CS4450 Lecture 13 Notes

Monday, September 17, 2018 3:00 PM

Chapter 3:

Types and Type Classes

In a functional language, you use functions as first-class values

What is the type of **foo**?

```
foo st = [ c | c <- st, c `elem` ['A'..'Z']
```

A lot of these questions will be on the midterm.

st is a list of characters. How do we know that it's a list of something? Well, it's here in the "generator" spot. We're saying "for all c in the list of st, …"

What does **foo** do? It filters the list to collect the uppercase letters.

foo applied to a string will return all uppercase letters.

There's a thing known as "type checking" in Haskell. What this means is that you have some Haskell expression, and there is a claim that it is a certain type. What type checking does is that it checks whether the expression has that type. It's effectively a logical statement.

```
("hey",True) :: (String,Bool) -- Yes!
("hey",True) :: (String, Char) -- No!
```

With type checking, you're given the type to check.

Type Inference: Given an expression, compute a type if it exists.

("hey", True) -~> (String, Bool) -- "-~>" is supposed to represent a squiggly arrow.

"hey" + 99 -~> error!

Static Types: A type system for which the types of expressions are known at compile-time. I.e., the type of every expression is known by inspecting the code, and not by running it.

Type Variables: Reintroducing what we called "parametric polymorphism"

The following type means that, for all types a and b, the function fst can be applied.

ghci> :t **fst**
fst :: (a,b) -> a -- Given a type of (a,b) as input, we return a out.

"Instances" of (a,b) -> a determine how **fst** can be applied. What does Instance mean? You take a type of a and a type of b and substitute them in there.

(Int, Char) -> Int -- fst (99,'A)
([Char], Float) -> [Char] -- fst ("Hey", 3.14)
(Int -> Int,Bool) -> Int -> Int -- fst (id, True)

All three fst refer to the same code, even though they're used in different contexts and given different data.

Type Classes

There are many predefined classes in Haskell, including Ord, Show, Enum, Num, etc.

Chapter 4:

A **pattern** is anything in the argument position of a function definition. We've seen wildcard patterns, variable patterns, and some simple constructor patterns. Haskell has many more pattern options available.

As previously discussed, _ is the wildcard pattern.

Variable patterns match anything.

Composite Patterns

Patterns can be composed to make bigger patterns, thereby giving you more expressiveness in matching values.

As Patterns

"as" is @.

As patterns don't give you any more expressiveness, but they give you more convenience at times. The slide helps As Patterns make a lot more sense.

Guards in Patterns

It's easy to write a maximum function using if-then-else. However, another way to define the function is with **guards**. It's really another way of writing if-then-else, but it has the flavor of a case expression.

With guards, they can be much more readable, so that's the advantage of using them. It doesn't give additional expressiveness, but it gives a neater way of writing code.

Let definitions

Variables defined in a let or where clause are local. This means that any functions defined in a let and where can only be used within the body of the let/where.

Lets and wheres are really the same thing, it's just about where you put the code.

Case Expressions

General form:

```
case expresssion of pattern -> result
                    pattern -> result
                    pattern -> result
                      . . .
```

Chapter 6 Material: Lambda expressions and Higher-Order Functions

Nameless functions in Haskell

Nameless functions in Haskell

So far, we've given our defined functions a name. This can be cumbersome for simple "knock-off" functions such as "inc" (for incrementing).

There is a notation called "lambda" epressions which allows definition of nameless functions.

Instead of writing your function as **inc x = x + 1**, you can write it as **\ x -> x + 1** as a lambda function.

You can do away entirely with a named declaration **if** the function is not recursive. If it is recursive, it needs to be named.

map is cool because it takes a function and maps it onto another thing (e.g. a list), then returns that thing (for example, a list) after it's been operated on by the function.

The blob "**\ x ->**" is called a lambda binding.

Lambda expressions, together with function application (and other stuff) form a lambda calculus.

Lambda calculus is a basic tool that becomes second nature if you're researching how programming languages work and how to make them.

History: Haskell B. Curry invented lambda calculus in his doctoral thesis.

Why are Higher-order functions useful?

Languages become very powerful when they have higher-order functions.

You can capture patterns as a single higher-order function and use them.

Domain specific lanaguages can be defined as collections of higher-order functions. It's like how Puppet has a DSL for it's infrastructure automation tasks that can only be used with Puppet.

The higher-order library function called **map** applies a function to every element of a list.

map :: (a -> b) -> [a] -> [b]

For example:

> map (+1) [1,3,5,7]

[2,4,6,8]

What is the type of map here? It takes a list of Ints and returns a list of Ints. :: [Int] -> [Int]

The Filter Function

filter is also a higher-order function. It selects every element from a list that satisfies a predicate.

For example:

> filter even [1..10]

[2,4,6,8,10]

Filter can be defined using recursion as well, this is shown in the slides.

Exercise:

Slides 12 and 13 on the Powerpoint.