Haskell Lecture Notes

Zachary May

June 17, 2014

1 Introducing Haskell

- Haskell is a **statically-typed**, **non-strict**, **pure functional** programming language.
  - **Functional:** Conceptually, computation via the application of functions to arguments rather than sequential instructions manipulating values in memory. Functions are first-class values.
  - **Pure:** The functions in question are more like mathematical functions than "procedures". They map values in an input domain to values in an output domain. Pure functions have no "side effects". This is a big win for reasoning about and testing our code.
  - **Non-strict:** By default, Haskell uses a "lazy" evaluation strategy. Expressions do not need to be evaluated until the results are needed. For example, Haskell can cleanly represent infinite lists because the language never tries to fully evaluate it.
  - **Statically-typed:** The type of very expression is known at compile time, preventing run-time errors caused by type incompatibilities. This prevents things like Java's `NullPointerException`, because a function that claims it returns a value of a specific type has to live up to that promise and returning the Haskell equivalent of `null` is a compile-time type error. Additionally, Haskell makes use of a technique called *type inference* to figure out the types of most things without needing explicit type annotations.

2 A Sample Program

```
1 sumSquares :: Integer -> [Integer] -> Integer
2 sumSquares count numbers =
3     sum (take count (map (^2) numbers))
4
5 printSquares :: IO ()
6 printSquares =
7    print $ sumSquares 10 [1..]
```

- The type signature in line 1 describes a function with two arguments: an `Integer` and a list of `Integer`s. It returns an `Integer`. In general, the type after the last `->` is the return value, and the others are the arguments.
- Haskell's type inference would actually figure out a more general type than what we provided in the type annotation on line 1. Haskell programmers usually include type annotations as a form of machine-checked documentation.
- Haskell functions are defined with an equational syntax as seen in lines 2-3: `sumSquares` applied to the arguments `count` and `numbers` is equal to the expression on the right-hand side of the equation.
- Line 3 shows us several examples of *function application*. The parentheses here are for grouping only. The notation for function application is very lightweight.
- Working from the inside out, `map` applies a function to each element in a list, producing a new new list containing the resulting values.
- `(^2)` is called a *section*, a shorthand for an anonymous function whose argument "fills in the blank" for a binary operator, exponentiation in this case. So `map (^2)` transforms a list of numbers into their squares.
- Given a number n and a list, `take` returns the first n items of the list, or the whole list if it has fewer than n elements. `sum` takes a list of numbers and returns the sum.
- Lines 5-7 define another function, `printSquares`. It takes no arguments, and its return type, `IO ()`, is quite interesting.
- Look back at the type `[Integer]`. The list type `[]` is a *parameterized* type. That is, "list" itself is not a concrete type, but a *type construc-*

tor. We need another type, the parameter, to create a concrete type. Similarly, `IO` is a type constructor.

- Here we instantiate it with the type `()`, the *unit type* with only a single instance, also written as `()`. The type `IO ()` represents an *I/O action* with no "return" value. The `IO` type constructor is the technique Haskell uses to perform I/O, inherently impure, in a world of pure functions, via a far more general abstraction called the *monad*.
- Consider the expression `[1..]`. This expression represents the infinite list of positive integers. Because Haskell is lazy, though, merely describing the value is not enough to force the run-time try evaluate the entire thing. In fact, in this case, evaluation of `[1..]` is only forced by `print` asking for the value produced by `sum`, which demands the values yielded by `take count`, and so on, on demand.
- Notice the `$` in line 7. Pronounced "applied to", it is an operator with extremely low precedence that breaks up the very high precedence of function application, allowing us to avoid nesting parentheses. We could have defined `sumSquares` using `$` as: `sum $ take count $ map (^2) numbers`.