Algebraic data types

CIS 194 Week 3 11 September 2014

Suggested reading:

- **Real World Haskell**, chapters 2 - 4.

Libraries

Haskell comes equipped with a decent standard library that can be accessed from any Haskell program. Haskell libraries are distributed in *packages*, each of which can contain any number of *modules*. A vanilla Haskell installation comes with the `base` package, among a few others. The `base` package contains the `Prelude` module, which contains definitions that are automatically available in any Haskell program. The `base` package also contains many other useful modules, which can be imported with a statement like this:

```
import Data.Char ( toUpper )
```

That line imports the `Data.Char` module, but only grabs the definition for `toUpper`, as we'll use below. The parenthesized bit is optional; if it is left out, all definitions from the imported module are brought in. You can read documentation for the `base` package on [Hackage][] – search for `base`. Of particular use toward the beginning of the course is the `Data.List` module.

Haskell also provides a facility to (somewhat) easily download and install new packages to use. [Hackage][] is the main distribution server for these packages, and `cabal` is a program, installed with the Haskell Platform, that pulls packages down from this server and installs them. Here is how you do it at a command prompt:

```
~> cabal update
~> cabal install text-1.1.1.3
```

The first line instructs `cabal` to download an updated list of available packages; the second line installs version 1.1.1.3 of the `text` package. You can leave the version number out; that will install the most recent version. The `text` package provides a way to store chunks of text (strings, essentially) that is considerably more efficient than `String`. These instructions tell you to use version 1.1.1.3 because the most recent version, 1.2.0.0, is not compatible with some other packages you might want to install later on.

Enumeration types

Like many programming languages, Haskell allows programmers to create their own *enumeration* types. Here's a simple example:

```
data Thing = Shoe
           | Ship
           | SealingWax
           | Cabbage
           | King
  deriving Show
```

This declares a new type called `Thing` with five *data constructors* `Shoe`, `Ship`, etc. which are the (only) values of type `Thing`. (The `deriving Show` is a magical incantation which tells GHC to automatically generate default code for converting `Things` to `Strings`. This is what `ghci` uses when printing the value of an expression of type `Thing`.)

```
shoe :: Thing
shoe = Shoe

listO'Things :: [Thing]
listO'Things = [Shoe, SealingWax, King, Cabbage, King]
```

We can write functions on `Things` by *pattern-matching*.

```
isSmall :: Thing -> Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

Recalling how function clauses are tried in order from top to bottom, we could also make the definition of `isSmall` a bit shorter like so:

```
isSmall2 :: Thing -> Bool
isSmall2 Ship = False
isSmall2 King = False
isSmall2 _    = True
```

Beyond enumerations

`Thing` is an *enumeration type*, similar to those provided by other languages such as Java or C++. However, enumerations are actually only a special case of Haskell's more general *algebraic data types*. As a first example of a data type which is not just an enumeration, consider the definition of `FailableDouble`:

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

This says that the `FailableDouble` type has two data constructors. The first one, `Failure`, takes no arguments, so `Failure` by itself is a value of type `FailableDouble`. The second one, `OK`, takes an argument of type `Double`. So `OK` by itself is not a value of type `FailableDouble`; we need to give it a `Double`. For example, `OK 3.4` is a value of type `FailableDouble`.

```
ex01 = Failure
ex02 = OK 3.4
```

Thought exercise: what is the type of `OK`?

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

More pattern-matching! Notice how in the `OK` case we can give a name to the `Double` that comes along with it.

```
failureToZero :: FailableDouble -> Double
```

```
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Data constructors can have more than one argument.

```
-- Store a person's name, age, and favorite Thing.
data Person = Person String Int Thing
  deriving Show

richard :: Person
richard = Person "Richard" 32 Ship

stan :: Person
stan  = Person "Stan" 94 Cabbage

getAge :: Person -> Int
getAge (Person _ a _) = a
```

Notice how the type constructor and data constructor are both named `Person`, but they inhabit different namespaces and are different things. This idiom (giving the type and data constructor of a one-constructor type the same name) is common, but can be confusing until you get used to it.

Algebraic data types in general

In general, an algebraic data type has one or more data constructors, and each data constructor can have zero or more arguments.

```
data AlgDataType = Constr1 Type11 Type12
                 | Constr2 Type21
                 | Constr3 Type31 Type32 Type33
                 | Constr4
```

This specifies that a value of type `AlgDataType` can be constructed in one of four ways: using `Constr1`, `Constr2`, `Constr3`, or `Constr4`. Depending on the constructor used, an `AlgDataType` value may contain some other values. For example, if it was constructed using `Constr1`, then it comes along with two values, one of type `Type11` and one of type `Type12`.

One final note: type and data constructor names must always start with a capital letter; variables (including names of functions) must always start with a lowercase letter. (Otherwise, Haskell parsers would have quite a difficult job figuring out which names represent variables and which represent constructors).

Pattern-matching

We've seen pattern-matching in a few specific cases, but let's see how pattern-matching works in general. Fundamentally, pattern-matching is about taking apart a value by *finding out which constructor* it was built with. This information can be used as the basis for deciding what to do—indeed, in Haskell, this is the *only* way to make a decision.

For example, to decide what to do with a value of type `AlgDataType` (the made-up type defined in the previous section), we could write something like

```
foo (Constr1 a b)   = ...
foo (Constr2 a)     = ...
foo (Constr3 a b c) = ...
foo Constr4         = ...
```

Note how we also get to give names to the values that come along with each constructor. Note also that parentheses are required around patterns consisting of more than just a single constructor.

This is the main idea behind patterns, but there are a few more things to note.

1. An underscore `_` can be used as a "wildcard pattern" which matches anything.

2. A pattern of the form `x@pat` can be used to match a value against the pattern `pat`, but *also* give the name `x` to the entire value being matched. For example:

   ```
   baz :: Person -> String
   baz p@(Person n _ _) = "The name field of (" ++ show p ++ ") is " ++ n

   *Main> baz richard
   "The name field of (Person \"Richard\" 32 Ship) is Richard"
   ```

3. Patterns can be *nested*. For example:

   ```
   checkFav :: Person -> String
   checkFav (Person n _ SealingWax) = n ++ ", you're my kind of person!"
   checkFav (Person n _ _)          = n ++ ", your favorite thing is lame."

   *Main> checkFav richard
   "Richard, you're my kind of person!"
   *Main> checkFav stan
   "Stan, your favorite thing is lame."
   ```

   Note how we nest the pattern `SealingWax` inside the pattern for `Person`.

In general, the following grammar defines what can be used as a pattern:

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

The first line says that an underscore is a pattern. The second line says that a variable by itself is a pattern: such a pattern matches anything, and "binds" the given variable name to the matched value. The third line specifies @-patterns. The last line says that a constructor name followed by a sequence of patterns is itself a pattern: such a pattern matches a value if that value was constructed using the given constructor, *and* `pat1` through `patn` all match the values contained by the constructor, recursively.

(In actual fact, the full grammar of patterns includes yet more features still, but the rest would take us too far afield for now.)

Note that literal values like 2 or `'c'` can be thought of as constructors with no arguments. It is as if the types `Int` and `Char` were defined like

```
data Int  = 0 | 1 | -1 | 2 | -2 | ...
data Char = 'a' | 'b' | 'c' | ...
```

which means that we can pattern-match against literal values. (Of course, `Int` and `Char` are not *actually* defined this way.)

Case expressions

The fundamental construct for doing pattern-matching in Haskell is the `case` expression. In general, a `case`

expression looks like

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

When evaluated, the expression `exp` is matched against each of the patterns `pat1`, `pat2`, … in turn. The first matching pattern is chosen, and the entire `case` expression evaluates to the expression corresponding to the matching pattern. For example,

```
ex03 = case "Hello" of
           []      -> 3
           ('H':s) -> length s
           _       -> 7
```

evaluates to 4 (the second pattern is chosen; the third pattern matches too, of course, but it is never reached).

In fact, the syntax for defining functions we have seen is really just convenient syntax sugar for defining a `case` expression. For example, the definition of `failureToZero` given previously can equivalently be written as

```
failureToZero' :: FailableDouble -> Double
failureToZero' x = case x of
                     Failure -> 0
                     OK d    -> d
```

First-class functions

Haskell is a *functional* programming language. This means that a function – an operation that can be run – is considered a chunk of data just like any other chunk of data. Functions can be stored in variables and passed to other functions just like any other data. Just like the type of a character is written as `Char`, the type of a function is written with an arrow, `->`. The difference is that a function type must also specify the domain of the function (what type of argument it takes) and the codomain of the function (what type of value it produces).

(I say “codomain” instead of “range”, because “range” means that set of values that the function actually produces, as opposed to a codomain, which is the set of values a function could potentially produce. A function’s range is a subset of its codomain.)

One of the most important functions in a functional programming language is `map`. But, the full definition of `map` requires talking about a Haskell feature called “parametric polymorphism”, so we’ll write a version of `map` that just works on `Integers`.

```
mapInteger :: (Integer -> Integer) -> [Integer] -> [Integer]
mapInteger _ []     = []
mapInteger f (x:xs) = f x : mapInteger f xs
```

`mapInteger` takes a *function* on `Integers` (that is, a function that takes an `Integer` and returns an `Integer`) and runs that function on a list of `Integers`. Say we write this function:

```
plus1 :: Integer -> Integer
plus1 x = x + 1
```

Notice the type of `plus1` – it matches up exactly with the type of the first argument to `mapInteger`: `plus1` is a function on `Integers`. So, we can say this:

```
twoThreeFour :: [Integer]
twoThreeFour = mapInteger plus1 [1,2,3]
```

Being able to take an operation on elements and convert that to an operation on a list, without really doing any work, is a wonderful feature of functional programming languages. The Prelude's `map` function does this with any types:

```
twoThreeFour' :: [Integer]
twoThreeFour' = map plus1 [1,2,3]

shout :: String -> String    -- remember a String is just a [Char]
shout x = map toUpper x      -- toUpper converts a char to upper case
```

Functions that take other functions as arguments are called higher-order. Look at this higher-order function. What does it do? Can you think of a way of calling it?

```
filterInteger :: (Integer -> Bool) -> [Integer] -> [Integer]
filterInteger _ []     = []
filterInteger f (x:xs)
  | f x       = x : filterInteger f xs
  | otherwise = filterInteger f xs
```

Recursive data types

Data types can be *recursive*, that is, defined in terms of themselves. In fact, we have already seen a recursive type—the type of lists. A list is either empty, or a single element followed by a remaining list. We could define our own list type like so:

```
data IntList = Empty | Cons Int IntList
```

Haskell's own built-in lists are quite similar; they just get to use special built-in syntax (`[]` and `:`). (Of course, they also work for any type of elements instead of just `Int`s; more on this next week.)

We often use recursive functions to process recursive data types:

```
intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

As another simple example, we can define a type of binary trees with an `Int` value stored at each internal node, and a `Char` stored at each leaf:

```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

(Don't ask me what you would use such a tree for; it's an example, OK?) For example,

```
tree :: Tree
tree = Node (Leaf 'x') 1 (Node (Leaf 'y') 2 (Leaf 'z'))
```

Generated 2014-12-04 13:37:48.642261