*Rectangle*, *Circle*).

We could also opt not to export any value constructors for *Shape* by just writing *Shape* in the export statement. That way, someone importing our module could only make shapes by using the auxilliary functions *baseCircle* and *baseRect*.

Data.Map uses that approach: we can't create a map by doing *Map.Map* $[(1,2),(3,4)]$ because that value constructor is not exported. However, we can make a mapping (dictionary) by using one of the auxilliary functions, e.g., *Map.fromList*.

Not exporting the value constructors of data types makes these types more abstract: we hide their implementation. But the users of these types can't pattern match against the value constructors if their not exported, so suitable auxiliary functions should be defined.

2 Record syntax

Suppose we've been tasked with creating a data type that describes a person. The info that we want to store about that person is:

- first name
- last name
- age
- height
- phone number
- favorite ice-cream flavor

We just define a data type with only one value constructor that takes 6 fields as arguments.

- note that we follow the usual Haskell idiom and use the same name for the type and the constructor

```
ghci 22> data Person = Person String String Int Float String String deriving (Show)
```

The value constructor *Person* is a higher-order function of the expected type:

```
ghci 23> :t Person
Person :: String → String → Int → Float → String → String → Person
```

The first field is the first name, the second is the last name, the third is the age and so on.
Let's make a person:

```
ghci 24> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

```
ghci 25> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

What if we want to create a function to get separate info from a person, i.e., a function that gets the first name, a function that gets the last name, etc.

```
ghci 26> let {firstName :: Person → String;
              firstName (Person firstname _ _ _ _ _) = firstname;
              lastName :: Person → String;
              lastName (Person _ lastname _ _ _ _) = lastname;
              age :: Person → Int;
              age (Person _ _ age _ _ _) = age;
              height :: Person → Float;
              height (Person _ _ _ height _ _) = height;
              phoneNumber :: Person → String;
              phoneNumber (Person _ _ _ _ number _) = number;
              flavor :: Person → String;
              flavor (Person _ _ _ _ _ flavor) = flavor}
```

This works, but it's very cumbersome.

```
ghci 27> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

```
ghci 28> firstName guy
"Buddy"
```

```
ghci 29> height guy
    184.2
```

```
ghci 30> flavor guy
    "Chocolate"
```

There is a better way to write data types with many fields like this one that enables us to both keep track of the fields and extract them: record syntax.

```
ghci 31> data Person' = Person' {firstName :: String, lastName :: String, age :: Int,
                                 height :: Float, phoneNumber :: String, flavor :: String}
           deriving (Show)
```

So instead of just naming the field types one after another and separating them with spaces, we use curly brackets:

- we first write the name of the field, for instance, *firstName*;
- then we write a double colon :: and specify the type.

The resulting data type and value constructor are exactly the same. For example, here's the type of the value constructor:

```
ghci 32> :t Person'
    Person' :: String → String → Int → Float → String → String → Person'
```

One of the main benefits is that record syntax creates functions that look up fields in the data type. In our case, Haskell automatically made these 6 functions: *firstName*, *lastName*, *age*, *height*, *phoneNumber* and *flavor*.

```
ghci 33> :t flavor
    flavor :: Person' → String
```

```
ghci 34> :t firstName
    firstName :: Person' → String
```

2.1 Deriving *Show* with record syntax

There's another benefit of using record syntax: when we derive *Show* for the type, entities of that type are displayed differently – their fields are explicitly labeled.

For example, suppose we need a type that represents a car. We want to keep track of the company that made it, the model name and its year of production.

```
ghci 35> data Car = Car String String Int deriving (Show)
```

```
ghci 36> Car "Ford" "Mustang" 1967
    Car "Ford" "Mustang" 1967
```

If we define it using record syntax ...

```
ghci 37> data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

... we can make a new car like this:

```
ghci 38> Car {company = "Ford", model = "Mustang", year = 1967}
    Car {company = "Ford", model = "Mustang", year = 1967}
```

Note how it's displayed.

When making a new car, we don't have to necessarily put the fields in the proper order, as long as we list all of them. But if we don't use record syntax, we have to specify them in order.

```
ghci 39> Car {company = "Ford", year = 1967, model = "Mustang"}
    Car {company = "Ford", model = "Mustang", year = 1967}
```

2.2 Default optional arguments using record syntax

Finally, we can update records and have default optional arguments if we define a default entity of the relevant type:

```
ghci 40> let defaultCar = Car {company = "", year = 0, model = ""}
```

```
ghci 41> defaultCar
    Car {company = "", model = "", year = 0}
```

```
ghci 42> let car1 = defaultCar {company = "BMW"}
```

```
ghci 43> car1
    Car {company = "BMW", model = "", year = 0}
```

```
ghci 44> let car2 = car1 {year = 2013}
```

```
ghci 45> car2
    Car {company = "BMW", model = "", year = 2013}
```

```
ghci 46> let car3 = car2 {model = "X1"}
```

```
ghci 47> car3
    Car {company = "BMW", model = "X1", year = 2013}
```

3 Type constructors, a.k.a. parametrized types

Value constructors can take some values (actual entities of specific types) as arguments and then produce a new value.

For instance, our *Car* constructor takes three values and produces a car value.

```
ghci 48> data Car = Car String String Int deriving (Show)
```

```
ghci 49> Car "Ford" "Mustang" 1967
    Car "Ford" "Mustang" 1967
```

```
ghci 50> :t Car
    Car :: String → String → Int → Car
```

In a similar manner, type constructors can take types as parameters to produce new types.

3.1 The *Maybe* type constructor

To understand type constructors better, let's take a look at how a type we've already encountered is implemented.

(2) **data** *Maybe a* = *Nothing* | *Just a*

ghci 51> :! *hoogle* -- info Maybe

Prelude data Maybe a

The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing). Using Maybe is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

The Maybe type is also a monad. It is a simple kind of error monad, error monad can be built using the Data.Either.Either type.

From package base data Maybe a

The *a* here is a type variable that is the argument of *Maybe*. And because there's a type parameter involved, we call *Maybe a* a type constructor, or a parametrized type.

Depending on what we want this data type to hold when it's not *Nothing*, this type constructor can end up producing a variety of types: a *Maybe Int* type, a *Maybe Car* type, a *Maybe String* type etc.

No value can be of type *Maybe* because that's not a type *per se*, it's a type *constructor*: in order to get a real type that an actual value can have, the type constructor has to have all its type parameters saturated.

So if we pass *Char* as the type parameter to *Maybe*, we get a type of *Maybe Char*. For example, the value *Just* `'f'` is of type *Maybe Char*:

ghci 52> :*t Just* `'f'`

Just `'f'` :: *Maybe Char*

When we do :*t Just* `'f'`, the type inference engine figures it out to be of type *Maybe Char*: since the `'f'` in *Just* `'f'` is a character, the *a* in *Maybe a* must be *Char*.

Type constructors are useful because:

- they yield different types depending on the types they take as parameters,
- but at the same time, they encapsulate generic ways of manipulating values of all these different types.

3.2 The list type constructor

We actually used a type constructor even before we introduced *Maybe*: the list type constructor.

The list type constructor takes a type parameter like *Int* or *Char* and produces a concrete type: the type of *Int* lists, namely [*Int*], or the type of *Char* lists, namely [*Char*] etc.

But we can't have a value that is of type [] – because [] is a type *constructor*, not a concrete type (note that [] symbolizes a *type constructor* here and not the empty list [], which is a polymorphic *value*).

ghci 53> :*t* `"hello"`

`"hello"` :: [*Char*]

ghci 54> :*t* [`"hello"`]

[`"hello"`] :: [[*Char*]]

```
ghci 55> :t [["hello"]]
[["hello"]] :: [[[Char]]]
```

```
ghci 56> :t [1..6]
[1..6] :: (Enum t, Num t) ⇒ [t]
```

```
ghci 57> :t [[1..6]]
[[1..6]] :: (Enum t, Num t) ⇒ [[t]]
```

```
ghci 58> :t [[[1..6]]]
[[[1..6]]] :: (Enum t, Num t) ⇒ [[[t]]]
```

A nice consequence of the fact that lists are a parametrized type is that an empty list can act like a list of anything, i.e., it is a polymorphic value.

That's why we can do both of these:

```
ghci 59> [1,2,3] ++ []
[1,2,3]
```

```
ghci 60> ["ha","ha","ha"] ++ []
["ha","ha","ha"]
```

This is very similar to the way *Nothing* behaves: its type is *Maybe a*, i.e., it is polymorphic just like the empty list.

```
ghci 61> :t Nothing
Nothing :: Maybe a
```

```
ghci 62> :t []
[] :: [a]
```

If a function requires an argument of type *Maybe Int* and another function requires an argument of type *Maybe Char*, we can pass *Nothing* to both of them because a *Nothing* doesn't contain a specific type of value. This is parallel to the way we can concatenate the empty list and lists of any type.

3.3 Creating a parametrized type: 3D vectors

Let's implement a 3D vector type and add some operations for it. We'll be using a parameterized type because it will support several numeric types.

```
ghci 63> data Vector a = Vector a a a deriving (Show)
```

```
ghci 64> let {vplus :: (Num a) => Vector a -> Vector a -> Vector a;
              (Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)}
```

```
ghci 65> (Vector 1 2 3) `vplus` (Vector 4 5 6)
    Vector 5 7 9
```

```
ghci 66> let {vectMult :: (Num a) => Vector a -> a -> Vector a;
              (Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)}
```

```
ghci 67> (Vector 1 2 3) `vectMult` 4
    Vector 4 8 12
```

```
ghci 68> let {scalarMult :: (Num a) => Vector a -> Vector a -> a;
              (Vector i j k) `scalarMult` (Vector l m n) = i*l+j*m+k*n}
```

```
ghci 69> (Vector 1 2 3) `scalarMult` (Vector 4 4 4)
    24
```

The three functions defined above can operate on various types: *Vector Int*, *Vector Integer*, *Vector Float*, *Vector Double* etc. The only restriction on the type variable *a* in *Vector a* is that *a* needs to be a member of the *Num* typeclass.

But this type freedom is properly constrained: if we examine the type declaration for these functions, we see that they can operate only on vectors of the same type and the numbers involved must also be of the type that is contained in the vectors.

Incidentally, note that we didn't put a *Num* class constraint in the data declaration, because we'd have to repeat it in the functions anyway.

3.4 Summary: Type constructors vs. value constructors

It's very important to distinguish between type constructors, which are functions that take types (or type variables) as arguments, and value constructors, which are functions that take actual values as arguments.

In particular, when we declare a data type, the part before the = is the type constructor, while the constructors after = (possibly separated by |'s) are value constructors.

This is very clear for some of the data types we discussed.

For example, in the definition of the Boolean type **data** *Bool* = *False* | *True*:

- *Bool* is a 0-arity type constructor, i.e., simply a type;
- *False* and *True* are 0-arity value constructors, i.e., simply values.

Similarly, in the **data** *Shape* = *Circle Float Float Float* | *Rectangle Float Float Float Float* definition:

- *Shape* is a 0-arity type constructor, i.e., simply a type;
- *Circle* is a value constructor with an arity of 3 (it takes 3 floating-point numbers and returns a shape value);
- *Rectangle Float Float Float Float* is a value constructor with an arity of 4.

In the definition of the *Maybe* type **data** *Maybe a* = *Nothing* | *Just a*:

- *Maybe* is a type constructor with an arity of 1, i.e., it takes one type as its argument / parameter and returns another type; the type parameter is represented with a type variable *a*;
- *Nothing* is a 0-arity value constructor, i.e., simply a value;
- *Just* is a value constructor of arity 1 that takes a value of type *a* as its argument and returns a value of type *Maybe a*; note that the type variable *a* is not an argument in *Just a*: *Just* takes actual values of type *a* as arguments, not types; *a* simply indicates what kind of values *Just* takes.

Finally, we defined our vector type above as **data** *Vector a* = *Vector a a a*. *Vector* is overloaded here (following the appropriate Haskell idiom):

- the occurrence of *Vector* before the = sign is a type constructor with an arity of 1: it takes a type *a* as its sole argument and returns another type as its value (the type *Vector a*);
- the occurrence of *Vector* after the = sign is a value constructor: it will take three numbers of type *a* (integers, floats, doubles etc.) and it will return an actual 3D vector (a *bona fide* value) built out of those three numbers.

Saying that the *scalarMult* function above is of type *Vector a a a* $\rightarrow$ *Vector a a a* $\rightarrow$ *a* is incorrect because we have to put types in the type declaration and the *Vector* type constructor takes only one parameter.

Similarly, passing *Vector* 1 as an argument to the *vectMult* function is incorrect because this function takes an actual vector as one of its arguments and the *Vector* value constructor needs three numbers to build a vector.

Why overload *Vector* like this and use it both as a type constructor and as a value constructor? (Even more confusingly, the two constructors have different arities.)

We don't have to do it, but this is a Haskell idiom that is (probably) universally used: if you have a data type that has only one value constructor, you should use the same label for the type constructor and the value constructor.

This is confusing when you encounter data types for the first time, but once you learn how to read these data type declarations, it'll make much more sense:

- if there is only one value constructor for your custom data type, it doesn't make sense to use two names instead of one because you'll be forced to remember both of them later on when you write functions over values of that data type;
- in addition, overloading a name like that is never ambiguous in context: it is always clear if we are talking about a type or a value because we will either be in the 'type' part of Haskell (immediately after a ::) or in the 'value' part (e.g., in the body of a function definition).