CS 314 Lecture 13

Haskell: recursion and polymorphism

(adapted from Brent Yorgey's CIS 194)

March 5, 2019

Enumeration types

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

Beyond enumerations

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

Beyond enumeration

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

More pattern-matching!

```
failureToZero :: FailableDouble -> Double
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Data constructors

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

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

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 later.)

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:

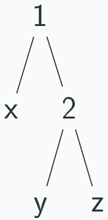
```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

Recursive data types

For example,

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

would represent the following tree:



Recursion

So far we've written many explicitly recursive functions, but in fact experienced Haskell programmers hardly ever write recursive functions!

Recursion

How is this possible?

- There are certain common patterns that come up over and over again
- Abstract out these patterns into library functions
- Frees programmers to think about problems at a higher level

Recursion patterns

Recall our simple definition of lists of Int values:

```haskell
data IntList = Empty | Cons Int IntList
  deriving Show
```

What sorts of things might we want to do with an IntList?

Recursion patterns

What sorts of things might we want to do with an IntList?

Here are a few common possibilities:

- Perform some operation on every element of the list
- Keep only some elements of the list, and throw others away, based on a test
- "Summarize" the elements of the list somehow (find their sum, product, maximum...).

Map

Let's think about the first one ("perform some operation on every element of the list").

For example, we could add one to every element in a list:

```
addOneToAll :: IntList -> IntList
addOneToAll Empty       = Empty
addOneToAll (Cons x xs) = Cons (x+1) (addOneToAll xs)
```

Map

Or we could ensure that every element in a list is nonnegative by taking the absolute value:

```
absAll :: IntList -> IntList
absAll Empty         = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)
```

Map

Or we could square every element:

```
squareAll :: IntList -> IntList
squareAll Empty       = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

Map

```
addOneToAll :: IntList -> IntList
addOneToAll Empty        = Empty
addOneToAll (Cons x xs) = Cons (x+1) (addOneToAll
    xs)

absAll :: IntList -> IntList
absAll Empty        = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)

squareAll :: IntList -> IntList
squareAll Empty        = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

Map

These three functions look way too similar. There ought to be some way to abstract out the commonality so we don't have to repeat ourselves!

There is indeed a way – which parts are the same in all three examples and which parts change?

The thing that changes, of course, is the operation we want to perform on each element of the list. We can specify this operation as a function of type Int -> Int.

Map

```haskell
mapIntList :: ? -> ? -> ?
mapIntList = undefined
```

Map

```haskell
mapIntList :: (Int -> Int) -> IntList -> IntList
mapIntList = undefined
```

Map

```
mapIntList :: (Int -> Int) -> IntList -> IntList
mapIntList _ Empty        = Empty
mapIntList f (Cons x xs) = Cons (f x) (mapIntList f xs)
```

Map

We can now use mapIntList to implement addOneToAll, absAll, and squareAll:

```
exampleList = Cons (-1) (Cons 2 (Cons (-6) Empty))

addOne x = x + 1
square x = x * x

mapIntList addOne exampleList
mapIntList abs    exampleList
mapIntList square exampleList
```

Filter

Another common pattern is when we want to keep only some elements of a list, and throw others away, based on a test.

For example, we might want to keep only the positive numbers:

```
[1, -3, 4, -8] -> [1, 4]
```

Or only the even ones:

```
[1, -3, 4, -8] -> [4, -8]
```

Filter

```
keepOnlyPositive :: IntList -> IntList
keepOnlyPositive = undefined

keepOnlyEven :: IntList -> IntList
keepOnlyEven = undefined
```

Filter

```haskell
keepOnlyPositive :: IntList -> IntList
keepOnlyPositive Empty = Empty
keepOnlyPositive (Cons x xs)
  | x > 0     = Cons x (keepOnlyPositive xs)
  | otherwise = keepOnlyPositive xs

keepOnlyEven :: IntList -> IntList
keepOnlyEven Empty = Empty
keepOnlyEven (Cons x xs)
  | even x    = Cons x (keepOnlyEven xs)
  | otherwise = keepOnlyEven xs
```

Filter

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

```
filter :: ? -> ? -> ?
filter = undefined
```

Filter

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

```
filter :: (Int -> Bool) -> IntList -> IntList
filter = undefined
```

Filter

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

```haskell
filter :: (Int -> Bool) -> IntList -> IntList
filter _ Empty = Empty
filter p (Cons x xs)
  | p x       = Cons x (filter p xs)
  | otherwise = filter p xs
```

Filter

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

```haskell
isPositive :: Int -> Bool
isPositive x = x > 0

keepOnlyEven xs = filter even xs
keepOnlyPositive xs = filter isPositive xs
```

Anonymous functions

But it's annoying to give isPositive a name, since we are probably never going to use it again. Instead, we can use an anonymous function, also known as a lambda abstraction:

```haskell
keepOnlyPositive2 :: [Integer] -> [Integer]
keepOnlyPositive2 xs = filter (\x -> x > 0) xs
```

`\x -> x > 0` is the function which takes a single argument x and outputs whether x is greater than 0.

(the backslash is supposed to look kind of like a lambda with the short leg missing)

Anonymous functions

Lambda abstractions can also have multiple arguments.

For example:

```
Prelude> (\x y z -> [x, 2*y, 3*z]) 5 6 3
[5,12,9]
```

Anonymous functions

However, in the particular case of keepOnlyPositive, there's an even better way to write it, without a lambda abstraction:

```haskell
keepOnlyPositive3 :: [Integer] -> [Integer]
keepOnlyPositive3 xs = filter (> 0) xs
```

Anonymous functions

`(>0)` is an operator section.

If ? is an operator:

- `(?y)` is equivalent to `\x -> x ? y`
- `(y?)` is equivalent to `\x -> y ? x`.

In other words, using an operator section allows us to partially apply an operator to one of its two arguments. What we get is a function of a single argument.

Sections

Some examples:

```
Prelude> (>100) 102
True
Prelude> (100>) 102
False
Prelude> map (*6) [1..5]
[6,12,18,24,30]
```

Fold

The final pattern we mentioned was to "summarize" the elements of the list; this is also variously known as a "fold" or "reduce" operation. We'll come back to this later.

Polymorphism

We've now written some nice, general functions for mapping and filtering over lists of Ints. But we're not done generalizing!

What if we wanted to filter lists of Integers? or Bools? Or lists of lists of trees of stacks of Strings?

We'd have to make a new data type and a new function for each of these cases.

Even worse, the code would be exactly the same; the only thing that would be different is the type signatures.

Can't Haskell help us out here?

Polymorphism

Of course it can!

Haskell supports polymorphism for both data types and functions.

The word “polymorphic” comes from Greek and means “having many forms”: something which is polymorphic works for multiple types.

Polymorphic data types

First, let's see how to declare a polymorphic data type.

```
data List t = E | C t (List t)
```

(We can't reuse Empty and Cons since we already used those for the constructors of IntList, so we'll use E and C instead.)

Polymorphic data types

```
data List t = E | C t (List t)
```

- Before: data IntList = ...
- Now: data List t = ...

The t is a type variable which can stand for any type (type variables must start with a lowercase letter, whereas types must start with uppercase.)

Polymorphic data types

```
data List t = E | C t (List t)
```

data List t = ... means that the List type is parameterized by a type, in much the same way that a function can be parameterized by some input.

Given a type t, a (List t) consists of either the constructor E, or the constructor C along with a value of type t and another (List t).

Polymorphic data types

Some examples:

```
lst1  :: List Int
lst1 = C 3 (C 5 (C 2 E))

lst2  :: List Char
lst2 = C 'x' (C 'y' (C 'z' E))

lst3  :: List Bool
lst3 = C True (C False E)
```

Polymorphic functions

Now, let's generalize filterIntList to work over our new polymorphic Lists.

```
filterIntList :: (Int -> Bool) -> IntList ->
    IntList
filterIntList _ Empty = Empty
filterIntList p (Cons x xs)
  | p x       = Cons x (filterIntList p xs)
  | otherwise = filterIntList p xs
```

Polymorphic functions

We can just take filterIntList and replace Empty by E and Cons by C:

```
filterList _ E = E
filterList p (C x xs)
  | p x       = C x (filterList p xs)
  | otherwise = filterList p xs
```

Now, what is the type of filterList?

Polymorphic data types

Now, what is the type of filterList?

Let's see what type ghci infers for it:

```
*Main> :t filterList
filterList :: (t -> Bool) -> List t -> List t
```

We can read this as: "for any type t, filterList takes a function from t to Bool, and a list of t's, and returns a list of t's."

Polymorphic data types

What about generalizing mapIntList? What type should we give to a function mapList that applies a function to every element in a List t?

Our first idea might be to give it the type

```
mapList :: (t -> t) -> List t -> List t
```

Polymorphic data types

This works, but it means that when applying mapList, we always get a list with the same type of elements as the list we started with. This is overly restrictive: we'd like to be able to do things like mapList show in order to convert, say, a list of Ints into a list of Strings.

Here, then, is the most general possible type for mapList, along with an implementation:

```
mapList :: (a -> b) -> List a -> List b
mapList _ E          = E
mapList f (C x xs) = C (f x) (mapList f xs)
```

Polymorphic data types

One important thing to remember about polymorphic functions is that the caller gets to pick the types.

When you write a polymorphic function, it must work for every possible input type.

Polymorphic data types

Note that Haskell has no way to directly make make decisions based on what type something is.

Types are static, and are erased by the compiler when generating machine code.

In dynamic languages, types must be maintained and checked at runtime (e.g., in “x + y”, are x and y both ints?)

Haskell guarantees this at compile time, so execution doesn’t have to check and can run faster.

Fold

We have one more recursion pattern on lists to talk about: folds. Here are a few functions on lists that follow a similar pattern: all of them somehow “combine” the elements of the list into a final answer.

```
sum' :: [Integer] -> Integer
sum' []       = 0
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer
product' [] = 1
product' (x:xs) = x * product' xs

length' :: [a] -> Int
length' []       = 0
length' (_:xs) = 1 + length' xs
```

Fold

```
fold :: b -> (a -> b -> b) -> [a] -> b
fold z f []     = z
fold z f (x:xs) = f x (fold z f xs)
```

Fold

Notice how fold essentially replaces [] with z and (:) with f, that is,

```
fold f z [a,b,c] == a `f` (b `f` (c `f` z))
```

(If you think about fold from this perspective, you may be able to figure out how to generalize fold to data types other than lists...)

Fold

Now let's rewrite sum', product', and length' in terms of fold:

```
sum''     = fold 0 (+)
product'' = fold 1 (*)
length''  = fold 0 (\_ s -> 1 + s)
```

(Instead of `(\_ s -> 1 + s)` we could also write `(\_ -> (1+))` or even `(const (1+))`.)

Fold

Of course, fold is already provided in the standard Prelude, under the name foldr. The arguments to foldr are in a slightly different order but it's the exact same function. Here are some Prelude functions which are defined in terms of foldr:

```haskell
length :: [a] -> Int
sum :: Num a => [a] -> a
product :: Num a => [a] -> a
and :: [Bool] -> Bool
or :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

Fold

There is also foldl, which folds “from the left”. That is,

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

(In general, however, you should use foldl’ from Data.List instead, which does the same thing as foldl but is more efficient.)

The Prelude

The Prelude is a module with a bunch of standard definitions that gets implicitly imported into every Haskell program. It's worth spending some time skimming through its documentation to familiarize oneself with the tools that are available.

Of course, polymorphic lists are defined in the Prelude, along with many useful polymorphic functions for working with them. For example, filter and map are the counterparts to our filterList and mapList. In fact, the Data.List module contains many more list functions still.

The Prelude

Another useful polymorphic type to know is Maybe, defined as

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

A value of type Maybe a either contains a value of type a (wrapped in the Just constructor), or it is Nothing (representing some sort of failure or error).

The Data.Maybe module has functions for working with Maybe values.

Total and partial functions

Consider this polymorphic type:

```
[a] -> a
```

What functions could have such a type? The type says that given a list of things of type a, the function must produce some value of type a. For example, the Prelude function head has this type.

Total and partial functions

...But what happens if head is given an empty list as input? Let's look at the source code for head...

```haskell
head :: [a] -> a
head (x:_) = x
head []    = errorEmptyList "head"
```

It crashes! There's nothing else it possibly could do, since it must work for all types. There's no way to make up an element of an arbitrary type out of thin air.

Total and partial functions

head is what is known as a *partial function*: there are certain inputs for which head will crash. Functions which have certain inputs that will make them recurse infinitely are also called partial. Functions which are well-defined on all possible inputs are known as *total functions*.

It is good Haskell practice to avoid partial functions as much as possible. Actually, avoiding partial functions is good practice in any programming language – but in most of them it's ridiculously annoying. Haskell tends to make it quite easy and sensible.

Total and partial functions

Using head is dangerous! You should try to avoid it whenever possible.

Other partial Prelude functions you should almost never use include tail, init, last, and (!!).

What to do instead?

Replacing partial functions

Often partial functions like head, tail, and so on can be replaced by pattern-matching:

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))

doStuff2 :: [Int] -> Int
doStuff2 []          = 0
doStuff2 [_]         = 0
doStuff2 (x1:x2:_)   = x1 + x2
```

These functions compute the same result and are both total. But only the second one is obviously total (and easier to read).

Writing partial functions

What if you find yourself writing a partial functions? There are two approaches to take. The first is to change the output type of the function to indicate the possible failure. Recall the definition of Maybe:

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

Now, suppose we were writing head. We could rewrite it safely like this:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Writing partial functions

Why is this a good idea?

- safeHead will never crash.
- The type of safeHead makes it obvious that it may fail for some inputs.
- The type system ensures that users of safeHead must appropriately check the return value of safeHead to see whether they got a value or Nothing.

Writing partial functions

In some sense, safeHead is still “partial”; but we have reflected the partiality in the type system, so it is now safe.

The goal is to have the types tell us as much as possible about the behavior of functions.

Writing partial functions

OK, but what if we know that we will only use head in situations where we are guaranteed to have a non-empty list? In such a situation, it is really annoying to get back a Maybe a, since we have to expend effort dealing with a case which we "know" cannot actually happen.

The answer is that if some condition is really guaranteed, then the types ought to reflect the guarantee! Then the compiler can enforce your guarantees for you.

Writing partial functions

For example:

```
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```