CS 314 Lecture 12

Haskell: algebraic data types

(adapted from Brent Yorgey's CIS 194)

March 5, 2019

Constructing lists

```haskell
emptyList = []
```

We can build larger lists using “:” (cons):

```haskell
ex18 = 1 : []
ex19 = 3 : (1 : [])
ex20 = 2 : 3 : 4 : []

ex21 = [2,3,4] == 2 : 3 : 4 : []
```

Constructing lists

```haskell
collatz :: Integer -> Integer
collatz n
  | isEven n  = n `div` 2
  | otherwise = 3*n + 1

-- Generate the sequence of collatz iterations
collatzSeq :: Integer -> [Integer]
collatzSeq 1 = [1]
collatzSeq n = n : collatzSeq (collatz n)
```

Functions on lists

```haskell
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

Note that we never use the x on the last line.

Functions on lists

```haskell
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (_:xs) = 1 + intListLength xs
```

We can replace unused variables with a placeholder "_".

Patterns

We can also use nested patterns:

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo []           = []
sumEveryTwo (x:[])       = [x]
sumEveryTwo (x:(y:zs))   = (x + y) : sumEveryTwo zs
```

The last line could also be written:

```
sumEveryTwo (x:y:zs) = (x + y) : sumEveryTwo zs
```

Combining functions

```haskell
-- The number of collatz steps needed to reach 1
collatzLen :: Integer -> Integer
collatzLen n = intListLength (collatzSeq n) - 1
```

Error messages

Haskell error messages can be a bit scary at first, but they're actually quite useful.

```
> 'x' ++ "foo"

<interactive>:1:1:
    Couldn't match expected type ‘[Char]’ with
                           actual type ‘Char’
    In the first argument of ‘(++)’, namely 'x'
    In the expression: 'x' ++ "foo"
    In an equation for ‘it’: it = 'x' ++ "foo"
```

Exercise: credit card numbers

To validate a credit card number:

- double every other number, starting from the right
- sum all the digits
- check if the result ends in 0

Exercise: credit card numbers

To validate a credit card number (e.g., “8573”):

- double every other number, starting from the right (“16 5 14 3”)
- sum all the digits (“$1 + 6 + 5 + 1 + 4 + 3 = 20$”)
- check if the result ends in 0 (20 ends in 0, so yes)

Exercise: credit card numbers

Let's write a few functions:

- toDigits :: Integer -> [Integer]
- toDigitsRev :: Integer -> [Integer]
- doubleEveryOther :: [Integer] -> [Integer]
- sumDigits :: [Integer] -> Integer
- validate :: Integer -> Bool

Exercise: credit card numbers

Let's write a few functions:

- toDigits 8573 = [8, 5, 7, 3]
- toDigitsRev 8573 = [3, 7, 5, 8]
- doubleEveryOther [8, 5, 7, 3] = [16, 5, 14, 3]
- sumDigits [16, 5, 14, 3] = 1 + 6 + 5 + 1 + 4 + 3 = 20
- validate 8573 = True

Enumeration types

We can create our own enumeration types:

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

Enumeration types

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

- Creates new type called Thing
- Creates five data constructors (Shoe, Ship, ...)
- These are the only values of type Thing
- "deriving Show" automatically generates code to convert a Thing to a String

Enumeration types

```
shoe :: Thing
shoe = Shoe

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

Enumeration types

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
```

Enumeration types

Since function clauses are tried in order from top to bottom, we could make this a bit shorter:

```
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. However, enumerations are only a special case of Haskell's more general algebraic data types.

Consider a data type that is not just an enumeration:

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

Beyond enumeration

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

- Creates type FailableDouble
- Creates two data constructors, Failure and OK
- Failure takes no arguments, so Failure is a value of type FailableDouble
- OK takes one argument
  - OK by itself is not a value of type FailableDouble
  - OK 3.4 would be

Beyond enumeration

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

What is the type of OK?

Beyond enumeration

```haskell
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

Data constructors can have more than one argument.

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

Data constructors

```
brent :: Person
brent = Person "Brent" 31 SealingWax

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

Pattern matching

```
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, 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. 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.

Type and data constructors

- 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, the parser would have 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.

Pattern-matching

For example, to decide what to do with a value of type AlgDataType, 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.

Pattern-matching

A few more things to note:

- An underscore can be used as a "wildcard pattern" which matches anything.
- 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.
- Patterns can be nested.

Pattern-matching

An example of using x@pat:

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

Pattern matching

An example of nested patterns:

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

Pattern matching

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

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

Pattern matching

- First line: an underscore is a pattern
- Second line: a variable by itself is a pattern: such a pattern matches anything, and “binds” the given variable name to the matched value
- Third line: specifies @-patterns
- Fourth line: 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

Pattern matching

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.

Case expressions

For example,

```haskell
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).

Case expressions

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
```

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 Ints; more on this next week.)

Recursive data types

We often use recursive functions to process recursive data types:

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

Recursive data types

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
```

Recursive data types

For example,

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

would represent the following tree: