Haskell Lecture Notes

Zachary May

June 22, 2014

1 Basic Haskell Types

In this section, we take a deeper look at Haskell's type system.

- Haskell offers the some basic primitive data types we would expect: [1]
  - `Int` and `Integer`: machine-sized and arbitrary precision integers, respectively.
  - `Float` and `Double`: single- and double-precision floating point numbers.
  - `Char`: Single Unicode characters.
  - `Bool`: Boolean values (but see below, `Bool` is actually a composite type).
- Additionally, we have several composite data types:
  - `cons` lists: Lisp-style linked lists. The type is written `[a]` where `a` is another type.
  - `String`: character strings; `String` is literally just a *type synonym* for `[Char]`.
  - Tuples: k-element tuples, $k \geq 2$. The type of an n-element tuple is `(a1, a2, ..., an)` where `a1`, `a2`, ..., and `an` are other types.
- New data types are introduced with the `data` keyword.
- Type synonyms can be introduced with the `type` keyword. `type EmailAddress = String` says that the identifier `EmailAddress` can be used interchangeably with the type `String`. The advantage is that `EmailAddress` is more descriptive.
- The keyword `newtype` creates a more controlled sort of type synonym.

  - If we wanted a type to describe e-mail address values but did not want it to be interchangeable with `String`s in general, we could define a new type that simply "tags" a `String` value: `data EmailAddress = EmailAddress String`.
  - This comes with some amount of overhead each time we want to "unwrap" the `EmailAddress` and get at the underlying `String`.
  - Instead we can use `newtype EmailAddress = EmailAddress String`. Haskell's type checker treats this exactly like type introduced with `data`, but drops the tagging for purposes of code generation, eliminating the overhead required to "unwrap" the `String`.

- Although Haskell can *infer* the types of most all expressions, types can be stated explicitly with a type annotation using `::`. For example:

```
1 nothing :: [String]
2 nothing = []
3
4 moreNothing = []
```

  - We define two values, `nothing` and `moreNothing`.
  - Although the equational definitions are identical, we explicitly define the type of `nothing` to be of type `[String]`, a list of strings, with the type annotation on line 1.
  - Without an explicit type annotation, Haskell will infer the type of `moreNothing`, in this case, the more general type `[a]`. (This is a *polymorphic type* which we will discuss later.)

- Haskell's lists and tuples are specific examples of the language's *algebraic type system*.
- Algebraic data types were introduced in the Hope programming language in 1980. [3]
- An algebraic type system generally offers two sorts of types:

  - *Product types*: A data type with one or more fields.
    - Tuples are the archetypal product type.
    - The "size" of a product type is the product of the sizes of the types of its fields.
    - E.g., `(Bool, Bool)`, the type of 2-tuples of two Boolean values, has a total of $2 \cdot 2 = 4$ possible values.

- An example:

```
data DimensionalValue =
    DimensionalValue Float Dimension
```

  - `DimensionalValue` represents a `Float` value tagged with a unit of measure of type `Dimension`. We will see later how we might describe that type.
  - The `data` keyword introduces a data type definition.
  - The identifier before the `=` is the name of the new type.
  - After the `=`, is the types constructor definition. The first identifier is the *data constructor*, followed by arbitrarily many field declarations.
  - Our `DimensionalValue` type is equivalent to a tuple `(Float, Dimension)`, but we have given it a distinct and descriptive name.

- *Sum types*: A data type with one or more alternatives.
  - Enumerations are the archetypal sum types
  - An example:

```
data Dimension = Seconds
               | Meters
               | Newtons
```

    - As before `data` introduces a new type, here named `Dimension`.
    - The vertical pipe `|` separates the various alternatives.
    - Each alternative is given as a constructor definition as described above. Here we define three simple type constructors, `Seconds`, `Meters`, and `Newtons`.
    - These identifiers can be used as literal values of the type `Dimension`.
  - The "size" of a sum type is the sum of the sizes of the types of its alternatives.
  - The type `Dimension` has $1 + 1 + 1 = 3$ possible values.
  - Haskell's `Bool` data type is defined as a sum type with data constructors `True` and `False`.

- The power of algebraic data types comes when we combine the two: sums of products and products of sums:

```
data PlaneTicket
```

```
        = PlaneTicket Section MealOption

data Section = FirstClass
             | Business
             | Coach

data MealOption = Regular
                | Vegetarian

data TravelDetails = Train
                   | Automobile
                   | Plane PlaneTicket
```

- Here we define several types that might describe the domain model of a travel agency application.
- `PlaneTicket` is a product type over two sum types: the section (`FirstClass` or `Coach`) and the meal option, (`Regular` or `Vegetarian`).
- `TravelDetails` is a sum type over two singleton data constructors `Train` and `Automobile` and a unary product alternative that tags `PlaneTicket` details with the data constructor `Plane`
- How many possible values are there for the `TravelDetails` type? $1 + 1 + (3 \cdot 2) = 8$.

2 Polymorphic types

- Earlier, we described the list and tuple types in terms of other, unspecified data types:
  - `[a]` is the type of lists with elements of some type `a`.
  - `(a, b)` is the type of 2-tuples with first element of some type `a` and second element of some type `b`.
- Here, `a` and `b` are *type variables*.
- Lexically, type variables must begin with a lowercase letter. Concrete data types (in addition to data constructors) must begin with an uppercase letter.
- Data types that contain type variables are called *polymorphic types*.

- This type of polymorphism is known as *parametric polymorphism*: substituting the concrete type `Char` for the *type parameter* `a` in `[a]` gives the concrete type `[Char]`.
- Parametric polymorphism is distinct from the *inclusion polymorphism* seen in object-oriented programming.
- This example shows how we might implement our own `cons`-list and 2-tuple types:

```haskell
data List a = Nil
            | Cons a (List a)

data Pair a b = Pair a b

data OtherPair a = OtherPair a a
```

- Introducing type variables on the left-hand side of the `=` indicates that we are defining a polymorphic types. `List` is parametric in a single type variable `a` and `Pair` is parametric in two type variables, `a` and `b`.
- The two type variables called `a` in the definitions of `List` and `Pair` are distinct.
- What is the difference between our definition of `Pair` and `OtherPair`? `OtherPair` is parametric in only one type variable so both of its elements must be of the same type.
- We see that `List` is a a "sum of products": A `List` of `a`s is either the empty list `Nil` *or* it is a value of type `a` followed by another `List` of `a`s. Thus, `List a` is a recursively-defined data type.
- Let us also make a distinction here between:
  - a *concrete type*, like `List Integer` or `(String, Dimension)` that has no type variables;
  - a *polymorphic type* like `List a` that has one or more type variables;
  - a *type constructor* like `List` that, if "applied" to a concrete type, yields concrete type, and if "applied" to a type variable yields a polymorphic type.
    - Type constructors are distinct from, but analagous to, data constructors.

- A data constructor with fields, when applied to values to populate those fields, yields a value of the type associated with that data constructor.
- A type constructor that admits type variables, when applied to types to instantiate those type variables, yields an instantiation of the associated polymorphic type.

3 Function Types

- The examples we have looked at so far are for the types of values. However, Haskell supports *first-class functions*: functions can be passed as parameters into functions and be be returned as the result of a function.
- That is to say, in Haskell, functions *are* values. So how do we describe their types?
- First, we never actually define new function types with `data`, although we can define synonyms for function types with `type`.
- The one true function type constructor is `->`, as in `a -> b`, the polymorphic type of functions with domain `a` and co-domain `b`.
  - What does the function type `a -> a` represent? Functions with identical domain and co-domain.
  - With no other information about the type `a`, what sort of function can have the type `a -> a`? The identity function.
- The functions described by `->` appear to only have one parameter, the type on the left of the `->`. Haskell has operations (read: functions) like addition that take two parameters, so how can we describe the type of such a function?
- Recall that functions can return other functions as their result. Haskell models multi-parameter functions with single parameter functions that return a new function ready to consume more parameters. This technique is called *currying*, named for the logician Haskell Curry.

```
add x = \y -> x + y
```

  - We define `add` as a function that takes a single parameter `x`.
  - It returns an anonymous function, introduced by (meant to suggest the Greek λ). It's parameter is called `y`. The result of this anonymous function is the sum of `x + y`.

  - When calling `add`, the actual parameter provided for the formal parameter `x` is preserved in a *closure* that, along with the body of the anonymous function, makes up the function value we return.

- Haskell does not actually inconvenience us by requiring this notation. We can just define `add` as:

```
add x y = x + y
```

- However, Haskell really is using currying under the hood. As such, we can *partially apply* functions. Even with the simple definition, `add 5` is not an error, it returns a function value ready to accept another argument and add it to 5.
- Now the type of `add` should be more clear. Assuming we are only adding `Integers`, it must be `Integer -> (Integer -> Integer)`.
- `->` is right associative, so we can simplify this to just `Integer -> Integer -> Integer`.
- In this form, we can view the type after the last `->` as the return type of the function and all the other types as the types of the function's parameters.
- We still need parentheses for grouping if one of the parameters is a function:
  - Consider the function `map :: (a -> b) -> [a] -> [b]`.
  - What are the types of the parameters and return value of `map`? The first parameter is a function with domain `a` and co-domainb. The second parameter is a list of `a`s. The result is a list of `b`s.
  - How is that different from `map' :: a -> b -> [a] -> [b]`? `map'` takes three parameters (an `a`, a `b`, and a list of `a`s) and returns a list of `b`s.
  - To what extent can you infer the semantics of `map` *from its type alone*?
- In general, we call functions that have one or more functions as their parameters or that return functions as their result *higher-order functions*. As we will see, they central to more advanced techniques in functional programming.

References

[1] Simon Peyton Jones, et al., *Haskell 98 Language and Libraries: The Revised Report*. `http://www.haskell.org/onlinereport/index.html`, 2002.

[2] Paul Hudak, et al., *A History of Haskell: Being Lazy With Class*. `http://www.scs.stanford.edu/ dbg/readings/haskell-history.pdf`, 2007.

[3] R.M. Burstall, D.B. MacQueen, D.T. Sannella, *Hope: An Experimental Applicative Language* `http://homepages.inf.ed.ac.uk/dts/pub/hope.pdf`, 1980.